

Practical 4: Express JS, File IO and Socket IO

Learning Outcomes:

- Create server applications using Express framework and Node JS.
- Explain and apply file input/output operations in Node JS.
- Apply real time event handling in server applications using Socket IO package.







In this practical, we are going to create a web application that allows users to register an account, login with an account to upload images and videos, write comments, and chat with each other.

1. Setting up the project and main navigation page.

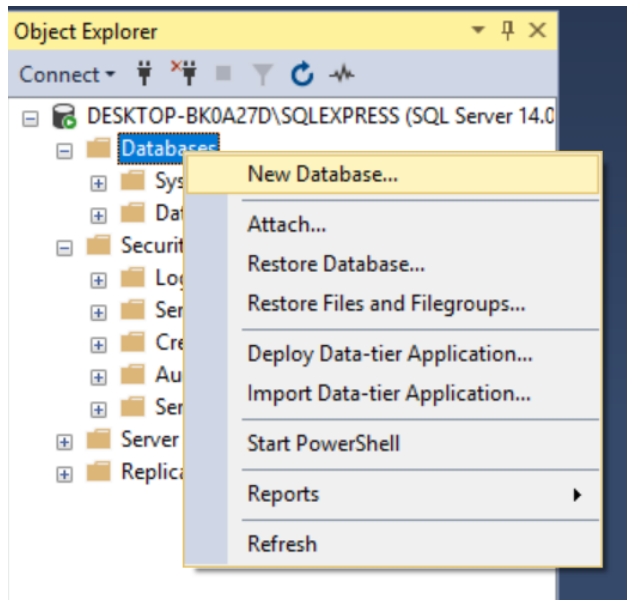
- Create a new folder “**practical4_express_file_socket**”, or any names that you prefer.
- Unzip the folder “public” from **extras.zip** into this new folder. For eg, if I created the folder “C:\practical4_express_file_socket”, then there should be “C:\practical4_express_file_socket\public”.
- Go to Visual Studio Code and open this folder.
- Open the Integrated Terminal in Visual Studio Code (Go to View menu, then click on Integrated Terminal).
- Using the existing package.json, restore all the node packages to this folder by typing “**npm install**”. NPM will download the packages into this folder.

2. Setting up database

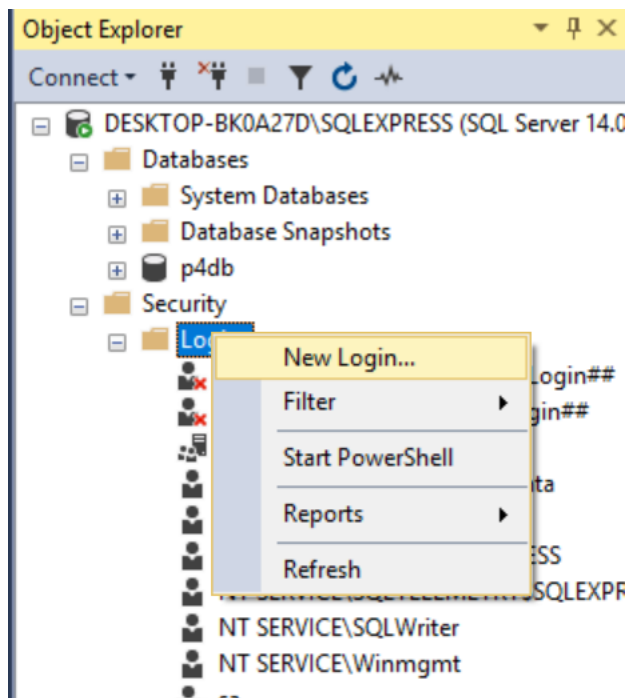
- Go to Windows Start menu, search for “Services”. Make sure “SQL Server (SQLEXPRESS)” and “SQL Server Browser” are both running.

