Code Summary Without Canvas

Here are the notes for the provided code snippet:

Code Overview:

The code defines a **Mongoose schema and model** for a "Listing" object.

Key Points:

1. **Dependencies:**
   - `mongoose` is required to define the schema and interact with the MongoDB database.
2. **Schema Definition:**
   - `listingSchema`: A schema for a listing object with various fields:
     - `title`:
       - Type: `String`
       - Required: `true`
     - `description`:
       - Type: `String`
       - Optional.
     - `image`:
       - Type: `String`
       - Default value:
         A fallback URL for an image is provided.
       - `set` **property**:
         If an empty string (`""`) is passed as the value, it is replaced with another image URL.

        - Ternary operator is used to handle the condition.
    - `price`:
        - Type: `Number`
        - Optional.
    - `location`:
        - Type: `String`
        - Optional.
    - `country`:
        - Type: `String`
        - Optional.
3. **Model Creation:**
    - A Mongoose model named `Listing` is created using the schema.
    - Model name: `'listing'`
    - Exports the model using `module.exports`.
4. **Default Values and Validation:**
    - The schema ensures `title` is required and provides default values for the `image` field when necessary.

Example Use Case:

- To create a new listing:

```javascript
const newListing = new Listing({
  title: "Beautiful Beach House",
  description: "A house by the beach with a stunning view.",
  price: 250000,
  location: "Miami, FL",
  country: "USA",
});
```

```
newListing.save().then(() => console.log("Listing saved!"));
```

Key Notes:

- The `set` function on `image` ensures a fallback image URL when an empty string is provided.
- `mongoose.model` binds the schema to the MongoDB collection named `listings`.

Let me know if you'd like me to add more details!

Here's a detailed explanation using **code snippets** from your provided example for better clarity:

---

1. Dependencies and Setup

Code:

```javascript
const express = require('express');
const mongoose = require('mongoose');
const Listing = require('./models/listing');
const path = require('path');
const methodOverride = require('method-override');
const ejsMate = require('ejs-mate');
const wrapAsync = require('./utils/wrapAsync');
const ExpressError = require('./utils/ExpressError');
const { listingSchema } = require('./schema.js');
```

- **Purpose**: Import essential dependencies:
  - `express`: Core framework for routing and handling requests.
  - `mongoose`: MongoDB connection and model handling.
  - `path`: To resolve directory paths.
  - `methodOverride`: Allows HTTP `PUT` / `DELETE` methods via query parameters.

- `ejsMate`: Simplifies layouts and partials for EJS templates.
- `wrapAsync` and `ExpressError`: Custom utilities for async error handling and structured errors.
- `listingSchema`: A Joi schema for validating data.

---

2. Database Connection

Code:

```javascript
const db_url = 'mongodb://127.0.0.1:27017/Air';

async function main() {
  await mongoose.connect(db_url);
}

main()
  .then(() => {
    console.log('Connected to DB');
  })
  .catch((err) => {
    console.log(err);
  });
```

- **Purpose**: Connect to the MongoDB database named `Air`.
- **Explanation**:
  - Uses `mongoose.connect()` with the local MongoDB URL.
  - Logs success or failure in the console.

---

3. Middleware Setup

Code:

```javascript
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(express.static(path.join(__dirname, 'public')));
app.use(methodOverride('_method'));
app.engine('ejs', ejsMate);
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

- **Purpose**: Configure the app with:
  - **Views and static files**:
    - EJS templates are in the `views` directory.
    - Static files (e.g., CSS/JS) are served from the `public` directory.
  - **Method Override**: Allows using `_method` in query strings for `PUT`/`DELETE` requests.
  - **EJS Mate**: Enables layout support for EJS templates.
  - **Form Parsing**:
    - Parses incoming JSON and form data.

---

4. Validation with Joi

Code:

```javascript
const validateListing = (req, res, next) => {
  let { error } = listingSchema.validate(req.body);
  if (error) {
    let errMsg = error.details.map((el) => el.message).join(',');
    throw new ExpressError(404, errMsg);
  } else {
    next();
  }
};
```

- **Purpose**: Middleware to validate incoming request data.
- **Explanation**:
  - `listingSchema.validate(req.body)` checks if the data matches the Joi schema.
  - If validation fails:
    - Collects error messages.
    - Throws a custom `ExpressError` with the 404 status.
  - Calls `next()` to continue to the next middleware if validation passes.

---

5. Routes

5.1. Index Route

