WHAT TO DO ON JS BACKED EXAM AFTER SKELETON IS OPENED AND RESOURCES ARE DOWNLOADED

**# Application Requirements**

[] Include resources (Html & Css, etc.)

[] Adapt User model and user service, auth middleware to project requirements

[] Implement register/login pages, register/login/logout actions

[] Update config to match requirements (db name eg.)

[] Create models for project-specific data

[] Create data services and middlewares for project-specific data

[] Create page actions for project-specific functionality (incl. modular routers)

1. Include resources
   1. From static -> css
   2. Images
   3. Views
2. Replace main.hbs with the home view from new project.
   1. Get rid of … in urls
   2. Delete all but main and put {{{body}}}
   3. Edit navigation links with / or /auth/logout and so on
   4. In all templates: from html to .hbs and remove all but main in them
   5. Delete login and register hbs and replace them with new files
   6. If any old view files exist, delete them as well
3. In config – index.js change connection string database name
4. Update models:
   1. Update user model according to requirements
   2. Create other models if needed, example with validation:

const {model, Schema} = require('mongoose')

const schema = new Schema({

    title: {type:String, required: [true, 'Title is required']},

    description: {type:String, required:[true, 'Description is required'], maxLength: [50, 'Description must be less than 50 characters']},

    imageUrl: {type:String, required:[true, 'ImageUrl is required']},

    public: {type:Boolean, required:[true, 'Public is required'], default: false},

    createdAt: {type:Date, required:true, default:Date.now},

    likedPlays: [{type: Schema.Types.ObjectId, ref: 'User', default: []}],

    author: {type: Schema.Types.ObjectId, ref: 'User'}

})

module.exports = model('Play', schema)

1. Edit user.js service to reflect the change in User model
   1. Const user part -> add whatever has been added
   2. If [], start at empty array
2. Check auth.js to see if more validation should be added to register/login
   1. Example: if more than just username is needed
3. Adapt authController.js
   1. Add additional validations for register/login if needed
   2. Use guards if more are needed
   3. Use .bail() to stop at first validation
   4. Example validation:

router.post('/register', isGuest(),

    body('username').isLength({min:3}).withMessage('Username must be at least 3 chars.').bail()

    .isAlphanumeric().withMessage('Username must contain only English letters and digits'),

    body('password').isLength({min:3}).withMessage('Password must be at least 3 chars.').bail()

    .isAlphanumeric().withMessage('Username must contain only English letters and digits'),

    body('rePass').custom((value, {req}) => {

        if (value !=req.body.password) {

            throw new Error('Passwords do not match')

        }

        return true;

    })

* 1. For the auth errors -> improve message for errors

throw new Error (Object.values(errors).map(e =>e.msg).join('\n'))

* 1. Errors to const ctx:

errors: err.message.split('\n'),

1. In templates for **register and login**
   1. Add {{userData.username}} to values
   2. Add action and method
   3. Make sure template input fields have names
2. Install libraries
   1. Npm I to install all from package-json
   2. Install nodemon if necessary
3. Test to see if it works
4. In auth.js (middleware) adapt error throwing for No such user and Wrong password

async function login(username,password) {

    const user = await userService.getUserByUserName(username)

    if (!user) {

        const err = new Error('No such user')

        err.type = 'credential';

        throw err;

    }

    const hasMatch = await bcrypt.compare(password, user.hashedPassword)

    if (!hasMatch) {

        const err = new Error('Wrong password')

        err.type = 'credential';

        throw err;

    }

    return generateToken(user)

}

Then throw a custom error in the authController login part such as  
catch(err) {

        let errors = [err.message]

        if (err.type =='credential') {

            errors = ['Incorrect username or password']

        }

        const ctx = {

            errors,

            userData: {

                username:req.body.username

            }

        }

        res.render('login', ctx)

    }

1. Create the service of the new model -> all the actions performed with this model and the database
   1. Functions getAll, getById, create, edit, delete etc
   2. EXPORT THEM!
2. In storage middleware, import service to decorate context with it. Destructure

const playService = require('../services/play')

module.exports = () => (req,res,next) => {

    //TODO import and decorate services

    req.storage  = {

        ...playService

    }

    next()

}

1. Add storageMiddleware to express.js, mount it with app.use() and import it just like authMiddleware
2. Create the new controller which responds to the service with starting code such as

const router = require('express').Router()

const {body,validationResult} = require('express-validator')

const {isGuest, isUser} = require('../middlewares/guards')

module.exports = router;

1. Import and mount the controller in routes.js (config
2. In create template
   1. Edit action, method
   2. Put field names and values if needed
   3. Example including **checkbox, textarea**

        <main>

            <form class="theater-form" action="/play/create" method="POST">

                <h1>Create Theater</h1>

                <div>

                    <label for="title">Theater Title:</label>

                    <input type="text" name="title" placeholder="Theater name" value="{{playData.title}}">

