

# LAB 3

## ❖ CONTENT

- Authorization (role-based access)
- Image upload with "multer" package
- Create and test Restful APIs for CRUD features

## ❖ INTRODUCTION

- API stands for Application Programming Interface. It serves as a bridge between different application, allowing them to communicate and interact with each other
- Common data format of API: JSON, XML
- Common types of API: RESTful, SOAP, GraphQL
- Common methods of API: GET (READ), POST (CREATE), PUT (UPDATE), DELETE (DELETE)
- The most popular usage of API nowadays: RESTful with JSON

## ❖ INSTRUCTION

### 1. Authorization

```
router.post('/register', async (req, res) => {  
  try {  
    var userRegistration = req.body;  
    var hashPassword = bcrypt.hashSync(userRegistration.password, salt);  
    var user = {  
      username: userRegistration.username,  
      password: hashPassword,  
      role: 'user'  
    }  
  }  
}
```

Figure 1 - Set custom role for new user (routes/auth.js)

```

    req.session.role = user.role;
    if (user.role == 'admin') {
        res.redirect('/admin');
    }
    else {
        res.redirect('/user');
    }
}

```

Figure 2 – Save user role to session and redirect page by role after login success (*routes/auth.js*)

```

//check login only
const checkLoginSession = (req, res, next) => {
    if (req.session.username) {
        next();
    } else {
        res.redirect('/auth/login');
    }
}

//check single role
const checkSingleSession = (req, res, next) => {
    if (req.session.username && req.session.role == 'admin') {
        next();
    }
    else {
        res.redirect('/auth/login');
        return;
    }
}

//check multiple roles
const checkMultipleSession = (allowedRoles) => (req, res, next) => {
    if (req.session.username && allowedRoles.includes(req.session.role)) {
        next();
    } else {
        res.redirect('/auth/login');
    }
}

module.exports = {
    checkLoginSession,
    checkSingleSession,
    checkMultipleSession
}

```

Figure 3 - update *auth* middleware for authorization (*middlewares/auth.js*)

```
const { checkSingleSession, checkMultipleSession } = require('../middlewares/auth');

router.get('/', checkMultipleSession(['user', 'admin'])) async (req, res) => {
  var productList = await ProductModel.find({}).populate('category');
  res.render('product/index', { productList });
};

router.get('/add', checkSingleSession, async (req, res) => {
  var categoryList = await CategoryModel.find({});
  res.render('product/add', { categoryList });
});
```

Figure 4 – import and add middleware to route functions (*routes* folder)

```
//IMPORTANT: place this code before setting router url
const { checkSingleSession } = require('../middlewares/auth');
app.use('/category', checkSingleSession);
```

Figure 5 – set user authorization for whole router (*app.js*)

## 2. Image upload

```
npm install multer
```

Figure 6 - Install new package

```
<form action="" method="post" enctype="multipart/form-data">
  Image: <input type="file" name="image" id="" required>
```

Figure 7 - update form add (*views/product/add.hbs*)

```
//import and config "multer" package
var multer = require('multer');

//generate an unique value for image name prefix
var prefix = Date.now();

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, './public/images/'); //set image upload location
  },
  filename: (req, file, cb) => {
    let fileName = prefix + file.originalname; //set final image name
    cb(null, fileName);
  }
});

const upload = multer({ storage: storage })
```

Figure 8 - import *multer* package, set image name & upload location (*routes/product.js*)

```
router.post('/add', upload.single('image'), async (req, res) => {
  try {
    var product = req.body;
    product.image = prefix + "_" + req.file.originalname;
  } catch (err) {
    res.status(400).send('Load product failed !' + err);
  }
})
```

Figure 9 - update route function to upload image and save it to database (*routes/product.js*)

```
{{#each productList }}
<tr>
  <td>{{ name }}</td>
  <td>${{ price }}</td>
  <td>
    
  </td>
</tr>
```

Figure 10 - update image location on view to display (*views/product/index.hbs*)

#### NOTES:

- The image itself will be uploaded to project folder, only image name will be saved to database
- Must set the new image name (such as original name with a unique value) before saving to prevent the possibility of duplicate image names and the new one will override the existing one
- Should create a function for image upload in middleware to reuse codes in different places

### 3. Create Restful APIs

```
var apiRouter = require('./routes/api');

app.use('/api', apiRouter);
```

Figure 11 - declare *apiRouter* to store APIs (*routes/api.js*)

```
router.get('/product', async (req, res) => {
  try {
    var products = await ProductModel.find({}).populate('category');
    res.status(200).json(products);
  } catch (err) {
    res.status(400).send('Load product list failed !' + err);
  }
})
```

Figure 12 - GET method (READ feature)

```
router.post('/product/add', async (req, res) => {
  try {
    await ProductModel.create(req.body);
    res.status(201).send('Create product succeed !');
  } catch (err) {
    res.status(400).send('Create product failed !' + err);
  }
})
```

Figure 13 - POST method (CREATE feature)

```
router.put('/product/edit/:id', async (req, res) => {
  try {
    await ProductModel.findByIdAndUpdate(req.params.id, req.body);
    res.status(200).send('Edit product succeed !');
  } catch (err) {
    res.status(400).send('Edit product failed !' + err);
  }
})
```

Figure 14 - PUT method (UPDATE feature)

```
router.delete('/product/delete/:id', async (req, res) => {
  try {
    await ProductModel.findByIdAndDelete(req.params.id);
    res.status(200).send('Delete product succeed !');
  } catch (err) {
    res.status(400).send('Delete product failed !' + err);
  }
})
```

Figure 15 - DELETE method (DELETE feature)