	Spot Verifier	Verifies potential file s...	Manual (Trigg...	Local System
	SQL Server (SQLEXPRESS)	Provides storage, proc...	Running	Automatic NT Service\MSSQL\$SQLEXPRESS
	SQL Server Agent (SQLEXPRESS)	Executes jobs, monitor...	Disabled	Network Service
	SQL Server Browser	Provides SQL Server co...	Running	Automatic Local Service
	SQL Server CEIP service (SQLEXPRESS)	CEIP service for Sql ser...	Running	Automatic NT Service\SQLTELEMETRY\$SQLEXPRESS
	SQL Server VSS Writer	Provides the interface ...	Running	Automatic Local System

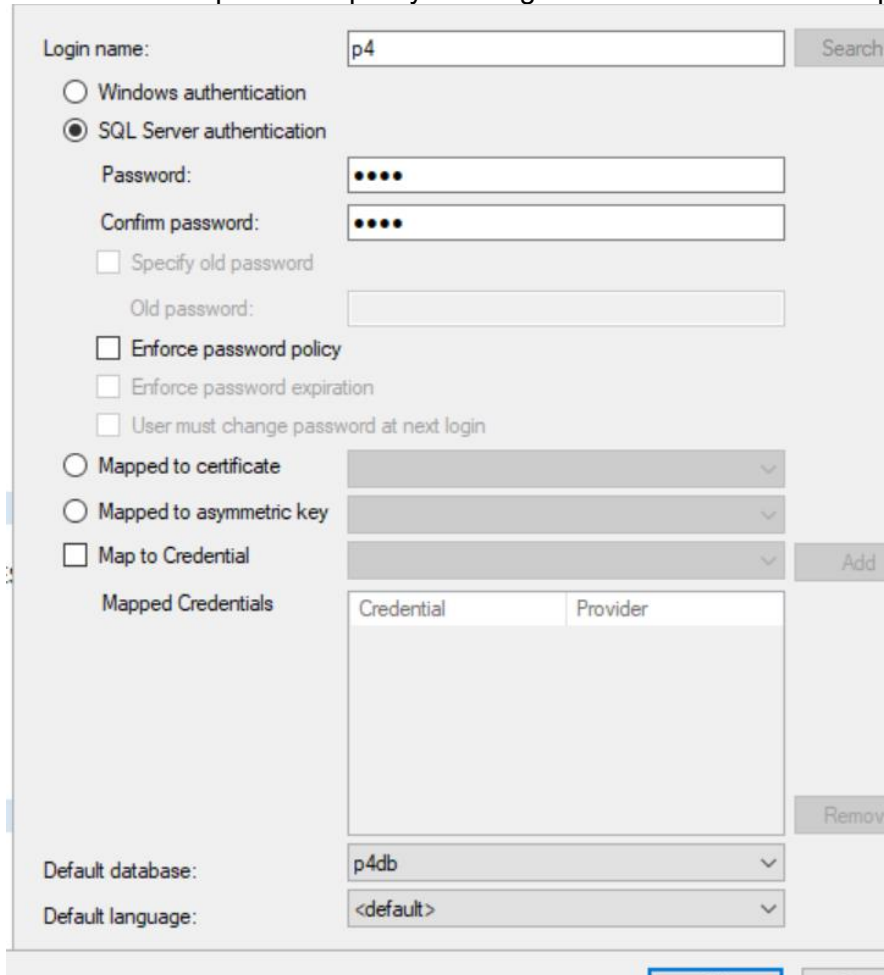
- Open Microsoft SQL Server Management Studio. You should be able to with your windows credentials.
- Create a new database by right clicking on “Databases” in the Object Explorer and click “New Database”. Name the new database “**p4db**”.



4. In the Object Explorer, expand "Security" and right click on "Login". Click on "New Login" to create a new SQL account on this local server.



5. Select "SQL Server authentication". Set the login name to "p4", and the password to "p4pw". Untick "Enforce password policy". Change the default database to "p4db".



Login name: Search...

☐ Windows authentication
☒ SQL Server authentication

Password:
 Confirm password:
☐ Specify old password
 Old password:

☐ Enforce password policy
☐ Enforce password expiration
☐ User must change password at next login