                </div>

                <div>

                    <label for="description">Theater Description:</label>

                    <textarea placeholder="Description" >{{playData.description}}</textarea>

                </div>

                <div>

                    <label for="imageUrl">Image url:</label>

                    <input type="text" placeholder="Image Url"  name="imageUrl" value="{{playData.imageUrl}}">

                </div>

                <div class="check">

                    <input type="checkbox" name="public" id="check-box" {{#if playData.public}} checked{{/if}} >

                    <label for="check-box">Public</label>

                </div>

                <button class="btn" type="submit">Submit</button>

            </form>

        </main>

)

1. Create parsers.js in util folder
   1. These distinguish between mongoose validation error and express-validator error and give the corresponding error

function parseError(err) {

    //to parse validation error

    if (err.name =="ValidationError") {

        //checks if mongoose error

        return Object.values(err.errors).map(e=>e.properties.message)

    } else {

        return [err.message]

    }

}

module.exports = {

    parseError

}

1. Establish get Create + post Create using the parser and prepared template.
   1. createPlay function in play.js service -> INCLUES CHECK FOR UNIQUE NAME

async function createPlay(playData) {

    const pattern = new RegExp(`^${playData.title}$`, 'i')

    const existing = await Play.findOne({title: {$regex: pattern}})

    if (existing) {

        throw new Error('A play with this name already exists!')

    }

    const play = new Play(playData)

    await play.save();

    return play;

}

* 1. Example of playController.js get and post

router.get('/create', isUser(), (req,res) => {

    res.render('play/create')

})

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

    try{

        const playData = {

            title: req.body.title,

            description: req.body.description,

            imageUrl: req.body.imageUrl,

            public: Boolean(req.body.public),

            author: req.user.\_id

        }

        await req.storage.createPlay(playData)

        res.redirect('/')

    }

    catch(err) {

        const ctx = {

            errors: parseError(err),

            playData: {

                title: req.body.title,

                description: req.body.description,

                imageUrl: req.body.imageUrl,

                public: Boolean(req.body.public),

            }

        }

        res.render('play/create',ctx)

    }

})

1. Create homeController.js and import it in routes <HERE>

const router = require('express').Router()

router.get('/', (req,res)=> {

    res.render('home')

})

module.exports = router;

1. Visualize catalog/home page:
   1. Tackle functions in service (to get from database) – play.js

async function getAllPlays() {

    return await Play.find({}).lean() //w/o getters, setters, etc

}

* 1. Get them in the controller

const router = require('express').Router()

router.get('/', async (req,res)=> {

    const plays = await req.storage.getAllPlays();

    res.render('home', {plays})

})

module.exports = router;

* 1. Create partial for each catalog item using the home template

                    <div>

                        <div class="home-image">

                            <img src="{{iamgeUrl}}"

                                alt="Card image cap">

                        </div>

                        <div class="info">

                            <h4>{{title}}</h4>

                            <div class="info-buttons">

                                <a class="btn details" href="/plays/details/{{\_id}}">Details</a>

                                <span class="likes">5 likes</span>

                            </div>

                        </div>

                    </div>

* 1. Apply partial to home

                    {{#each plays}}

                     {{> play}}

                    {{else}}

                    <h4>No plays yet...</h4>

                    {{/each}}

1. NAVIGATIOn for user/non-user:
   1. In auth.js put user info in res.locals.user = userData in parseToken so it’s visible to handlebars
   2. Example navigation for user

 <ul>

                <li><a href="/">Home </a></li>

                {{#if user}}

                <li><a href="/auth/logout">Logout</a></li>

                {{else}}

                <li><a href="/auth/login">Login</a></li>

                <li><a href="/auth/register">Register</a></li>

                {{/if}}

            </ul>

1. If items must be filtered by a particular property of theirs, use code such as in their service

 return await Play.find({public:true}).lean()

1. Sorting ascending/descending

return await Play.find({public:true}).sort({createdAt: -1}).lean()

1. ITEM DETAILS functionality
   1. Create getPlayById in the play.js service

async function getPlayById(id) {

    return Play.findById(id).lean()

}

1. In playController, make route for /play/details
   1. Make sure to establish logic for whether there is a user, an author, an owner etc.

router.get('/details/:id', async (req,res) => {

    try {

        const play = await req.storage.getPlayById(req.params.id)

        console.log(play)

        console.log(req.user)

        play.hasUser = Boolean(req.user)

        // attach variables to play -> if there is a user and this user is the play's author

        play.isAuthor = req.user && req.user.\_id == play.author

        // if the populated usersLiked includes the current user (so they cant like it again)

        play.liked = req.user && play.usersLiked.find(u => u.\_id == req.user.\_id)

        res.render('play/details', {play})

    }

    catch(err) {

        console.log(err.message)

        res.redirect('/404')

    }

})

* 1. Establish logic for author/likes and so on
     1. Make sure to populate necessary fields (example – users that liked a particular play)

async function getPlayById(id) {

    return await Play.findById(id).populate('usersLiked').lean()

