Server-side Development – Assignment #4

Author: Lorraine Figueroa (nyguerrillagirl@brainycode.com)

In this assignment, you will be extending the router to support the ability to save and retrieve a list of favorite dishes by each of the registered users. All registered users in the system should have the ability to save any dish as their favorite dish, retrieve all their favorite dishes and remove one or all their favorite dishes.

Step-By-Step Assignment Instructions

less

Assignment Overview

At the end of this assignment, your should have completed the following:

- Allowed users to select a dish as their favorite, and add it to the list of favorites that are saved on the server.
- Allowed users to retrieve the list of their favorite dishes from the server
- Delete one or all of their favorite dishes from their favorites list on the server.

Assignment Resources

Table 1 - db.json

```
{
  "dishes": [
    {
      "name": "Uthappizza",
      "image": "images/uthappizza.png",
      "category": "mains",
      "label": "Hot",
      "price": "4.99",
      "featured": "true",
      "description": "A unique combination of Indian Uthappam (pancake) and Italian pizza, topped with
Cerignola olives, ripe vine cherry tomatoes, Vidalia onion, Guntur chillies and Buffalo Paneer.",
      "comments": [
        {
          "rating": 5,
          "comment": "Imagine all the eatables, living in conFusion!",
          "author": "John Lemon",
          "date": "2012-10-16T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Sends anyone to heaven, I wish I could get my mother-in-law to eat it!",
          "author": "Paul McVites",
          "date": "2014-09-05T17:57:28.556094Z"
        },
        {
          "rating": 3,
          "comment": "Eat it, just eat it!",
          "author": "Michael Jaikishan",
          "date": "2015-02-13T17:57:28.556094Z"
```

```
        },
        {
          "rating": 4,
          "comment": "Ultimate, Reaching for the stars!",
          "author": "Ringo Starry",
          "date": "2013-12-02T17:57:28.556094Z"
        },
        {
          "rating": 2,
          "comment": "It's your birthday, we're gonna party!",
          "author": "25 Cent",
          "date": "2011-12-02T17:57:28.556094Z"
        }
      ]
    },
    {
      "name": "Zucchipakoda",
      "image": "images/zucchipakoda.png",
      "category": "appetizer",
      "label": "",
      "price": "1.99",
      "featured": "false",
      "description": "Deep fried Zucchini coated with mildly spiced Chickpea flour batter accompanied
with a sweet-tangy tamarind sauce",
      "comments": [
        {
          "rating": 5,
          "comment": "Imagine all the eatables, living in conFusion!",
          "author": "John Lemon",
          "date": "2012-10-16T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Sends anyone to heaven, I wish I could get my mother-in-law to eat it!",
          "author": "Paul McVites",
          "date": "2014-09-05T17:57:28.556094Z"
        },
        {
          "rating": 3,
          "comment": "Eat it, just eat it!",
          "author": "Michael Jaikishan",
          "date": "2015-02-13T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Ultimate, Reaching for the stars!",
          "author": "Ringo Starry",
          "date": "2013-12-02T17:57:28.556094Z"
        },
        {
          "rating": 2,
          "comment": "It's your birthday, we're gonna party!",
          "author": "25 Cent",
          "date": "2011-12-02T17:57:28.556094Z"
        }
      ]
    },
    {
      "name": "Vadonut",
      "image": "images/vadonut.png",
      "category": "appetizer",
      "label": "New",
      "price": "1.99",
      "featured": "false",
      "description": "A quintessential ConFusion experience, is it a vada or is it a donut?",
      "comments": [
        {
          "rating": 5,
          "comment": "Imagine all the eatables, living in conFusion!",
```

```
          "author": "John Lemon",
          "date": "2012-10-16T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Sends anyone to heaven, I wish I could get my mother-in-law to eat it!",
          "author": "Paul McVites",
          "date": "2014-09-05T17:57:28.556094Z"
        },
        {
          "rating": 3,
          "comment": "Eat it, just eat it!",
          "author": "Michael Jaikishan",
          "date": "2015-02-13T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Ultimate, Reaching for the stars!",
          "author": "Ringo Starry",
          "date": "2013-12-02T17:57:28.556094Z"
        },
        {
          "rating": 2,
          "comment": "It's your birthday, we're gonna party!",
          "author": "25 Cent",
          "date": "2011-12-02T17:57:28.556094Z"
        }
      ]
    },
    {
      "name": "ElaiCheese Cake",
      "image": "images/elaicheesecake.png",
      "category": "dessert",
      "label": "",
      "price": "2.99",
      "featured": "false",
      "description": "A delectable, semi-sweet New York Style Cheese Cake, with Graham cracker crust
and spiced with Indian cardamoms",
      "comments": [
        {
          "rating": 5,
          "comment": "Imagine all the eatables, living in conFusion!",
          "author": "John Lemon",
          "date": "2012-10-16T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Sends anyone to heaven, I wish I could get my mother-in-law to eat it!",
          "author": "Paul McVites",
          "date": "2014-09-05T17:57:28.556094Z"
        },
        {
          "rating": 3,
          "comment": "Eat it, just eat it!",
          "author": "Michael Jaikishan",
          "date": "2015-02-13T17:57:28.556094Z"
        },
        {
          "rating": 4,
          "comment": "Ultimate, Reaching for the stars!",
          "author": "Ringo Starry",
          "date": "2013-12-02T17:57:28.556094Z"
        },
        {
          "rating": 2,
          "comment": "It's your birthday, we're gonna party!",
          "author": "25 Cent",
          "date": "2011-12-02T17:57:28.556094Z"
        }
```

```
      ]
    }
  ],
  "promotions": [
    {
      "name": "Weekend Grand Buffet",
      "image": "images/buffet.png",
      "label": "New",
      "price": "19.99",
      "featured": "true",
      "description": "Featuring mouthwatering combinations with a choice of five different salads, six
enticing appetizers, six main entrees and five choicest desserts. Free flowing bubbly and soft drinks.
All for just $19.99 per person "
    }
  ],
  "leaders": [
    {
      "name": "Peter Pan",
      "image": "images/alberto.png",
      "designation": "Chief Epicurious Officer",
      "abbr": "CEO",
      "featured": "false",
      "description": "Our CEO, Peter, credits his hardworking East Asian immigrant parents who
undertook the arduous journey to the shores of America with the intention of giving their children the
best future. His mother's wizardy in the kitchen whipping up the tastiest dishes with whatever is
available inexpensively at the supermarket, was his first inspiration to create the fusion cuisines for
which The Frying Pan became well known. He brings his zeal for fusion cuisines to this restaurant,
pioneering cross-cultural culinary connections."
    },
    {
      "name": "Dhanasekaran Witherspoon",
      "image": "images/alberto.png",
      "designation": "Chief Food Officer",
      "abbr": "CFO",
      "featured": "false",
      "description": "Our CFO, Danny, as he is affectionately referred to by his colleagues, comes from
a long established family tradition in farming and produce. His experiences growing up on a farm in the
Australian outback gave him great appreciation for varieties of food sources. As he puts it in his own
words, Everything that runs, wins, and everything that stays, pays!"
    },
    {
      "name": "Agumbe Tang",
      "image": "images/alberto.png",
      "designation": "Chief Taste Officer",
      "abbr": "CTO",
      "featured": "false",
      "description": "Blessed with the most discerning gustatory sense, Agumbe, our CFO, personally
ensures that every dish that we serve meets his exacting tastes. Our chefs dread the tongue lashing
that ensues if their dish does not meet his exacting standards. He lives by his motto, You click only
if you survive my lick."
    },
    {
      "name": "Alberto Somayya",
      "image": "images/alberto.png",
      "designation": "Executive Chef",
      "abbr": "EC",
      "featured": "true",
      "description": "Award winning three-star Michelin chef with wide International experience having
worked closely with whos-who in the culinary world, he specializes in creating mouthwatering Indo-
Italian fusion experiences. He says, Put together the cuisines from the two craziest cultures, and you
get a winning hit! Amma Mia!"
    }
  ],
  "feedback": []
}
```

Assignment Requirements

In this assignment, you will be supporting a new route https://localhost:3443/favorites, where the users can do a GET to retrieve all their favorite dishes, a POST to add a list of dishes to their favorites, and a DELETE to delete the list of their favorites. In addition, you will support the new route https://localhost:3443/favorites/:dishId where the users can issue a POST request to add the dish to their list of favorite dishes, and a DELETE request to delete the specific dish from the list of their favorite dishes.

This assignment consists of the following three tasks:

Task 1

In this task you will be implementing a new Mongoose schema named *favoriteSchema*, and a model named *Favorites* in the file named f*avorite.js* in the *models* folder. This schema should take advantage of the mongoose population support to populate the information about the user and the list of dishes when the user does a GET operation.