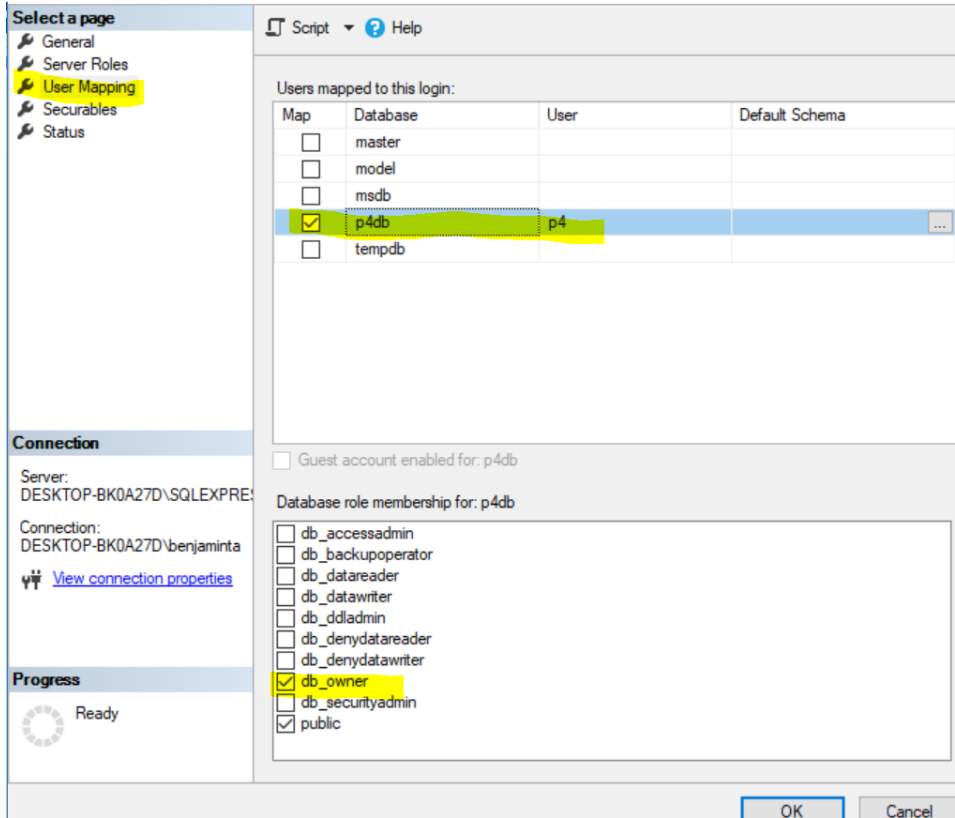
☐ Mapped to certificate ▼
☐ Mapped to asymmetric key ▼
☐ Map to Credential ▼ Add

Credential	Provider

Remove

Default database: ▼
 Default language: ▼

- Still on the Login creation screens, click on “User Mapping” on the left, tick on the database “p4db”, check the database role membership to “db_owner”. Click OK to create this new Login.



Select a page

- General
- Server Roles
- User Mapping**
- Securables
- Status

Connection

Server: DESKTOP-BK0A27D\SQLEXPRESS

Connection: DESKTOP-BK0A27D\benjaminta

[View connection properties](#)

Progress

Ready

Script ? Help

Users mapped to this login:

Map	Database	User	Default Schema
<input type="checkbox"/>	master		
<input type="checkbox"/>	model		
<input type="checkbox"/>	msdb		
<input checked="" type="checkbox"/>	p4db	p4	
<input type="checkbox"/>	tempdb		