}

* 1. Make details template reflect that logic

        <main>

            {{#with play}}

            <div class="container">

                <section class="details">

                    <h1>Theater title: {{title}}</h1>

                    <div>

                        <img src="{{imageUrl}}" />

                    </div>

                </section>

                <section class="details">

                    <h3>Theater Description</h3>

                    <p>{{description}}</p>

                    {{#if hasUser}}

                    <div class="buttons">

                        {{#if isAuthor}}

<a class="btn delete" href="/play/delete {{\_id}}">Delete</a>

                        <a class="btn edit" href="/play/edit {{\_id}}">Edit</a>

                        {{else}}

                            {{#if liked}}

                        <span class="liked">You have already liked this play!</span>

                        {{else}}

                        <a class="btn like" href="/play/like/{{\_id}}">Like</a>

                            {{/if}}

                        {{/if}}

                    </div>

                    {{/if}}

                </section>

            </div>

            {{/with}}

        </main>

1. Delete functionality without a delete page
   1. In play.js service create DB function to delete

async function deletePlay(id) {

    return  Play.findByIdAndDelete(id)

}

* 1. Create delete in controller, GUARDS && Check if it’s author

router.get('/delete/:id', isUser(), async (req,res) => {

    //delete in the case of no delete page just button

    try {

        const play = await req.storage.getPlayById(req.params.id)

        // check if this is the play's author

        if (play.author != req.user.\_id) {

            throw new Error ('Cannot delete a play you did not create!')

        }

        await req.storage.deletePlay(req.params.id)

        res.redirect('/')

    }catch(err) {

        console.log(err.message)

        res.redirect('/play/details/' + req.params.id)

    }

})

1. Edit functionality:
   1. First create edit function in service

async function editPlay(id,playData) {

    const play = await Play.findById(id)

    play.title = playData.title

    play.description = playData.description

    play.imageUrl = playData.imageUrl

    play.public = Boolean(playData.public)

    return await play.save();

}

* 1. Create edit GET in controller – make sure to put Guard and check if user is author!

router.get('/edit/:id', isUser(), async (req,res) => {

    try {

    const play = await req.storage.getPlayById(req.params.id)

    // check if this is the play's author

    if (play.author != req.user.\_id) {

        throw new Error ('Cannot edit a play you did not create!')

    }

    res.render('play/edit', {play})

    }

    catch(err) {

        console.log(err.message)

        res.redirect('/play/details/' + req.params.id)

    }

})

* 1. Edit POST – Guard, author, parse errors

router.post('/edit/:id', isUser(), async (req,res) => {

    try {

    const play = await req.storage.getPlayById(req.params.id)

    // check if this is the play's author

    if (play.author != req.user.\_id) {

        throw new Error ('Cannot edit a play you did not create!')

    }

    await req.storage.editPlay(req.params.id, req.body)

    res.redirect('/')

    }

    catch(err) {

        // so error can be populated correctly via parser

        const ctx=  {

            errors: parseError(err),

            play: {

                \_id: req.params.id,

                title: req.body.title,

                description: req.body.description,

                imageUrl: req.body.imageUrl,

                public: Boolean(req.body.public)

            }

        }

        console.log(err.message)

        res.render('play/edit', ctx)

    }

})

* 1. Edit the view so it preloads play data
     1. Values
     2. Make sure fields have a name
     3. Action, method
     4. Logic if needed

        <main>

            <form class="theater-form" action="/play/edit/{{play.\_id}}" method="POST">

                <h1>Edit Theater</h1>

                <div>

                    <label for="title">Theater Title:</label>

                    <input type="text" name="title" placeholder="Theater name"value="{{play.title}}">

                </div>

                <div>

                    <label for="description">Theater Description:</label>

                    <textarea name="description" placeholder="Description" >{{play.description}}</textarea>

                </div>

                <div>

                    <label for="imageUrl">Image url:</label>

                    <input type="text" name="imageUrl" placeholder="Image Url" value="{{play.imageUrl}}">

                </div>

                <div class="check">

                    <input type="checkbox" name="public" id="check-box" {{#if play.public}} checked{{/if}} >

                    <label for="check-box">Public</label>

                </div>

                <button class="btn" type="submit">Submit</button>

            </form>

        </main>

1. Likes functionality
   1. Create like function in service (gets id of play and of user), export it in module exports

async function likePlay(id, userId) {

    const play = await Play.findById(id)

    play.usersLiked.push(userId)

    return play.save();

}

* 1. Create a get function for the likes in playController, check if author is user,put guards

router.get('/like/:id', isUser(), async (req,res) => {

    try {

        const play = await req.storage.getPlayById(req.params.id)

        // check if this is the play's author

        if (play.author == req.user.\_id) {

            throw new Error ('Cannot like your own play!')

        }

        await req.storage.likePlay(req.params.id, req.user.\_id)

        res.redirect('/play/details/' + req.params.id)

        }

        catch(err) {

            console.log(err.message)

            res.redirect('/play/details/' + req.params.id)

        }

})

* 1. Use the length of usersLiked in partial or details (depends on where it has to be shown)

<span class="likes">{{usersLiked.length}}</span>