Task 2

In this task, you will implement the Express router() for the '/favorites' URI such that you support GET, POST and DELETE operations

- When the user does a GET operation on '/favorites', you will populate the user information and the dishes information before returning the favorites to the user.
- When the user does a POST operation on '/favorites' by including [{"_id":"dish ObjectId"}, . . ., {"_id":"dish ObjectId"}] in the body of the message, you will (a) create a favorite document if such a document corresponding to this user does not already exist in the system, (b) add the dishes specified in the body of the message to the list of favorite dishes for the user, if the dishes do not already exists in the list of favorites.
- When the user performs a DELETE operation on '/favorites', you will delete the list of favorites corresponding to the user, by deleting the favorite document corresponding to this user from the collection.
- When the user performs a POST operation on '/favorites/:dishId', then you will add the specified dish to the list of the user's list of favorite dishes, if the dish is not already in the list of favorite dishes.
- When the user performs a DELETE operation on '/favorites/:dishId', then you will remove the specified dish from the list of the user's list of favorite dishes.

Task 3

You will update app.js to support the new '/favorites' route.

Review criteria

Your assignment will be graded on the basis of the following review criteria:

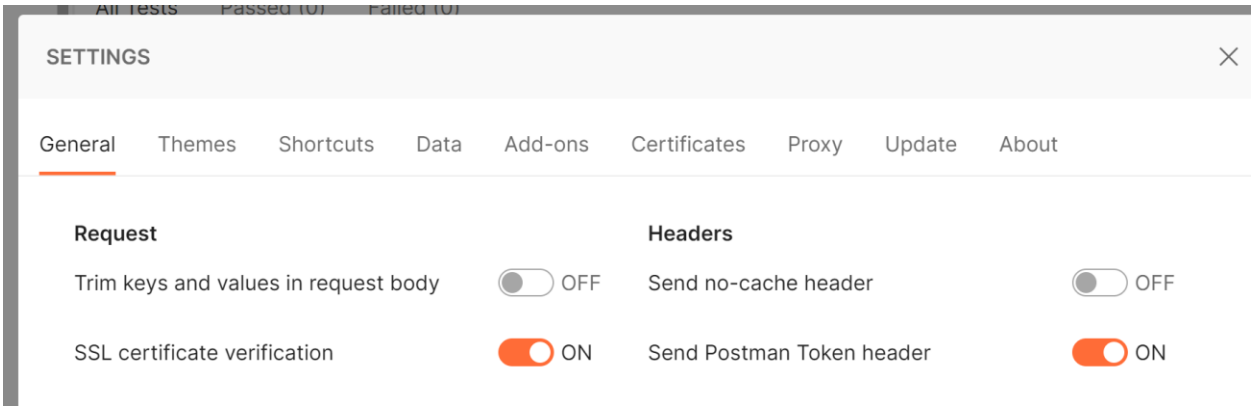
- A new favoriteSchema and Favorites model has been correctly implemented to take advantage of Mongoose Population support to track the users and the list of favorite dishes using their ObjectIds in the favoriteSchema and Favorites model.
- The GET, POST and DELETE operations are well supported as per the specifications above
- The app.js has been updated to support the new route.

Work For Assignment

- Create a new branch for this assignment. COURSERA_SS_ASSIGNMENT_04
- Added branch to remote
- Start mongodb for this project

In order to make sure I am starting from a stable position from assignment #3 left off I will test the previous assignment's collection. Due to the changes made in week #4 I have to make sure to

- Update the {{baseUrl}} and turn off SSL certificate verification



We will use the {{baseUrl}} throughout the test cases

- Create new Postman collection for this assignment:

Note: The only assumption that I will make is that the database has an admin user with the username of admin and password of 'password'. The collection ConfusionServerJWTAssignment04 has the following variables defined (at the collection level):

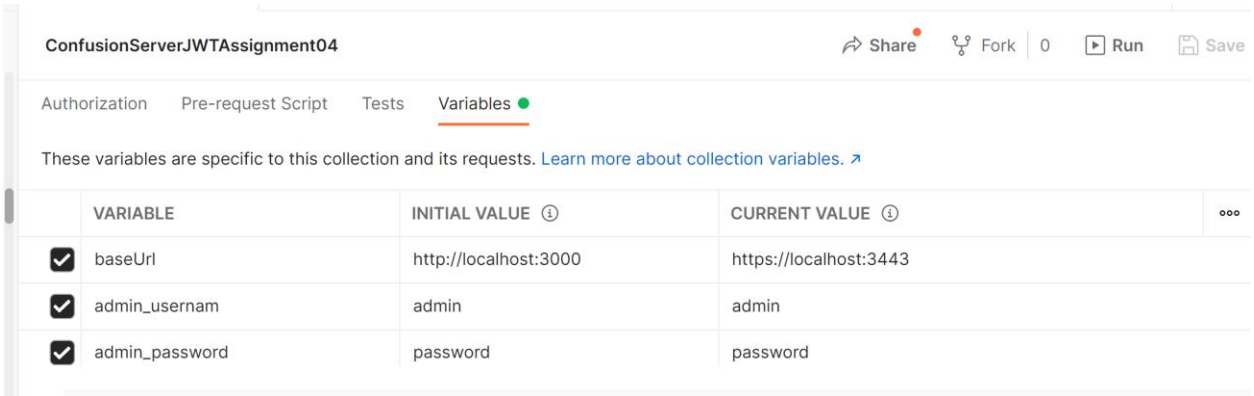


Figure 1 - Collection Variables

We shall update the first request of the collection to be one that inserts the four dishes and populates a test array for each one.

The first new request "Adds All Dishes Request" shall send a request for each dish in the dishes array.

When a dish is posted (by the ADMIN) the response is:

Table 2 - Example RESPONSE from POST a DISH


```
{
    "label": "Hot",
    "featured": true,
    "_id": "61f5bf7decdc8309fc969a9d",
    "name": "Uthappizza",
    "image": "images/uthappizza.png",
    "category": "mains",
    "price": 499,
    "description": "A unique combination of Indian Uthappam (pancake) and Itali
an pizza, topped with Cerignola olives, ripe vine cherry tomatoes, Vidalia onio
n, Guntur chillies and Buffalo Paneer.",
    "comments": [],
    "createdAt": "2022-01-29T22:28:13.786Z",
    "updatedAt": "2022-01-29T22:28:13.786Z",
    "__v": 0
}
```


PROBLEM: Saving each dish was failing!

ANALYSIS: Each dish contained comments that were failing because the users for the comments DID NOT EXIST!

I had to remove all the comments, so I changed things to:

```
{
  "dishes": [
    {
      "name": "Uthappizza",
      "image": "images/uthappizza.png",
      "category": "mains",
      "label": "Hot",
      "price": "4.99",
      "featured": "true",
      "description": "A unique combination of Indian Uthappam (pancake) and Italian pizza, topped with Cerignola olives, ripe vine cherry tomatoes, Vidalia onion, Guntur chillies and Buffalo Paneer.",
      "comments": [ ]
    },
    {
      "name": "Zucchipakoda",
      "image": "images/zucchipakoda.png",
      "category": "appetizer",
      "label": "",
      "price": "1.99",
      "featured": "false",
      "description": "Deep fried Zucchini coated with mildly spiced Chickpea flour batter accompanied with a sweet-tangy tamarind sauce",
      "comments": [ ]
    },
    {
      "name": "Vadonut",
      "image": "images/vadonut.png",
      "category": "appetizer",
      "label": "New",
      "price": "1.99",
      "featured": "false",
      "description": "A quintessential ConFusion experience, is it a vada or is it a donut?",
      "comments": [ ]
    },
    {
      "name": "ElaiCheese Cake",
      "image": "images/elaicheesecake.png",
      "category": "dessert",
      "label": "",
      "price": "2.99",
      "featured": "false",
      "description": "A delectable, semi-sweet New York Style Cheese Cake, with Graham cracker crust and spiced with Indian cardamoms",
      "comments": [ ]
    }
  ]
}
```