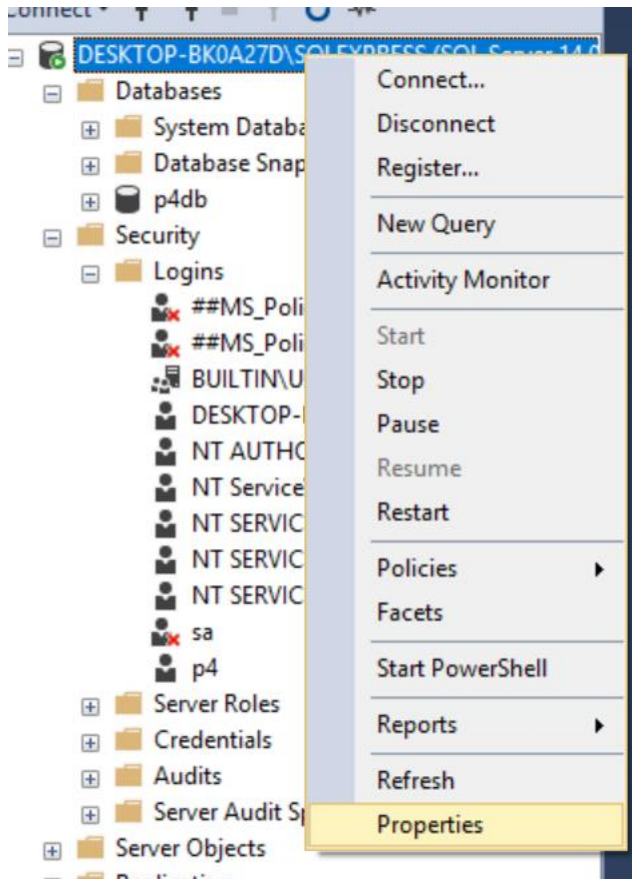
☐ Guest account enabled for: p4db

Database role membership for: p4db

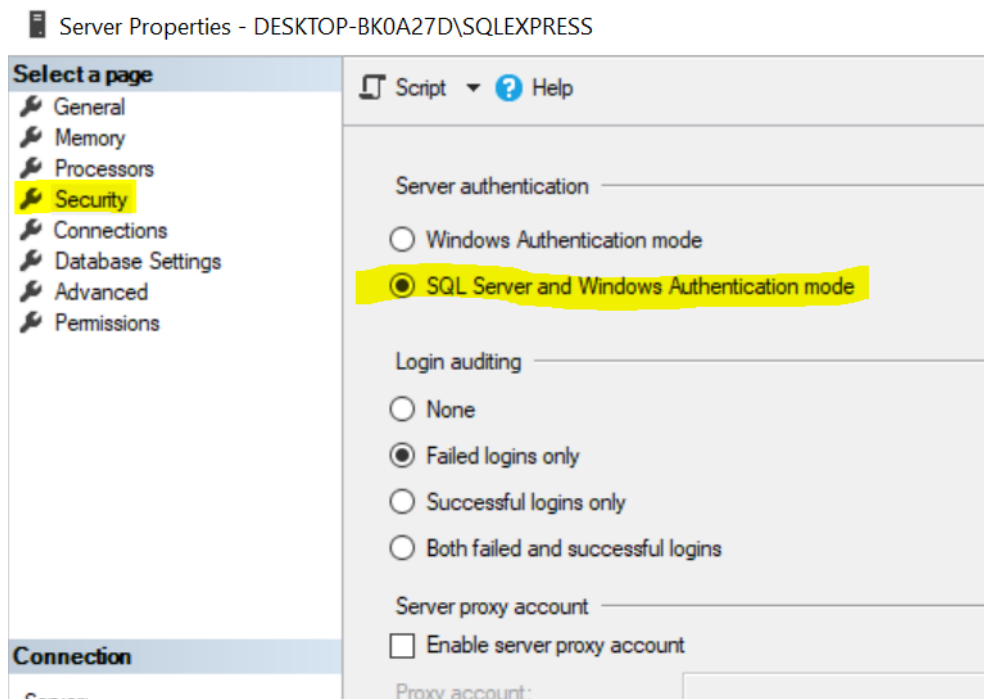
- ☐ db_accessadmin
- ☐ db_backupoperator
- ☐ db_datareader
- ☐ db_datawriter
- ☐ db_ddladmin
- ☐ db_denydatareader
- ☐ db_denydatawriter
- ☒ db_owner
- ☐ db_securityadmin
- ☒ public

OK Cancel

7. In the Object Explorer, right click on the very top database Instance “DESKTOP-XXXXX\SQLEXPRESS”. Click on Properties.



8. Click on “Security” on the left, then select “SQL Server and Windows Authentication mode”.



9. Now we are set to connect to this database from the node server.
10. Go back to Visual Studio Code. Open "**database.js**" file in `"/server/controllers"`. If you have chosen to create a database or login account that is not the same as described above, you can specify another database name, username, password in this JS file in order to connect to the database.
11. Go to the terminal and run "**node app**". If the database has been setup correctly, the server app should launch and start connecting to the database and creating new tables. You can use the SQL Server Management Studio to refresh the database "p4db", and look at the new tables created.
12. Open the browser, go to "**http://localhost:3000**". The webpage should load, and you should be able to sign in with the default profile email: **a@b.com**, password: **1234**.
13. Try to sign up for a new account with a new email and password. Log out of the current account in the "Profile" page, and sign in with the newly created account.

3. Posting Comments function

1. Now that the default website is working well with the SQL Server database, we can start to create new functions. First, we will create a Comment section where we can post new comments, view existing comments, and delete existing comments.

The database model `"/server/models/comments.js"` and the ejs template for the comments page rendering `"/server/views/pages/comments.ejs"` are already created for you. We will need to create the controller, and make modifications to `"app.js"` to route requests to the Comments page.