## 4. Test Restful APIs

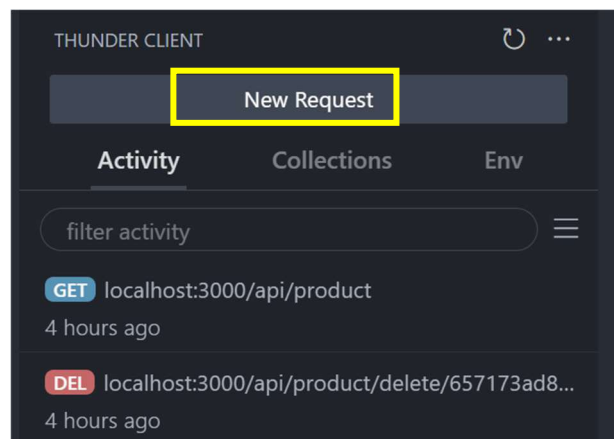


Figure 16 – Create new request to test APIs with Thunder Client in VS Code

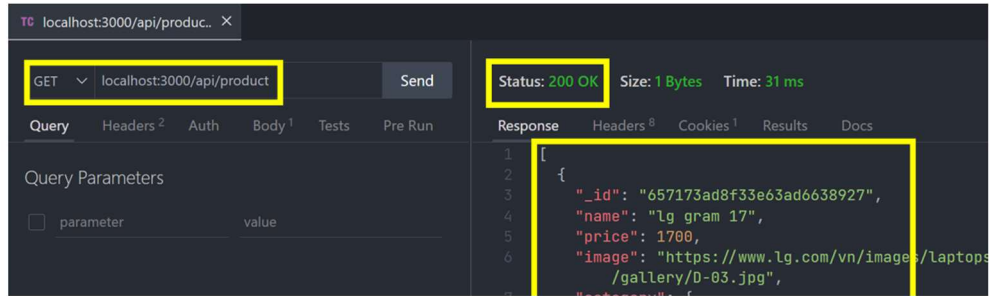


Figure 17 - Test GET method

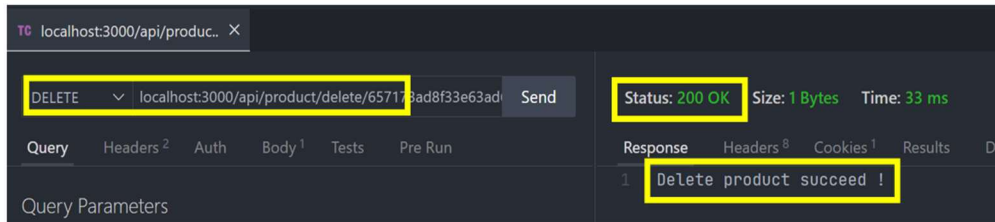


Figure 18 - Test DELETE method

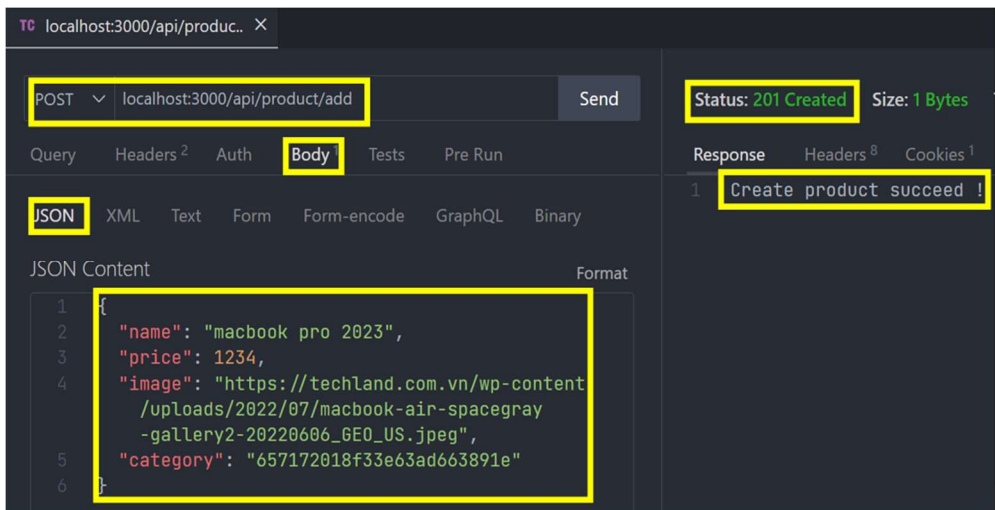


Figure 19 - Test POST method

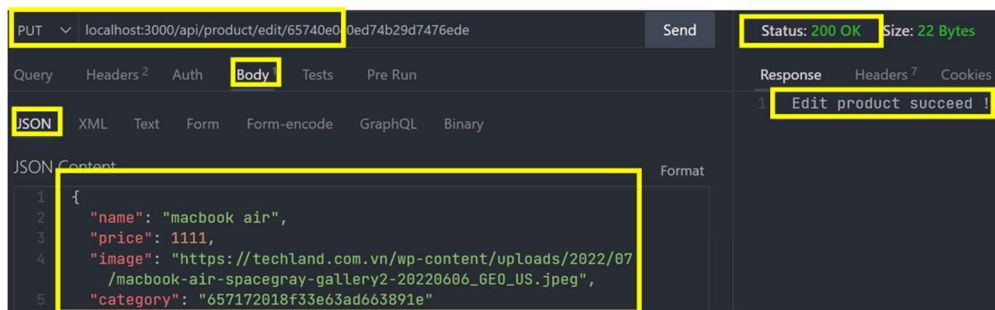


Figure 20 - Test PUT method