Postman Setup Request

Request name: Adds All Dishes Request

Request URL: GET {{baseUrl}}/dishes

Pre-request Script:

```
// Set up the four key dishes
DishObject =
{
  "dishes": [
    {
      "name": "Uthappizza",
      "image": "images/uthappizza.png",
      "category": "mains",
      "label": "Hot",
      "price": "4.99",
      "featured": "true",
      "description": "A unique combination of Indian Uthappam (pancake) and Italian pizza, topped with
Cerignola olives, ripe vine cherry tomatoes, Vidalia onion, Guntur chillies and Buffalo Paneer.",
      "comments": [ ]
    },
    {
      "name": "Zucchipakoda",
      "image": "images/zucchipakoda.png",
      "category": "appetizer",
      "label": "",
      "price": "1.99",
      "featured": "false",
      "description": "Deep fried Zucchini coated with mildly spiced Chickpea flour batter accompanied w
ith a sweet-tangy tamarind sauce",
      "comments": [ ]
    },
    {
      "name": "Vadonut",
      "image": "images/vadonut.png",
      "category": "appetizer",
      "label": "New",
      "price": "1.99",
      "featured": "false",
      "description": "A quintessential ConFusion experience, is it a vada or is it a donut?",
```

```
      "comments": [ ]
    },
    {
      "name": "ElaiCheese Cake",
      "image": "images/elaicheesecake.png",
      "category": "dessert",
      "label": "",
      "price": "2.99",
      "featured": "false",
      "description": "A delectable, semi-
sweet New York Style Cheese Cake, with Graham cracker crust and spiced with Indian cardamoms",
      "comments": [ ]
    }
  ]
};

// Holds key: name and value: _id for each dish created
let dishNameList = new Object();
pm.environment.set('dishNameList', dishNameList);
// Holds key: _id and value name for each dish
let dishIdList = new Object();
pm.environment.set('dishIdList', dishIdList);
// obtain the adminToken
const adminToken = 'Bearer ' + pm.environment.get('admin_token');
const base_url = pm.collectionVariables.get("baseUrl");
for (let i=0; i < DishObject.dishes.length; i++) {
    console.log("===> dish name: ", DishObject.dishes[i].name);
    let newDish = DishObject.dishes[i];
    // Send request to insert dish i into database
    pm.sendRequest({
        url: `${base_url}/dishes`,
        method:'POST',
        header: {'Content-Type' : 'application/json', 'Authorization': adminToken},
        body : {mode: 'raw', raw: newDish}},
        function(err, response) {
        if (err) {
            console.log("===> Callback: error");
            console.log(err);
        } else if (pm.expect(response.status).to.eql('OK')) {
            console.log("===> Callback success");
            var jsonData = response.json();
```

```
            // Now populate the global dicts
            let dishNameListDict = pm.environment.get('dishNameList');
            dishNameListDict[newDish.name] = jsonData._id;
            pm.environment.set('dishNameList', dishNameListDict);
            let dishIdListDict = pm.environment.get('dishIdList');
            dishIdListDict[jsonData._id] = newDish.name;
            pm.environment.set('dishIdList',dishIdListDict);
        } else {
            console.log("===> Callback: not OK");
            console.log(pm.response);
        }
    });
}
```

The pre-request script sends a request as ADMIN to POST each dish in the for loop. Once the dish is added it is also added to two object acting as dictionaries:

- dishNameList – holds each dishName:dishId
- dishIdList – holds each dishId: dishName

The above dictionaries will allow me to randomly select dishes to make as favorites and to compare the result for a user.

Note: In addition each registered user shall hold a registered_user_dishList (with only ids) so I can check against results.

Test Script:

```
pm.test("CHECK_ALL_FOUR_DISHES_ADDED_1", function () {
    pm.response.to.have.status(200);
    const jsonData = pm.response.json();
    // Check

    const dishNameList = pm.environment.get('dishNameList');
    let keys = Object.keys(dishNameList);
    console.log("===> dishNameList keys: \n", keys);

    const dishIdList = pm.environment.get('dishIdList');
    keys = Object.keys(dishIdList);
    console.log("===> dishIdList keys:\n", keys);
    // Make sure all the dishes were set up
    pm.expect(jsonData.length).eql(4);
});
```

The test checks that the pre-request script successfully added all 4 dishes.

The above will be the first test case in the collection. I plan on running the following test cases:

1. Set up 4 dishes into the mongodb database
2. Register and Login User 1
3. Register and Login User 2
4. For user 1 add dish #1 as a favorite using POST {{baseUrl}}/favorites/{{dishId1}}
5. For user 1 do a GET {{baseUrl}}/favorites
6. For user 2 add dish #2 as a favorite using POST {{baseUrl}}/favorites/{{dishId2}}
7. For user 1 add dishes 2 and 3 using the body {{baseUrl}}/favorites
8. For user 1 do a GET {{baseUrl}}/favorites
9. For user 2 add dish #2 again as a favorite – Error (repeat entry)
10. For user 2 add a non-existing dish as a favorite – (Unfound resource)
11. For user 2 delete all favorites using DELETE {{baseUr}}/favorites/
12. For user 1 delete all favorites using {{baseUrl}}/favorites/
13. For ANY user PUT operation is not supported

Task #1

- Create the file in the models folder named favorites.js

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const favoriteSchema = new Schema( {
      user: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
    },
    dishes: [mongoose.Schema.Types.ObjectId]
});


var Favorites = mongoose.model('Favorite', favoriteSchema);

module.exports = Favorites;
```

Task #2

What does a user object in the req look like?

- Created favoriteRouter.js file in the routes folder
- Only implemented /favorites

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const authenticate = require('../authenticate');
const cors = require('./cors');

const Favorites = require('../models/favorites');

const favoriteRouter = express.Router();

favoriteRouter.use(bodyParser.json());

favoriteRouter.route('/')
.options(cors.corsWithOptions, (req, res) => { res.sendStatus(200)})
.get(cors.cors, authenticate.verifyUser, (req,res,next) => {
    console.log("favorites ===> ", req.user);
    Favorites.find({})
    .then((favorites) => {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'application/json');
        res.json(favorites);
    }, (err) => next(err))
    .catch((err) => next(err));

 })
```

The rest of the commands return

```
   res.statusCode = 403;
    res.end('XXX operation not supported on /favorites');
```

- Edit app.js for this new routing:

```
var favoriteRouter = require('./routes/favoriteRouter');
```

And

```
app.use('/favorites', favoriteRouter);
```

- Test GET /favorites with Postman

```
JWT payload:  { _id: '61f5a35aecdc8309fc969a9b',
  iat: 1643551742,  exp: 1643555342 }
/favorites ===>  { firstname: 'James',
  lastname: 'Gosling',
  admin: false,
  _id: 61f5a35aecdc8309fc969a9b,
  username: 'coursera_registered_logged_student1.1643488090616',
  __v: 0 }
```

I need to use req.user._id to filter the get so that ONLY requests pertaining to the current user is returned.

I cannot really test the GET /favorites for a user until I can POST

GET /favorites/{{dishId}}

This operation is not supported. I do not see it specified in the instructions.

Code:

```
.get(cors.cors,(req, res,next) => {
    res.statusCode = 403;
    res.end('GET operation not supported on /favorites/'+ req.params.dishId);

})
```

Postman test case successful

POST /favorites/{{dishId}}

I thought this was easier to process than a body that requires additional logic (to make none of the entries are already IN the existing list).

- Update GET /favorites to populate user and dish details in the request:

```
.get(cors.cors, authenticate.verifyUser, (req,res,next) => {
    console.log("favorites ===> ", req.user);
    Favorites.find({user:req.user._id})
    .populate('user')
```

```
    .populate('dishes')
    .then((favorites) => {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'application/json');
        res.json(favorites);
    }, (err) => next(err))
    .catch((err) => next(err));

 })
```

- Fixed the favorites.js model

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const favoriteSchema = new Schema( {
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
},
dishes: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Dish'}]

});


var Favorites = mongoose.model('Favorite', favoriteSchema);

module.exports = Favorites;
```

- Added first code pass for /favorites/{{dishId}}

This works when this is the first entry into a user's favorite dishes

```
.post(cors.corsWithOptions,authenticate.verifyUser, (req, res,next) => {
    // Check if user has any entries (so we can create and/or add to)
    console.log("POST /dishId ===>", req.params.dishId);
    Favorites.find({user:req.user._id})
    .then( (userFavorites) => {
        console.log("POST favorites/dishId ==>", userFavorites);
        if (userFavorites && !userFavorites.length) {
```

```
            console.log("POST /favorites/dishId ===> user does not have any
favorites");
            favoriteItem = {user: req.user._id, dishes:[req.params.dishId]};
            Favorites.create(favoriteItem)
            .then((favorite) => {
                console.log('Favorite Created ', favorite);
                res.statusCode = 200;
                res.setHeader('Content-Type', 'application/json');
                res.json(favorite);
            }, (err) => next(err))
            .catch( (err) => next(err));
        } else {
            console.log("POST /favorites/dishId ===> user has a current list of
favorite");
            res.statusCode = 200;
            res.setHeader('Content-Type', 'application/json');
            res.json(userFavorites);
        }

    }, (err) => next(err))
    .catch((err) => next(err));
 })
```

Response:

{"dishes":["61f6b2e2d3e8d54e04b01670"],"_id":"61f7250b34efda3a2009534d","user":"61f5a35aecdc8309fc969a9b","__v":0}

- Add the side of the else to add dishId to an existing list)

Testing Notes:
I start the Postman tests by following these steps:

1. Start mongodb
2. Open a mongo repl (mongo)
3. Enter: use conFusion;
4. Enter: show collections;
5. Enter: db.favorites.find();
6. If any documents in the collection, remove them all by entering: db.favorites.deleteMany({});

Testing this code:

```
.post(cors.corsWithOptions,authenticate.verifyUser, (req, res,next) => {
```

```
    // Check if user has any entries (so we can create and/or add to)
    console.log("POST /dishId ===>", req.params.dishId);
    Favorites.find({user:req.user._id})
    .then( (userFavorites) => {
        console.log("POST favorites/dishId ==>", userFavorites);
        if (userFavorites && !userFavorites.length) {
            console.log("POST /favorites/dishId ===> user does not have any favorites yet");
            favoriteItem = {user: req.user._id, dishes:[req.params.dishId]};
            Favorites.create(favoriteItem)
            .then((favorite) => {
                console.log('Favorite Created ', favorite);
                res.statusCode = 200;
                res.setHeader('Content-Type', 'application/json');
                res.json(favorite);
            }, (err) => next(err))
            .catch( (err) => next(err));
        } else {
            console.log("POST /favorites/dishId ===> user has a current list of favorite");
            var favoriteEntry = userFavorites[0]; // since always returned in a list
            var newDishId = req.params.dishId;