2. Under the folder `"/server/controllers"`, create a new file called **"comments.js"**. This will be our comments controller. Write the following codes to call the libraries/packages that we need:

```
// get gravatar icon from email
var gravatar = require('gravatar');
// get Comments model
var Comments = require('../models/comments');
var myDatabase = require('./database');
var sequelize = myDatabase.sequelize;
```

3. Next block is the function to read all the comments that current exists in the database.

```
// List Comments
exports.list = function (req, res) {
  // List all comments and sort by Date
  sequelize.query('select c.id, c.title, c.content, u.email AS [user_id] from Comments c join Users u on c.user_id = u.id', { model: Comments }).then((comments) => {

    res.render('comments', {
      title: 'Comments Page',
      comments: comments,
      gravatar: gravatar.url(comments.user_id, { s: '80', r: 'x', d: 'retro' }, true),
      urlPath: req.protocol + "://" + req.get("host") + req.url
    })
  }).catch((err)=>{
    return res.status(400).send({
      message: err
    });
  });
};
```

4. This block is the function to create a new comment in the database.

```
// Create Comments
exports.create = function (req, res) {
  console.log("creating comments")

  var commentData = {
    title: req.body.title,
    content: req.body.content,
    user_id: req.user.id
  }

  Comments.create(commentData).then((newComment, created) => {
    if (!newComment) {
      return res.send(400, {
        message: "error"
      });
    }

    res.redirect('/comments');
  })
};
```

5. This block is the function to delete a comment from the database based on the ID of the comment.

```
exports.delete = function (req, res) {
  var record_num = req.params.comments_id;
  console.log("deleting comments " + record_num);
  Comments.destroy({where: {id: record_num}}).then((deletedComment)=>{
    if(!deletedComment){
      return res.send(400, {
        message: "error"
      });
    }


    res.status(200).send({ message: "Deleted comments :" + record_num });
  })
}
```

6. This block takes care of the user login authorization. Only user who logged in can see, post, delete comments.

```
// Comments authorization middleware
exports.hasAuthorization = function (req, res, next) {
  if (req.isAuthenticated())
    return next();
  res.redirect('/login');
};
```

7. That's all for "comments.js" controller. Now, let's go "**app.js**" and setup the routes and link the controller for comments.
8. In "**app.js**", between the import of login controller and the import of database controller, we will insert the Comments controller.

```
// Import home controller
var index = require('./server/controllers/index');
// Import login controller
var auth = require('./server/controllers/auth');
// Modules to store session
var myDatabase = require('./server/controllers/database');
var expressSession = require('express-session');
```

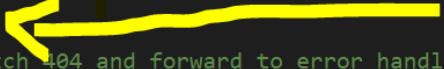


9. Type the following code to import the Comments controller.

```
// Import comments controller
var comments = require('./server/controllers/comments');
```

10. Between the Logout page route and catch 404 page function, we will insert new codes to go to Comments page.

```
// Logout Page
app.get('/logout', function (req, res) {
  req.logout();
  res.redirect('/');
});
// catch 404 and forward to error handler
app.use(function (req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```



11. Type the following code to route comments page requests to the controller and render the comments page.

```
// Setup routes for comments
app.get('/comments', comments.hasAuthorization, comments.list);
app.post('/comments', comments.hasAuthorization, comments.create);
app.delete('/comments/:comments_id', comments.hasAuthorization, comments.delete);
```

12. Save the changes and restart the node server: "node app". Visit your website in the browser once again. Click on the "Comments" heading after you have logged in. Now you should be able to create new comments, view the comments posted, and delete comments from the Comments page!

4. Upload Images and Videos function

1. In a similar way, let's add the Images and Videos functions. The Images page will allow you to upload images from your computer to the server folder, show all the images that were uploaded before. The Videos page will allow you to do the same for videos, and allows you to play the Videos that were uploaded in mp4 format. Both pages do not have the delete function.

The database models for the images (../models/images.js) and videos (../models/videos.js) are already created for you. The ejs templates are also created (../views/pages/image-gallery.ejs and ../views/pages/videos.ejs).

2. Under the folder **“/server/controllers”**, create a new file called **“images.js”**. This will be our images controller. Write the following codes to call the libraries/packages that we need. Here we also set the image file types that we accept only.

