**App.js**

1. Import all required modules and dependencies:

- express, mongoose, path, method-override, ejs-mate, routers, utilities, session, flash, schemas, models

2. Set up MongoDB connection string.

3. Connect to MongoDB using mongoose.

- If successful, log "Connected to DB"

- If error, log the error

4. Set up Express app configuration:

- Set view engine to ejs

- Set views directory

- Parse URL-encoded bodies

- Use method-override for HTTP verbs

- Set ejs-mate as the engine for ejs files

- Serve static files from /public

5. Configure session options (secret, cookie settings, etc.)

6. Use session and flash middleware.

7. Middleware to set flash messages in res.locals for use in templates.

- Store success and error messages from flash in res.locals

- Log these messages for debugging

8. Define a root route ("/") that sends a simple response.

9. Define a validateReview middleware:

- Validate request body against reviewSchema

- If error, throw ExpressError with error message

- Otherwise, call next()

10. Mount routers:

- Mount listings router at "/listings"

- Mount reviews router at "/listings/:id/reviews"

11. Error-handling middleware:

- Deconstruct statusCode and message from error

- Render error.ejs with error object

12. Start the server on port 8000 and log a message.

**listings.js**

1. Import required modules and utilities:

- express

- wrapAsync (for wrapping async functions with error handling)

- ExpressError (custom error class)

- listingSchema (Joi schema for validation)

- Listing (Mongoose model for listings)

2. Create Express Router instance

3. Define a middleware function `validateListing`:

- Validate the request body using `listingSchema`

- If validation fails, throw an ExpressError with message

- Else, call next() to continue

4. ROUTES:

[NEW Route]

- GET /new

- Render a form to create a new listing

[SHOW Route]

- GET /:id

- Get listing by ID from DB

- Populate 'reviews' associated with the listing

- If listing not found:

- Flash an error message

- Redirect to /listings

- Else:

- Render the show view with listing details

[INDEX Route]

- GET /

- Fetch all listings from DB

- Render the index view to show all listings

[CREATE Route]

- POST /

- Use validateListing middleware to validate input

- Create a new listing from request body

- Save to DB

- Flash success message

- Redirect to /listings

[EDIT Route]

- GET /:id/edit

- Find listing by ID

- If not found:

- Flash error and redirect

- Else:

- Render edit form with listing data

[UPDATE Route]

- PUT /:id

- Use validateListing middleware

- Update the listing with form data

- If not found:

- Throw ExpressError

- Else:

- Flash success message

- Redirect to /listings/:id

[DELETE Route]

- DELETE /:id

- Find listing by ID and delete it

- If not found:

- Throw ExpressError

- Else:

- Flash success message

- Redirect to /listings

5. Export the router

**review.js**

1. Import express and create a router with mergeParams enabled.

2. Import ExpressError, wrapAsync, reviewSchema, Review model, and Listing model.

3. Define validateReview middleware:

- Validate req.body against reviewSchema.

- If validation fails, throw an ExpressError with the error message.

- Otherwise, call next().

4. Define POST route for creating a review:

- Validate the review using validateReview middleware.

- Use wrapAsync to handle async errors.

- Find the listing by id from req.params.

- If listing not found, throw a 404 ExpressError.

- Create a new Review from req.body.review.

- Push the new review into the listing's reviews array.

- Save the new review and the listing.

- Flash a success message.

- Redirect to the listing's show page.

5. Define DELETE route for deleting a review:

- Use wrapAsync to handle async errors.

- Extract id and reviewsId from req.params.

- Delete the review from the Review collection by reviewsId.

- Remove the review reference from the listing's reviews array using $pull.

- Flash a success message.

- Redirect to the listing's show page.

6. Export the router.

**user.js**

1. Import mongoose and extract Schema from it.

2. Import passport-local-mongoose plugin.

3. Define a new Schema called UserSchema with:

- email: String, required

4. Add the passport-local-mongoose plugin to UserSchema.

- This plugin automatically adds username, password hash, and salt fields, and provides authentication methods.

5. Create a User model from UserSchema.

6. Export the User model.

**schema.js**

1. Import Joi validation library.

2. Define and export listingSchema:

- listing: object (required)

- title: string (required)

- description: string (required)

- price: number (required, minimum 0)

- country: string (required)

- location: string (required)

- image: object (required)

- url: string (can be empty or null)

3. Define and export reviewSchema:

- review: object (required)

- comment: string (required)

- rating: number (required, minimum 1, maximum 5)

**show.ejs**

1. Use main layout boilerplate:

- <% layout("/layouts/boilerplate") %>

2. Main container row with some top margin:

- <div class="row mt-3">

3. Listing Title Section:

- Centered using col-8 offset-3

- Display listing.title as a heading

4. Listing Details Card:

- Bootstrap card with image, description, price, location

- listing.image.url is shown as the card image

- Price is shown with INR symbol using toLocaleString("en-IN")

- Also shows listing location and country

5. Action Buttons (Edit/Delete):

- Flex row with Edit (anchor link) and Delete (form using method override)

- Buttons styled using Bootstrap dark theme

6. Review Submission Form:

- `action="/listings/:id/reviews"` → Posts to backend

- Uses `review[rating]` and `review[comment]` fields

- `type="range"` for rating (1 to 5)

- `textarea` for comments (with validation)

- Button to submit review

7. Display all Reviews:

- Loop through `listing.reviews`

- For each review:

- Display comment, rating badge

- Static name "Sukanya" (can be dynamic later)

- Delete button for each review (`/listings/:id/reviews/:reviewId`)

8. Close all open divs

**ExpressError.js**

1. Define a class ExpressError that extends the built-in Error class.

2. In the constructor, accept statusCode and message as arguments.

- Call the parent Error constructor.

- Set the statusCode and message properties on the instance.

3. Export the ExpressError class.

wrapAsync.js

1. Export a function that takes an async function (fn) as an argument.

2. Return a new function that takes req, res, and next as arguments.

3. Call the original async function (fn) with req, res, and next.

4. If fn throws an error (promise is rejected), catch the error and pass it to next().

**Script.js**

1. Use an IIFE (Immediately Invoked Function Expression) to create a private scope and enable strict mode.

2. Select all forms with the class 'needs-validation'.

3. For each selected form:

a. Add a 'submit' event listener.

b. When the form is submitted:

i. If the form is not valid:

- Prevent the default form submission.

- Stop the event from propagating further.

ii. Add the 'was-validated' class to the form to trigger Bootstrap's validation styles.