            // Find out if the newDishId is already a favorite or not
            var dishList = favoriteEntry.dishes;
            if (dishList.indexOf(newDishId) == -1) {
                console.log("===> POST /favorites/dishId - this is a new favorite!");
                dishList.push(newDishId);
                favoriteEntry.dishes = dishList;
                favoriteEntry.save()
                    .then( (favorites) => {
                        res.statusCode = 200;
                        res.setHeader('Content-Type', 'application/json');
                        res.json(favorites);
                    }, (err) => next(err))
                    .catch((err) => next(err));
            } else {
                console.log("===> POST /favorites/dishId - already a favorite!");
                err = new Error('Dish ' + req.params.dishId + ' already a favorite dish.');
                err.status = 400;
                return next(err);
            }
         }

    }, (err) => next(err))
    .catch((err) => next(err));
})
```

All paths tested in Postman

DELETE /favorites/{{dishId}}

There general logic goes like this:

Get current list of favorites for the current user.

If the current list of favorites is empty ➔ send an error

Else if the dishId is NOT in the current list ➔ send an error

Else delete from the dishes list and send back current list

The code:

```
.delete(cors.corsWithOptions,authenticate.verifyUser, (req, res,next) => {
    Favorites.find({user:req.user._id})
        .then((userFavorites) => {
            console.log("==> DELETE favorites/dishId processing ...", userFavorites);
            if (userFavorites && !userFavorites.length) {
                console.log("==> DELETE favorites/dishId no favorites found. Delete in error.");
                err = new Error('Dish ' + req.params.dishId + ' not found as favorite dish.');
                err.status = 404; // not found (even if vacuously true)
                return next(err);
            } else {
                var dishIdToDelete = req.params.dishId;
                // Find the dish in the list
                console.log("==> DELETE favorites/dishId looking for dishId", dishIdToDelete);
                var favoriteEntry = userFavorites[0]; // since always returned in a list
                var dishList = favoriteEntry.dishes;
                var dishLocation = dishList.indexOf(dishIdToDelete);
                if (dishLocation == -1) {
                    console.log("==> DELETE favorites/dishId not found.");
                    err = new Error('Dish ' + req.params.dishId + ' not found as favorite dish.');
                    err.status = 404; // not found in current list
                    return next(err);
                } else {
                    // OK we know it is in the dish list
                    dishList.splice(dishLocation, 1);
                    favoriteEntry.dishes = dishList;
                    favoriteEntry.save()
                        .then( (favorites) => {
                            res.statusCode = 200;
                            res.setHeader('Content-Type', 'application/json');
```

```
                            res.json(favorites);
                        }, (err) => next(err));
                }
            }
        }, (err) => next(err))
        .catch((err) => next(err));
});
```

All Postman test cases passed

POST /favorites

- When the user does a POST operation on '/favorites' by including [{"_id":"dish ObjectId"}, . . ., {"_id":"dish ObjectId"}] in the body of the message, you will (a) create a favorite document if such a document corresponding to this user does not already exist in the system, (b) add the dishes specified in the body of the message to the list of favorite dishes for the user, if the dishes do not already exists in the list of favorites.

So the body if a list of dishes as in:

`[{ “_id”: “dishObjectId”}, …{“_id”:”anotherDishObjectId”}]`

Each dish will be added to user’s existing favorites as long as the dish is NOT found in the existing favorites list

The code logic is the following:

Obtain list of user’s favorites

If user has no favorites then entire list supplied and CREATE new entry

Else for each unique (not already in the list) dishId enter into existing list

Save the updated favorite list for the user

The post code:

```
.post(cors.corsWithOptions,authenticate.verifyUser, (req, res, next) => {
    console.log("===> POST /favorites invoked.");
    Favorites.find({user:req.user._id})
        .then((userFavorites) => {
            if (userFavorites && !userFavorites.length) {
                console.log("===> POST /favorite - no favorites for user yet.");
```

```
                // This user does not have favorites yet
                let dishList = [];
                let userProvidedDishList = req.body;
                for (let i=0; i < userProvidedDishList.length; i++) {
                    console.log("dishItem: ", userProvidedDishList[i]);
                    dishList.push(userProvidedDishList[i]._id);
                }
                let favoriteItem = {user: req.user._id, dishes:dishList};
                Favorites.create(favoriteItem)
                    .then((favorite) => {
                        console.log('Favorite Created ', favorite);
                        res.statusCode = 200;
                        res.setHeader('Content-Type', 'application/json');
                        res.json(favorite);
                    }, (err) => next(err))
                    .catch( (err) => next(err));
            } else {
                console.log("===> POST /favorite - favorites for user found.");
                let favoriteEntry = userFavorites[0]; // get the single entry for user
                let dishList = favoriteEntry.dishes;
                let userProvidedDishList = req.body;
                // For each new dishId provided - if in dishList - skip it, otherwise add
                for (let i=0; i < userProvidedDishList.length; i++) {
                    let newDishItem = userProvidedDishList[i]._id;
                    if (dishList.indexOf(newDishItem) === -1) {
                        // add this item to the dishList
                        dishList.push(newDishItem);
                    }
                }
                // Now let's update the favorites for this user in the db
                favoriteEntry.dishes = dishList;
                favoriteEntry.save()
                    .then( (favorites) => {
                        res.statusCode = 200;
                        res.setHeader('Content-Type', 'application/json');
                        res.json(favorites);
                    }, (err) => next(err))
                    .catch((err) => next(err));
            }
        }, (err) => next(err))
       .catch((err) => next(err));