```
// Import modules
var fs = require('fs');
var mime = require('mime');
var gravatar = require('gravatar');
// set image file types
var IMAGE_TYPES = ['image/jpeg', 'image/jpg', 'image/png'];

var Images = require('../models/images');
var myDatabase = require('../database');
var sequelize = myDatabase.sequelize;
```

3. This block creates the function to get all the uploaded images from the database and show it on the page.

```
// Show images gallery
exports.show = function (req, res) {

  sequelize.query('select i.id, i.title, i.imageName, u.email AS [user_id] from Images i join Users u on i.user_id = u.id', { model: Images }).then((images) => {

    res.render('images-gallery', {
      title: 'Images Gallery',
      images: images,
      gravatar: gravatar.url(images.user_id, { s: '80', r: 'x', d: 'retro' }, true)
    });

  }).catch((err) => {
    return res.status(400).send({
      message: err
    });
  });
});
```

4. This block creates the function to handle the upload of images, saving the image to the server file directory (./public/images), and also save the information to database.

```

// Image upload
exports.uploadImage = function (req, res) {
  var src;
  var dest;
  var targetPath;
  var targetName;
  var tempPath = req.file.path;
  console.log(req.file);
  //get the mime type of the file
  var type = mime.lookup(req.file.mimetype);
  // get file extension
  var extension = req.file.path.split(/[. ]+/.pop();
  // check support file types
  if (IMAGE_TYPES.indexOf(type) == -1) {
    return res.status(415).send('Supported image formats: jpeg, jpg, jpe, png.');
```

```

  }
  // Set new path to images
  targetPath = './public/images/' + req.file.originalname;
  // using read stream API to read file
  src = fs.createReadStream(tempPath);
  // using a write stream API to write file
  dest = fs.createWriteStream(targetPath);
  src.pipe(dest);

  // Show error
  src.on('error', function (err) {
    if (err) {
      return res.status(500).send({
        message: error
      });
    }
  });
});

```

```

3 // Save file process
4 src.on('end', function () {
5   // create a new instance of the Images model with request body
6   var imageData = {
7     title: req.body.title,
8     imageName: req.file.originalname,
9     user_id: req.user.id
10  }
11  //Save to database
12  Images.create(imageData).then((newImage, created) => {
13    if (!newImage) {
14      return res.send(400, {
15        message: "error"
16      });
17    }
18    res.redirect('images-gallery');
19  })
20
21  // remove from temp folder
22  fs.unlink(tempPath, function (err) {
23    if (err) {
24      return res.status(500).send('Something bad happened here');
25    }
26    // Redirect to galley's page
27    res.redirect('images-gallery');
28  });
29 });
30 };

```

5. This block is the function to only allow logged in users to view and upload images.

```

1 // Images authorization middleware
2 exports.hasAuthorization = function (req, res, next) {
3   if (req.isAuthenticated())
4     return next();
5   res.redirect('/login');
6 };

```

6. The controller for images is now complete, let's create the controller for videos.
7. Under the folder **"/server/controllers"**, create a new file called **"videos.js"**. This will be our video controller. Write the following codes to call the libraries/packages that we need. Here we also set the video file types that we accept only.

```

// Import modules
var fs = require('fs');
var mime = require('mime');
// get gravatar icon from email
var gravatar = require('gravatar');
// set image file types
var VIDEO_TYPES = ['video/mp4', 'video/webm', 'video/ogg', 'video/ogv'];
// get video model
var Videos = require('../models/videos');
var myDatabase = require('../database');
var sequelize = myDatabase.sequelize;

```

8. This block is the function to get all the videos from database.

```

// List Videos
exports.show = function (req, res) {
  sequelize.query('select v.id, v.title, v.videoName, u.email AS [user_id] from Videos v join Users u on v.user_id = u.id', { model: Videos }).then((videos) => {

    res.render('videos', {
      title: 'Videos Page',
      videos: videos,
      gravatar: gravatar.url(videos.user_id, { s: '80', r: 'x', d: 'retro' }, true)
    });

  }).catch((err) => {
    return res.status(400).send({
      message: err
    });
  });
};

```

9. This block is the function to handle the upload of videos, saving the files to server directory (./public/videos), and saving the information to database.