})
```

I considered combining both sides into one but since one uses favoriteEntry.create and the other favoriteEntry.save I decided to leave things as is.

Postman test cases:

1. Add 2 more to user 1, for a total of 3 dish favorites, note it already had 1 dish favorite
2. Try to add the same two dishes to user 1, no error but repeated dishId's are not added, total still 3
3. Add 2 new dish favorites to user 2 for a total of 2

DELETE /favorites

This operation is pretty straightforward. The test cases will be to remove all of user 1 favorites and all of user 2 favorites.

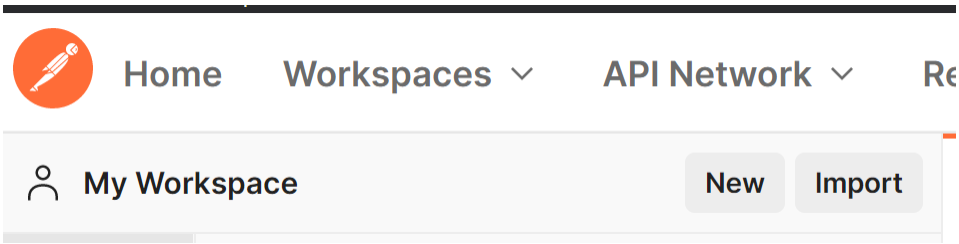
Postman Test Cases

I created the collection: ConfusionServerJWTAssignment04 with the environment ConfusionServerEnv.

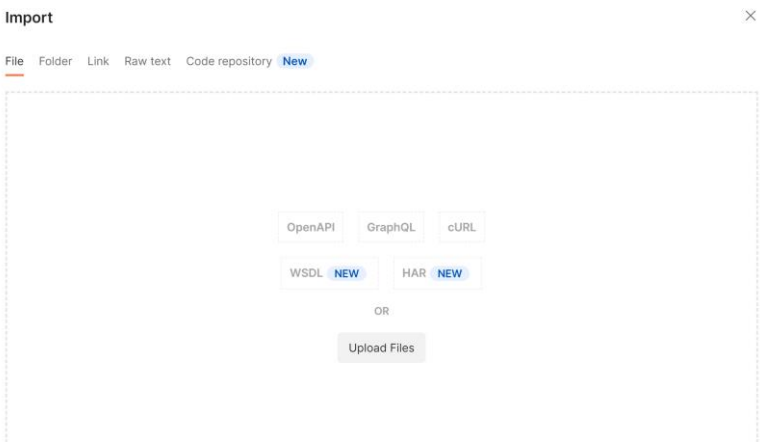
This collection is intended to only be used to test the features added for assignment #4 for the course.

Importing the Collection

1. Click on the Import tab below the top menu

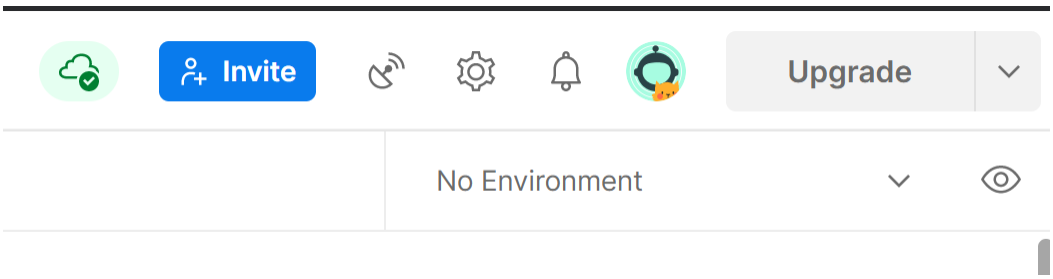


2. Click on Upload Files and select the collection and environment files for import.

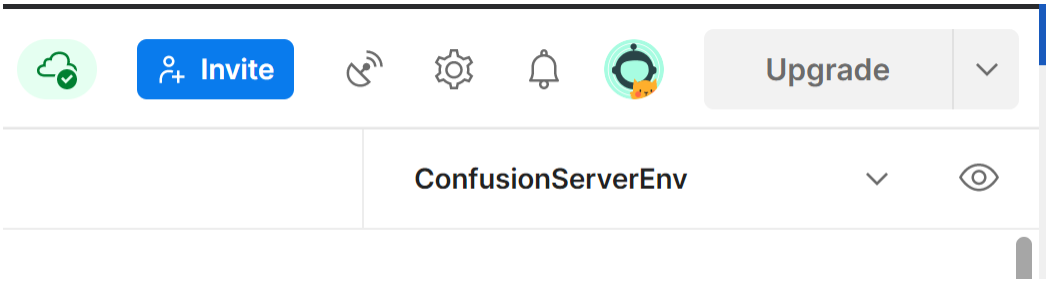


3. Select the collection. In the environment area on the screen select the imported environment from the drop-down list.

BEFORE:



AFTER:



If you click on the eye icon you will see all the imported environment variables this collection (and the other associated assignment collections used for testing).

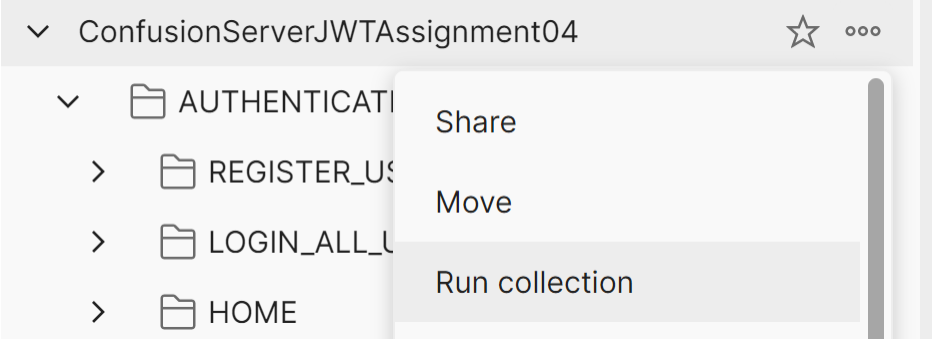
The “current” values will of course be invalid for your Postman collection. That is okay. When you run the collection test request test suite all the variables will be re-set to values that make sense for your mongdb and application settings. (Crossing-fingers!) I try to clean-up the database before and after test cases – except in deleting the randomly created user document. There is a simple way to clear all but the admin user out from your User table via the mongo db repl.

db.users.deleteMany({ "admin": false })

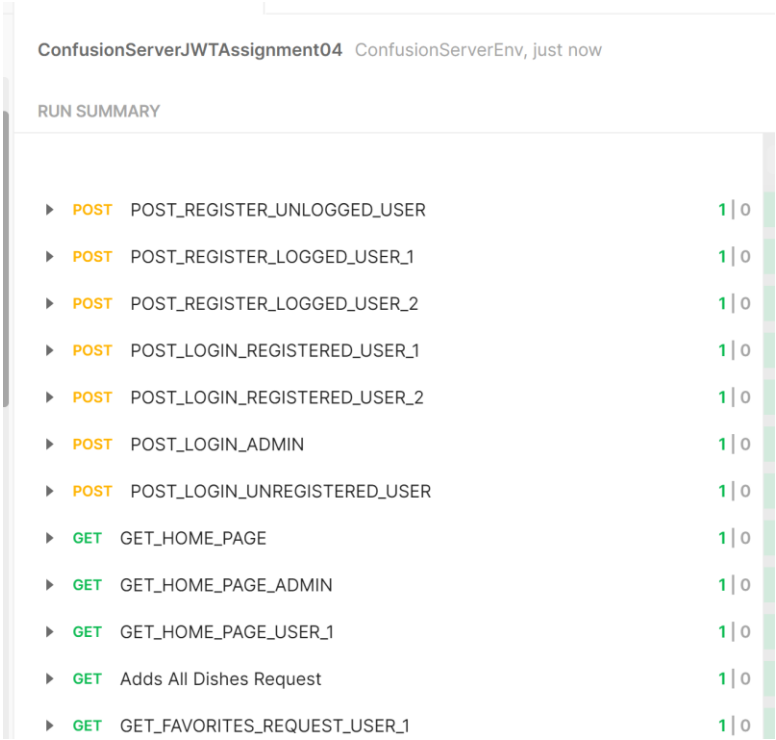
I had over 40 users in my database due to the repeated runs of this and associated collection test suites.

Running the Collection

To run the collection select the collection name and “Run collection” option from the ooo menu.



A successful run of the test cases will show all of them passing:

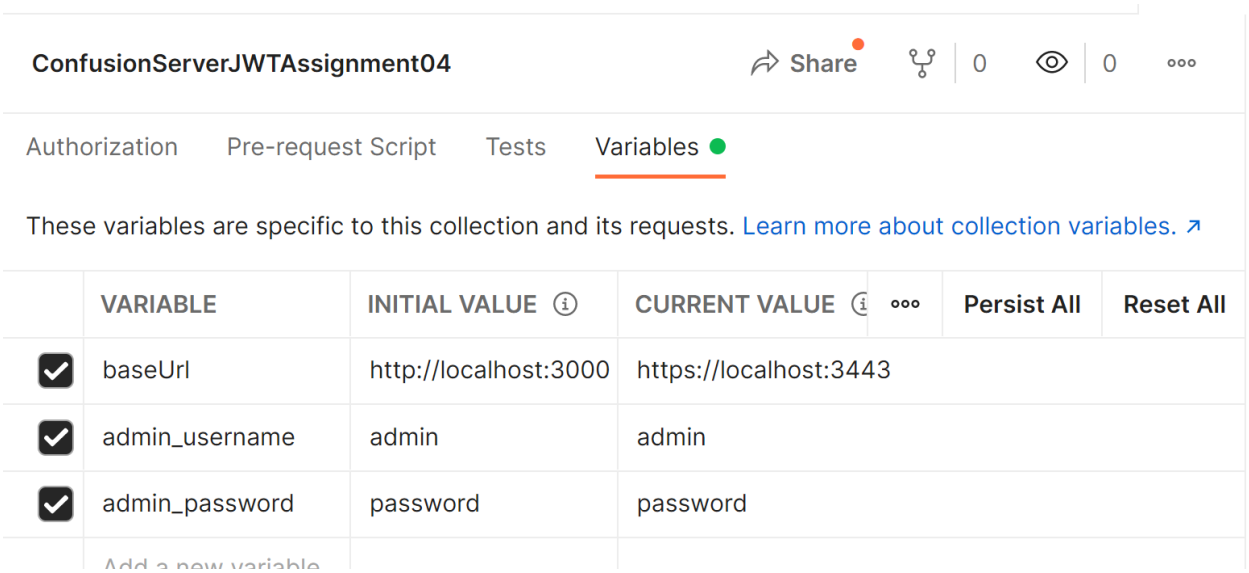


The Details

Running a collection test assumes that the following collection variables are set:

- baseUrl
- admin_username
- admin_password

To make testing easier I assumed all user passwords were ‘password’ but that can be easily changed.



Note the change from http to https for the baseUrl and the assumed admin name. The test case assumes that the admin username and password is already set in mongodb User collection.

The collection has the following folders in the main folder AUTHENTICATION_VERSION:

1. REGISTER_USERS – This folder contains requests that registers three users, where one is NEVER logged in and the other two USER_1 and USER_2 are registered and logged in. This means all regular users are newly created on each test suite run
2. LOGIN_ALL_USERS – This folder contains requests to login USER_1, USER_2 and the ADMIN. Note: It is assumed the tester has created a user that is in the database with the username – admin.
3. HOME – requests to test basic access to the home page (more or less a regression test)
4. SETUP_THE_DISHES – A request to insert into the database 4 dishes (by the admin). This will mean the dishIds is newly created and unknown before the test case is run. We keep track of each dishId in an environment object named dishNameList (and dishIdList[1])
5. FAVORITES – Only tests the GET /favorites
6. FAVORITES_DISHID – This folder contains various test cases for requests /favorites/:dishId
7. FAVORITE_DISHES – This folder contains various test cases for requests /favorites
8. CLEANUP – removes all the dishes

The test suite cleans up ALL records created (if it completes successfully)

What if you find a failed test case?

There is some dependency between test cases (e.g. adding 2 favorite dishes to a user that already has 1 so the resulting dish list is expected to have 3).

I frequently found that I would create/add all the dishes running the request “Adds All Dishes Request” in the SETUP_THE_DISHES folder and then leave it alone until my test cases all worked.

[1] I found I really did not need this dictionary like object

After logging all the key users in (USER_1, USER_2 and the ADMIN). I usually just run everything in FAVORITES_DISHID (one by one) and FAVORITE_DISHES (one by one) until you reach the one that is failing.

Note: When you “Run Collection” you can Deselect All the test cases and just Select those you want to run.

Environment Variables

User related environment variables

VARIABLE	TYPE	INITIAL VALUE	CURRENT VALUE
username_unregistered	default		
password_unregistered	default		password
username_registered_not_logged	default		coursera_registered_notlogged_student.16434...
password_registered_not_logged	default		password
username_registered_logged_1	default		coursera_registered_logged_student1.164348...
password_registered_logged_1	default		password
username_registered_logged_2	default		coursera_registered_logged_student2.164348...
password_registered_logged_2	default		password
registered_logged_1_token	default		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfa...
registered_logged_2_token	default		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfa...
unregistered_username	default		unregistered_username
unregistered_password	default		unregistered_password
admin_username	default		admin
admin_password	default		password
admin_token	default		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfa...

Persist All | Reset All

Other Test Variables

☑	dishNameList	default		[object Object]
☑	dishIdList	default		[object Object]
☑	dishId01	default		61f6b2e2d3e8d54e04b01670
☑	dishId02	default		61f6b2e2d3e8d54e04b01671
☑	garbageDishId	default		12345678900987654321
☑	dishId03	default		61f6b2e2d3e8d54e04b01672
☑	dishId04	default		61f6b2e2d3e8d54e04b01673
	Add a new variable			

Note: There are other variables (e.g. promoId) but they can be safely removed. I use the same environment file for all confusionServer related collections. It would be best to create a specific environment for a specific collection.

REGISTER_USERS

In order to successfully create a user that does not exist in our database, the code generates a date timestamp and appends it to each name. This means that usernames will appear in the database as something like: coursera_registered_notlogged_student.1643488090094. This technique for constructing usernames is used for our UNLOGGED, USER_1 and USER_2 clients that we use in the test cases.

Note: The admin is not registered but assumed to already exist in the database with the username: admin and the admin flag set to true. You can change the assumed admin username in the collection variable section.

REQUEST: POST_REGISTER_UNLOGGED_USER

REQUEST: POST {{baseUrl}}/users/signup

BODY:

```
{
"username" : "{{username_registered_not_logged}}",
"password": "{{password_registered_not_logged}}",
"firstname" : "Alan",
"lastname" : "Turing"
```

Pre-request SCRIPT:

```
// Create a new account for a user that will be registered BUT not logged in
var date = Date.now();
```

```
username_registered_not_logged = "coursera_registered_notlogged_student." + date;
password_registered_not_logged = "password";
// Add to environment for other test cases
pm.environment.set("username_registered_not_logged", username_registered_not_logged);
pm.environment.set("password_registered_not_logged", password_registered_not_logged);

// Create a username and password for an unregistered user
username_unregistered = "unregistered_username"
password_unregistered = "unregistered_password";
// Add to environment for other test cases
pm.environment.set("unregistered_username", username_unregistered);
pm.environment.set("unregistered_password", password_unregistered);
```

Note: Thisrequest could probably be removed with NO IMPACT to this collection test suite.

TESTS:

```
pm.test('POST_REGISTER_UNLOGGED_USER', function () {
    const data = pm.response.json();
    console.log("data: " + data);
    pm.response.to.have.status(200);
    pm.expect(data.success).to.be.true;
    pm.expect(data.status).to.eql("Registration Successful!");
});
```

REQUEST: POST_REGISTER_LOGGED_USER_1

REQUEST: POST {{baseUrl}}/users/signup

BODY:

```
{
"username" : "{{username_registered_logged_1}}",
"password": "{{password_registered_logged_1}}",
"firstname" : "James",
"lastname" : "Gosling"
}
```

The Pre-request Script and Tests is similar to the previous request except that the new username is generated for USER_1.

REQUEST: POST_REGISTER_LOGGED_USER_2

REQUEST: POST {{baseUrl}}/users/signup

BODY:

```
{
"username" : "{{username_registered_logged_2}}",
"password": "{{password_registered_logged_2}}",
"firstname" : "Barbara",
"lastname" : "Liskov"
}
```

The Pre-request Script and Tests is similar to the previous request except that the new username is generated for USER_2.

The key point for this folder is to establish three[2] new users

LOGIN_ALL_USERS

This folder contains requests to login USER_1, USER_2 and the ADMIN user. The token is saved in the respective variable for future requests.

REQUEST: POST_LOGIN_REGISTERED_USER_1

REQUEST: POST {{baseUrl}}/users/login

BODY:

```
{
"username" : "{{username_registered_logged_1}}",
"password": "{{password_registered_logged_1}}"
}
```

Pre-request SCRIPT:

```
console.log("===> LOGIN REGISTERED USER 1: ", pm.environment.get("username_registered_logged_1"));
```

This can be safely cleaned up.

[2] As noted in the Notes we can probably safely remove the unlogged in user since the environment variables created are unused.

TESTS:

```
pm.test('POST_LOGIN_REGISTERED_USER_1', function() {
    const data = pm.response.json(); // Obtain response
    pm.response.to.have.status(200);
    pm.expect(data.success).to.be.true;
    // Save Token
    let registered_logged_1_token = data.token;

    // Save the JWT Token associated with REGISTERED LOGGED USER #1
    pm.environment.set('registered_logged_1_token', registered_logged_1_token);
});
```

The token returned is saved.

REQUEST: POST_LOGIN_REGISTERED_USER_2

REQUEST: POST {{baseUrl}}/users/login

BODY:

Similar to the above except with the variables pertaining to USER_2

The Pre-request Script and Tests is similar to the above request except the USER_2 token variable is updated.

REQUEST: POST_LOGIN_ADMIN

REQUEST: POST {{baseUrl}}/users/login

BODY:

Similar to the above except with the variables pertaining to USER_2

The Pre-request Script and Tests is similar to the above request except the admin token variable is updated.

REQUEST: POST_LOGIN_UNREGISTERED_USER

REQUEST: POST {{baseUrl}}/users/login

BODY:

```
{
"username" : "bozo_the_clown",
"password": "bozo_the_clown"
}
```

I imagine no one has a registered user with the name 'bozo_the_clown' I probably should have created a collection variable for this so users can make up their own name.

Note: This is the first test case I expect to fail. I usually used specific status values to check for in the test case:

```
pm.test('POST_LOGIN_UNREGISTERED_USER_FAIL', function() {
    pm.response.to.have.status(401);
});
```

But, users of this collection may not use the same HTTP error code I use in my application. I decided to change these to be more generic for certain requests but for this one – we will stay with checking for 401.

But, if your code sends back something different change to something like:

```
pm.expect(pm.response.code).to.be.oneOf([401,403]);
```

another option is to use:

```
pm.response.to.have.statusCodeClass(4);
```

So the test case knows it should fail with a 4xx but does not want to assume which one. I changed several test cases to use this check since I did not want to assume other developers used the same HTTP STATUS CODE as I did but I knew it would probably be in the same class.

HOME

These requests are from the previous collection to test access to the home page by all users, unlogged/unregistered, admin and USER_1. This is left in as a way to regression test previous requests that referenced an endpoint accessible to all users.

Note: The previous test case collection had over 100 test cases to test successful and unsuccessful requests using all the known endpoints at the time. I did not want to unnecessarily regression test all those endpoints in this collection – I could always run that set idependently.

REQUEST: GET_HOME_PAGE

REQUEST: POST {{baseUrl}}/

BODY: None

Pre-request Script: None

Tests:

```
pm.test('GET_ALL_HOME_PAGE', function() {
    pm.response.to.have.status(200);
    pm.expect(pm.response.text()).to.include("Express");
});
```

REQUEST: GET_HOME_PAGE_ADMIN

REQUEST: POST {{baseUrl}}/

BODY: None

Pre-request Script:

```
var authTokenValue = "Bearer " + pm.environment.get("admin_token");
//pm.request.upsertHeader({"Authorization": authTokenValue});
pm.request.headers.add({
    'key': "Authorization",
    'value': authTokenValue
});
```

It is not required for this request but we send the admin_token in the request to make sure the admin sees the same thing.

Tests:

Same as previous request

*REQUEST: GET_HOME_PAGE_*

REQUEST: POST {{baseUrl}}/

BODY: None

Pre-request Script:

```
var authTokenValue = "Bearer " + pm.environment.get("registered_logged_1_token");
//pm.request.upsertHeader({"Authorization": authTokenValue});
pm.request.headers.add({
    'key': "Authorization",
    'value': authTokenValue
});
```

Sets up the authorization for USER_1

Tests:

Same as previous request

The last two requests demonstrates how the test requests will establish the user as admin, USER_1 or USER_2 throughout the rest of the test requests by specifying the associated token variable.

SETUP_THE_DISHES

This folder only has one request. The request actually does the bulk of the work in the Pre-request Script, where we add the 4 dishes. This requires that we set the authorization token to the admin. The request itself does a GET /dishes to check that all four dishes have been added to the database.

REQUEST: Adds All Dishes Request

REQUEST: GET {{baseUrl}}/dishes

BODY: None

Pre-request Script:

```
// Set up the four key dishes
DishObject =
{
  "dishes": [
    {
      "name": "Uthappizza",
      "image": "images/uthappizza.png",
      "category": "mains",
      "label": "Hot",
      "price": "4.99",
      "featured": "true",
      "description": "A unique combination of Indian Uthappam (pancake) and Italian pizza, topped with
Cerignola olives, ripe vine cherry tomatoes, Vidalia onion, Guntur chillies and Buffalo Paneer.",
      "comments": [ ]
    },
    {
      "name": "Zucchipakoda",
      "image": "images/zucchipakoda.png",
      "category": "appetizer",
      "label": "",
      "price": "1.99",
      "featured": "false",
      "description": "Deep fried Zucchini coated with mildly spiced Chickpea flour batter accompanied w
ith a sweet-tangy tamarind sauce",
```

```
      "comments": [ ]
    },
    {
      "name": "Vadonut",
      "image": "images/vadonut.png",
      "category": "appetizer",
      "label": "New",
      "price": "1.99",
      "featured": "false",
      "description": "A quintessential ConFusion experience, is it a vada or is it a donut?",
      "comments": [ ]
    },
    {
      "name": "ElaiCheese Cake",
      "image": "images/elaicheesecake.png",
      "category": "dessert",
      "label": "",
      "price": "2.99",
      "featured": "false",
      "description": "A delectable, semi-
sweet New York Style Cheese Cake, with Graham cracker crust and spiced with Indian cardamoms",
      "comments": [ ]
    }
  ]
};

// Holds key: name and value: _id for each dish created
let dishNameList = new Object();
pm.environment.set('dishNameList', dishNameList);
// Holds key: _id and value name for each dish
let dishIdList = new Object();
pm.environment.set('dishIdList', dishIdList);
// obtain the adminToken
const adminToken = 'Bearer ' + pm.environment.get('admin_token');
const base_url = pm.collectionVariables.get("baseUrl");
for (let i=0; i < DishObject.dishes.length; i++) {
    console.log("===> dish name: ", DishObject.dishes[i].name);
    let newDish = DishObject.dishes[i];
    // Send request to insert dish i into database
    pm.sendRequest({
        url: `${base_url}/dishes`,
```

```
        method:'POST',
        header: {'Content-Type' : 'application/json', 'Authorization': adminToken},
        body : {mode: 'raw', raw: newDish}},
        function(err, response) {
        if (err) {
            console.log("===> Callback: error");
            console.log(err);
        } else if (pm.expect(response.status).to.eql('OK')) {
            console.log("===> Callback success");
            var jsonData = response.json();
            // Now populate the global dicts
            let dishNameListDict = pm.environment.get('dishNameList');
            dishNameListDict[newDish.name] = jsonData._id;
            pm.environment.set('dishNameList', dishNameListDict);
            let dishIdListDict = pm.environment.get('dishIdList');
            dishIdListDict[jsonData._id] = newDish.name;
            pm.environment.set('dishIdList',dishIdListDict);
        } else {
            console.log("===> Callback: not OK");
            console.log(pm.response);
        }
    });
}
```

At this point in my testing I did not know if I would need the dish name to map to a dishId or the dishId to map to a dishname – so I created both dictionaries. I found I only really needed dishNameList where it contained the following

```
[     "Uthappizza": <dishId Returned>,
      "Zucchipakoda":<dishId Returned>,
      "Vadonut":<dishIdReturned>,
      "ElaiCheese Cake": <dishId Returned>]
```

The only assumption I made in the subsequent test cases is that a) there were at least 4 dishes in the dishNameList b) I could access a dish by obtaining the keys and using key list to obtain the ith dishId. Here is an example,

```
let keys = Object.keys(dishNameList);
let dishId01 = dishNameList[keys[0]]
```

Tests:

The test case just confirmed that 4 dishes are available in the database.

```
pm.test("CHECK_ALL_FOUR_DISHES_ADDED_1", function () {
    pm.response.to.have.status(200);
    const jsonData = pm.response.json();
    // Check

    const dishNameList = pm.environment.get('dishNameList');
    let keys = Object.keys(dishNameList);
    console.log("===> dishNameList keys: \n", keys);

    const dishIdList = pm.environment.get('dishIdList');
    keys = Object.keys(dishIdList);
    console.log("===> dishIdList keys:\n", keys);
    // Make sure all the dishes were set up
    pm.expect(jsonData.length).eql(4);
});
```

FAVORITES

This folder contains two request, one for an logged in user (with no favorites) and unlogged in user.

REQUEST: GET_FAVORITES_REQUEST_USER_1

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Set up authorization for USER_1

Tests: Check for empty list of favorites

```
pm.test("GET_FAVORITES_REQUEST_USER_1_NONE_FOUND", function () {
    pm.response.to.have.status(200);
    var jsonData = pm.response.json();
    pm.expect(jsonData.length).to.eql(0);
});
```

REQUEST: GET_FAVORITES_REQUEST_UNLOGGED_USER

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: None

Tests: The user(?) is not authorized to see

```
pm.test("GET_FAVORITES_REQUEST_UNLOGGED_USER", function () {
    pm.response.to.have.status(401);
});
```

A good question is “shouldn’t a registered user with favorites be allowed to see their own list even if they are not logged in?” I determined that the user needed to be logged in so that req.user property would be set, that is, it did not make sense to allow an unlogged in user to try to see anything since the application would not have enough information (which user?) to obtained a document from the Favorites collection.

FAVORITES_DISHID

I found it easier as a developer to implement the favorites/:dishId operations. I felt the code would be less complex than trying to process a list of dishes in the body. So the test cases for this endpoint was implemented and tested first.

REQUEST: GET_DISHID01_USER_1_FAIL

REQUEST: GET {{baseUrl}}/favorites/:dishId01

BODY: None

Pre-request Script: Setting up authorization for USER_1 and dishId01

TESTS: Check for class 4xx error.

The application does not support this operation, the user will see the message:

GET operation not supported on /favorites/61f96dbc2a3a1a6590e5a5df

REQUEST: POST_DISHID01_USER_1

REQUEST: POST {{baseUrl}}/favorites/{{dishId01}}

BODY: None

Pre-request Script:

```
// Setting up get request for registered user #1
var authTokenValue = "Bearer " + pm.environment.get("registered_logged_1_token");
pm.request.headers.add({
    'key': "Authorization",
    'value': authTokenValue
});
```

```
const dishNameList = pm.environment.get('dishNameList');
let keys = Object.keys(dishNameList);
let dishId01 = dishNameList[keys[0]]
pm.environment.set("dishId01", dishId01);
```

TESTS:

- Success
- Addes dishId01 as a favorite for USER_1

```
pm.test("POST_DISHID01_USER_1", function () {
    pm.response.to.have.status(200);
    let jsonData = pm.response.json();
    let dishList = jsonData.dishes;
    pm.expect(dishList).contains(pm.environment.get("dishId01"));
});
```

Example Response:

```
{
    "dishes": [
        "61f96dbc2a3a1a6590e5a5df"
    ],
    "_id": "61f9c4ce2a3a1a6590e5a5e8",
    "user": "61f96dba2a3a1a6590e5a5dd",
    "__v": 0
}
```

REQUEST: POST_GARBAGE_DISHID_USER_2

This request sends an invalid dishId to add as a favorite. This is where mongoose validation will fail since by definition the dishId needs to be a valid document in the Dishes collection.

REQUEST: POST {{baseUrl}}/favorites/{{garbageDishId}}

BODY: None

Pre-request Script:

```
// Setting up get request for registered user #1
var authTokenValue = "Bearer " + pm.environment.get("registered_logged_2_token");
pm.request.headers.add({
```

```
    'key': "Authorization",
    'value': authTokenValue
});

let garbageDishId = "12345678900987654321"
pm.environment.set("garbageDishId", garbageDishId);
```

TESTS:

Returns 500 error.

REQUEST: GET_FAVORITES_REQUEST_USER_1_ONE_FOUND

This request is used to confirm that USER_1 only has one favorite dish

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_1

TESTS:

```
pm.test("GET_FAVORITES_REQUEST_USER_1_ONE_FOUND", function () {
    pm.response.to.have.status(200);
    var jsonData = pm.response.json();
    var dishList = jsonData[0].dishes;
    pm.expect(dishList.length).to.eql(1);
});
```

I expect to find only one (dishId1) for USER_1

The response:


```
[
    {
        "dishes": [
            {
                "label": "New",
                "featured": false,
                "_id": "61f9c6eb2a3a1a6590e5a5ea",
                "name": "Vadonut",
                "image": "images/vadonut.png",
```