```

// Create Videos
exports.uploadVideo = function (req, res) {
  var src;
  var dest;
  var targetPath;
  var targetName;
  console.log(req);
  var tempPath = req.file.path;
  //get the mime type of the file
  var type = mime.lookup(req.file.mimetype);
  // get file extension
  var extension = req.file.path.split(/[. ]+/.pop());
  // check support file types
  if (VIDEO_TYPES.indexOf(type) == -1) {
    return res.status(415).send('Supported video formats: mp4, webm, ogg, ogv');
  }
}

```

```

// Set new path to images
targetPath = './public/videos/' + req.file.originalname;
// using read stream API to read file
src = fs.createReadStream(tempPath);
// using a write stream API to write file
dest = fs.createWriteStream(targetPath);
src.pipe(dest);

// Show error
src.on('error', function (error) {
  if (error) {
    return res.status(500).send({
      message: error
    });
  }
});

// Save file process
src.on('end', function () {
  // create a new instance of the Video model with request body
  var videoData = {
    title: req.body.title,
    videoName: req.file.originalname,
    user_id: req.user.id
  }

  // Save to database
  Videos.create(videoData).then((newVideo, created) => {
    if (!newVideo) {
      return res.send(400, {
        message: "error"
      });
    }
    res.redirect('videos');
  })
  // remove from temp folder
  fs.unlink(tempPath, function (err) {
    if (err) {
      return res.status(500).send({
        message: error
      });
    }
    // Redirect to galley's page
    res.redirect('videos');
  });
});
};

```

10. This block is the function to allow only logged in users to upload and view the videos.

```
// Videos authorization middleware
exports.hasAuthorization = function (req, res, next) {
  if (req.isAuthenticated())
    return next();
  res.redirect('/login');
};
```

11. Now the controller for videos is complete. Let's go to "app.js" to link up the images and videos controller.
12. Next we need to install a package called "multer" to help us handle multipart file upload sent from the browsers as the image and video files tend to be large in file size. Go to Terminal and type "**npm install multer**".
13. Go to "app.js" in the section between bodyParser and importing of the home controller.

```
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

// Import home controller
var index = require('./server/controllers/index');
// Import login controller
var auth = require('./server/controllers/auth');
// Import comments controller
```

Insert the import of multer and set the destination folder to (./public/uploads/) with a file size limit of 1.5 MB.

```
// import multer
var multer = require('multer');
var upload = multer({ dest: './public/uploads/', limits: { fileSize: 1500000, files: 1 } });
```

14. In "app.js", after the section where we last imported the comments controller, we shall add the imports of the images and videos controller here.

```
// Import comments controller
var comments = require('./server/controllers/comments');
// Import videos controller
var videos = require('./server/controllers/videos');
// Import images controller
var images = require('./server/controllers/images');
```

15. In the section where we last setup the routes for handling comments page requests, add the routes for handling images and videos page requests.

```
// Setup routes for videos
app.get('/videos', videos.hasAuthorization, videos.show);
app.post('/videos', videos.hasAuthorization, upload.single('video'), videos.uploadVideo);

// Setup routes for images
app.post('/images', images.hasAuthorization, upload.single('image'), images.uploadImage);
app.get('/images-gallery', images.hasAuthorization, images.show);
```

- Now we can save changes and launch the server app in Terminal: **node app**. Go to the browser and visit our website <http://localhost:3000>, login and go to Images and Videos pages and upload the sample images and videos from “sample files to upload.zip”. You can try uploading any image or videos as long as they are within the 1.5MB size limit. View the uploaded images and playback the uploaded videos as well.

The images and videos pages are now complete.

5. Chat function

- Finally, we will create a new Chat / Messaging page in our website based on the package “socket.io” for real time messaging functions. We will need to add the controller for chat, the database model, the ejs template to render the page and update app.js to setup the routes and real time message server.
- Go to Terminal in Visual Studio Code and install the “socket.io” package in the project: **npm install socket.io**.
- First we write the ejs template that will render the Chat page. In the folder **/server/views/pages**, create the file **chatMsg.ejs**. This code in the ejs template will create a page with 2 text fields to enter name and message to send.

```
<!doctype html>
<html>
<head>
<script src="/javascripts/socket.io.js"></script>
<% include ../partials/javascript %>
<% include ../partials/stylesheet %>
</head>
<body>
<% include ../partials/header %>
<div class="container">
  <br>
  <div class="jumbotron">
    <h1 class="display-4">Send Message</h1>
    <br>
    <input id="name" class="form-control" placeholder="Name">
    <br>
    <textarea id="message" class="form-control" placeholder="Message"></textarea>
    <br>
    <button id="send" class="btn btn-success">Send</button>
  </div>
  <div id="messages">
    <%data.forEach(function(msg){%>
      <h4> <%=msg.name%> </h4> <p> <%=msg.message%> </p>
    <%})%>
  </div>
</div>
<script>
  var socket = io()
```

4. This block of the code will insert the script into that page which sends the message that the user has typed, to the server and listen on the channel "message" for messages sent by the server. If the server sends any message back, it will update the page to display the message.

```
<script>
  var socket = io()
  $((() => {
    $("#send").click(()=>{
      var message = { name: $("#name").val(), message: $("#message").val() }
      postMessage(message)
    })
  })

  socket.on('message', addMessage)

  function addMessage(msg){
    $("#messages").append(`<h4> ${msg.name} </h4> <p> ${msg.message} </p>`)
  }

  function postMessage(message) {
    $.post('<%-url%>', message)
  }
</script>
</body>
</html>
```

5. We are now done with the ejs template for the Chat page. We will modify the main navigation links to add a link to the Chat page. Open the file "**header.ejs**" in **/server/views/partials**. Add the following code to create a new link **"/messages"** for Chat:

```
<li class="nav-item">
  <a class="nav-link" href="/images-gallery">Photos</a>
</li>
<li class="nav-item">
  <a class="nav-link" href="/messages">Chat</a>
</li>
</ul>
```

6. The ejs templates are updated for the chat, we will now create the database model for the chat. Create a new file "**chatMsg.js**" in **/server/models** and enter the following code.


```

// models/chatMsg.js
var myDatabase = require('../controllers/database');
var sequelize = myDatabase.sequelize;
var Sequelize = myDatabase.Sequelize;

const ChatMsg = sequelize.define('ChatMsg', {
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  name: {
    type: Sequelize.STRING
  },
  message: {
    type: Sequelize.STRING,
    allowNull: false,
    defaultValue: '',
    trim: true
  }
});

// force: true will drop the table if it already exists
ChatMsg.sync({ force: false, logging: console.log }).then(() => {
  // Table created
  console.log("ChatMsgs table synced");
});

module.exports = sequelize.model('ChatMsg', ChatMsg);

```

7. Now that the database model is created, we will modify the “**app.js**” to include a new route for Chat and also initialize socket.io on the server side to listen for messages sent from browser clients, save the messages to database and broadcast the messages to all the clients.
8. In “**app.js**”, after the part where we setup the routes for image page, insert these codes to initialize socket.io.

```

// Setup routes for images
app.post('/images', images.hasAuthorization, upload.single('image'), images.uploadImage);
app.get('/images-gallery', images.hasAuthorization, images.show);

// Setup chat
var io = require('socket.io')(httpServer);
var chatConnections = 0;
var ChatMsg = require('../server/models/chatMsg');

io.on('connection', function(socket) {
  chatConnections++;
  console.log("Num of chat users connected: "+chatConnections);

  socket.on('disconnect', function() {
    chatConnections--;
    console.log("Num of chat users connected: "+chatConnections);
  });
});

```

- Next, we setup the routes for the Chat page at `/messages`, which will go to the database and retrieves all the past messages that were stored inside. We will setup to process messages sent from browsers to `/messages`, and store them into the database.

```

app.get('/messages', function (req, res) {
  ChatMsg.findAll().then((chatMessages) => {
    res.render('chatMsg', {
      url: req.protocol + "://" + req.get("host") + req.url,
      data: chatMessages
    });
  });
});

app.post('/messages', function (req, res) {
  var chatData = {
    name: req.body.name,
    message: req.body.message
  }
  //Save into database
  ChatMsg.create(chatData).then((newMessage) => {
    if (!newMessage) {
      sendStatus(500);
    }
    io.emit('message', req.body)
    res.sendStatus(200)
  })
});

```

- Once `app.js` is modified, chat function is complete, we can test it out. Save all changes and go to Terminal to run the server app: **node app**. Go to the browser and open the url **http://localhost:3000**. The navigation links and menu should now have "Chat". Click on it will open the new Chat page at **http://localhost:3000/messages**. If you look at the console logs, you will see that 1 user is connected.

```

GET /stylesheets/style.css 200 18.895 ms - 1359
Num of chat users connected: 1

```

Open another tab in the browser and go to **http://localhost:3000/messages**. Now 2 chat users are connected, and you can type a name and message, hit the send button, and see both tabs get updated with the new message at the bottom of the page.

-- End --