```
                "category": "appetizer",
                "price": 199,
                "description": "A quintessential ConFusion experience, is it a vada or is it a donut?",
                "comments": [],
                "createdAt": "2022-02-01T23:48:59.500Z",
                "updatedAt": "2022-02-01T23:48:59.500Z",
                "__v": 0
            }
        ],
        "_id": "61f9c77c2a3a1a6590e5a5ee",
        "user": {
            "firstname": "James",
            "lastname": "Gosling",
            "admin": false,
            "_id": "61f96dba2a3a1a6590e5a5dd",
            "username": "coursera_registered_logged_student1.1643736506408",
            "__v": 0
        },
        "__v": 0
    }
]
```


REQUEST: POST_DISHID01_USER_1_FAIL

This request tries to post the same dish as a favorite

REQUEST: POST {{baseUrl}}/favorites/{{dishId01}}

BODY: None

Pre-request Script: Same as before

TESTS:

```
pm.test("POST_DISHID01_USER_1_FAIL", function () {
    pm.response.to.have.statusCodeClass(4);
});
```

Response:

REQUEST: POST_DISHID02_USER_1

This request adds a second favorite dish for USER_1

REQUEST: POST {{baseUrl}}/favorites/{{dishId02}}

BODY: None

Pre-request Script: Similar to the previous request but now dishId02 is setup

TESTS:

Expect Success with two dishes in USER_1 favorites

```
{
    "dishes": [
        "61f9c6eb2a3a1a6590e5a5ea",
        "61f9c6eb2a3a1a6590e5a5e9"
    ],
    "_id": "61f9c77c2a3a1a6590e5a5ee",
    "user": "61f96dba2a3a1a6590e5a5dd",
    "__v": 1
}
```

REQUEST: DELETE_DISHID01_USER_1

This request deletes dish01 as a use 1 favorite

REQUEST: DELETE {{baseUrl}}/favorites/{{dishId01}}

BODY: None

Pre-request Script:

TESTS:

Expect success

```
pm.test("DELETE_DISHID01_USER_1", function () {
    // Adding  a second favorite for user 1
```

```
    pm.response.to.have.status(200);
    let jsonData = pm.response.json();
    let dishList = jsonData.dishes;
    pm.expect(dishList).contains(pm.environment.get("dishId02"));
    pm.expect(dishList.length).eql(1);
});
```

Expect to se only one dish in the favorite list

Example Response:

```
{
    "dishes": [
        "61f9c6eb2a3a1a6590e5a5e9"
    ],
    "_id": "61f9c77c2a3a1a6590e5a5ee",
    "user": "61f96dba2a3a1a6590e5a5dd",
    "__v": 2
}
```

REQUEST: DELETE_DISHID02_USER_2_FAIL

This request attempts to delete a dish that is NOT in USER_2 favorites

REQUEST: DELETE {{baseUrl}}/favorites/{{dishId01}}

BODY: None

Pre-request Script: Sets up authorization for USER_2

TESTS:

```
pm.test("DELETE_DISHID02_USER_2_FAIL", function () {
    // User #2 does not have any favorite dishes so deleting one not found
    // associated with the user
    pm.response.to.have.statusCodeClass(4);
});
```

Example Response:

REQUEST: DELETE_DISHIDX_USER_1_FAIL

This request sends a delete request to USER_1 favorites where the dishId does not exist.

REQUEST: DELETE {{baseUrl}}/favorites/{{garbageDishId}}

BODY: None

Pre-request Script: Sets up authorization for USER_1

TESTS: Expect a 4xx failure

FAVORITE_DISHES

These requests test the /favorites endpoint

REQUEST: GET_FAVORITES_REQUEST_USER_1_AGAIN

This request just checks the number of favorite dishes for USER_1. We expect 1.

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_1

TESTS: Success is expected with user and all dishes populated

Example response:


```
[
    {
        "dishes": [
            {
                "label": "",
                "featured": false,
                "_id": "61f9c6eb2a3a1a6590e5a5e9",
                "name": "Zucchipakoda",
                "image": "images/zucchipakoda.png",
                "category": "appetizer",
```

```
                "price": 199,
                "description": "Deep fried Zucchini coated with mildly spiced C
hickpea flour batter accompanied with a sweet-tangy tamarind sauce",
                "comments": [],
                "createdAt": "2022-02-01T23:48:59.500Z",
                "updatedAt": "2022-02-01T23:48:59.500Z",
                "__v": 0
            }
        ],
        "_id": "61f9c77c2a3a1a6590e5a5ee",
        "user": {
            "firstname": "James",
            "lastname": "Gosling",
            "admin": false,
            "_id": "61f96dba2a3a1a6590e5a5dd",
            "username": "coursera_registered_logged_student1.1643736506408",
            "__v": 0
        },
        "__v": 2
    }
]
```


REQUEST: GET_FAVORITES_REQUEST_USER_2_AGAIN

This request checks the number of favovite dishes associated with USER_2. We expect to see []

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_2

TESTS: Success

Example Response:

[ ]

REQUEST: POST_TWO_NEW_DISHES_USER_1

This request will send a POST for two new favorite dishes for USER_1 dish03 and dish04

REQUEST: POST {{baseUrl}}/favorites

BODY:

```
[
   {"_id": "{{dishId03}}"},
   {"_id": "{{dishId04}}"}
]
```

Pre-request Script: Sets up authorization for USER_1 and the dishes in the body.

TESTS:

Expect Success and for USER_1 to have a total of 3 dishes

```
pm.test("POST_TWO_NEW_DISHES_USER_1", function () {
    // User should have a total of 3 now
    pm.response.to.have.status(200);
    let jsonData = pm.response.json();
    let dishList = jsonData.dishes;
    pm.expect(dishList.length).eql(3);
});
```

Example response:

```
{
    "dishes": [
        "61f9c6eb2a3a1a6590e5a5e9",
        "61f9c6eb2a3a1a6590e5a5eb",
        "61f9c6eb2a3a1a6590e5a5ec"
    ],
    "_id": "61f9c77c2a3a1a6590e5a5ee",
    "user": "61f96dba2a3a1a6590e5a5dd",
    "__v": 3
}
```

REQUEST: POST_TWO_NEW_DISHES_USER_1_ADD_SAME

This request posts the same two dishes for USER_1. It does not error out, but it does not duplicate them to the favorites dish.

REQUEST: POST {{baseUrl}}/favorites

BODY:

Same as previous request

Pre-request Script: Same as previous request

TESTS:

Success and checks that USER_1 still only has 3 dishes. Same as previous request

REQUEST: POST_TWO_NEW_DISHES_USER_2

This request posts the same two new dishes for USER_2.

REQUEST: POST {{baseUrl}}/favorites

BODY:

Same as previous request.

Pre-request Script: Sets up authorization for USER_2.

TESTS:

Success and checks that USER_2 has two favorites in its list.

Example response:

```
{
    "dishes": [
        "61f9c6eb2a3a1a6590e5a5eb",
        "61f9c6eb2a3a1a6590e5a5ec"
    ],
    "_id": "61f9cbb42a3a1a6590e5a5f0",
    "user": "61f96dba2a3a1a6590e5a5de",
    "__v": 0
}
```

REQUEST: DELETE_ALL_USER_1_FAVORITES

This request deletes ALL favorites associated with USER_1

REQUEST: DELETE {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_1

TESTS:

Success expected and checks for record to be deleted.

```
pm.test('DELETE_ALL_USER_1_FAVORITES', function() {
    pm.response.to.have.status(200);
```

```
    const responseJson = pm.response.json();
    pm.expect(responseJson.n).to.eql(1);
    pm.expect(responseJson.ok).to.eql(1);

 })
```

Example response:

```
{
    "n": 0,
    "ok": 1
}
```

REQUEST: DELETE_ALL_USER_2_FAVORITES

This request removes all the favorites from USER_2

REQUEST: DELETE {{baseUrl}}/favorites

BODY: None

Pre-request Script: Set up authorization for USER_2

TESTS:

Same as previous request.

REQUEST: DELETE_ALL_USER_2_NO_FAVORITES

Sends the same request knowing USER_2 should have no favorites

REQUEST: DELETE {{baseUrl}}/favorites

BODY: None

Pre-request Script: Set up authorization for USER_2

TESTS:

```
{
    "n": 0,
    "ok": 1
}
```

REQUEST: GET_FAVORITE_REQUEST_USER_1_AGAIN_2

This request checks to see that USER_1 has no favorites

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_2

TESTS:

Success expected

Example response:

[ ]

REQUEST: GET_FAVORITE_REQUEST_USER_2_AGAIN_2

This request checks to see that USER_2 has no favorites

REQUEST: GET {{baseUrl}}/favorites

BODY: None

Pre-request Script: Sets up authorization for USER_2

TESTS:

Same result as the previous request.

CLEANUP

These request remove all the dishes created

REQUEST: DELETE_ALL_DISHES_ADMIN

REQUEST: DELETE {{baseUrl}}/dishes

BODY: None

Pre-request Script: Set up ADMIN authorization

TESTS:

Success with all 4 dish records removed.

Example response:

```
{
    "n": 4,
    "ok": 1
```

```
}
```

REQUEST: GET_ALL_DISHES_CHECK_ALL_GONE

This request checks that no dishes remain in the database (otherwise it can create an issue of this test suite is run again).

REQUEST: GET {{baseUrl}}/dishes

BODY: None

Pre-request Script: None

TESTS:

Success expected.

Example response:

[ ]

This cleans up the collections dishes and favorites