```javascript
app.get(
  '/listings',
  wrapAsync(async (req, res) => {
    let data = await Listing.find();
    res.render('listings/index', { data });
  })
);
```

- **Purpose**: Fetch all listings from the database and render the `index` view.
- **Explanation**:
  - Fetches data using `Listing.find()`.
  - Renders the `listings/index.ejs` file with the fetched data.

5.2. Create Route

```javascript
```

```
app.post(
  '/listings',
  validateListing,
  wrapAsync(async (req, res) => {
    let data = req.body;
    let newListing = new Listing(data);
    await newListing.save();
    res.redirect('/listings');
  })
);
```

- **Purpose**: Create a new listing after validating the data.
- **Explanation**:
  - Validates the request body with `validateListing`.
  - Saves a new listing using `new Listing(data).save()`.
  - Redirects to `/listings` after saving.

5.3. Show Route

```javascript
app.get(
  '/listings/:id',
  wrapAsync(async (req, res) => {
    let { id } = req.params;
    let place = await Listing.findById(id);
    res.render('listings/show', { place });
  })
);
```

- **Purpose**: Fetch and display details of a specific listing.
- **Explanation**:
  - Fetches a listing by its `id` using `Listing.findById()`.
  - Renders the `listings/show.ejs` file with the listing's details.

5.4. Edit and Update Routes

```javascript
app.get(
  '/listings/:id/edit',
  wrapAsync(async (req, res) => {
    let { id } = req.params;
    let listing = await Listing.findById(id);
    res.render('listings/edit', { listing });
  })
);

app.put(
  '/listings/:id',
  validateListing,
  wrapAsync(async (req, res) => {
    let { id } = req.params;
    let data = req.body;
    await Listing.findByIdAndUpdate(id, data);
    res.redirect(`/listings/${id}`);
  })
);
```

- **Purpose**:
  - **Edit Route**: Fetch a listing and render an edit form.
  - **Update Route**: Update the listing in the database.
- **Explanation**:
  - Uses `findById()` to fetch and `findByIdAndUpdate()` to update listings.

5.5. Delete Route

```javascript
```

```javascript
app.delete(
  '/listings/:id',
  wrapAsync(async (req, res) => {
    let { id } = req.params;
    await Listing.findByIdAndDelete(id);
    res.redirect('/listings');
  })
);
```

- **Purpose**: Deletes a specific listing and redirects to the index route.
- **Explanation**:
  - Uses `findByIdAndDelete()` to remove the listing from the database.

---

6. Error Handling

Code:

```javascript
app.all('*', (req, res, next) => {
  next(new ExpressError(404, 'Page Not Found'));
});

app.use((err, req, res, next) => {
  let { statusCode = 500, message = 'Something went wrong' } = err;
  res.status(statusCode).render('Error', { message });
});
```

- **Purpose**:
  - Handle undefined routes with a 404 error.
  - Provide error messages for all errors using middleware.
- **Explanation**:
  - `app.all('*')`: For any unrecognized route.
  - Error-handling middleware renders the `Error.ejs` template.

7. Listening on Port

Code:

```javascript
app.listen(port, () => {
  console.log(`Example app listening on port ${port}!`);
});
```

- **Purpose**: Starts the server on port `3000` and logs the startup.

Let me know if you'd like further clarification or examples!

Here's an explanation of the provided **Joi schema** code and its purpose:

Code Explanation

Code:

```javascript
const Joi = require('joi');

module.exports.listingSchema = Joi.object({
  title: Joi.string().required(),
  description: Joi.string().required(),
  image: Joi.string().allow('', null),
  price: Joi.number().required().min(0),
  location: Joi.string().required(),
  country: Joi.string().required(),
}).required();
```

Purpose of the Code

- The code defines a **validation schema** using the **Joi** package to ensure that incoming data conforms to specific requirements before being saved to the database.
- This is exported as `listingSchema` and used in the application for validating data in routes like `POST` and `PUT`.

Schema Breakdown

1. `title`

   ```javascript
   title: Joi.string().required(),
   ```

   - Must be a **string**.
   - **Required**: The validation will fail if `title` is missing.

2. `description`

   ```javascript
   description: Joi.string().required(),
   ```

   - Must be a **string**.
   - **Required**: The validation will fail if `description` is missing.

3. `image`

   ```javascript
   image: Joi.string().allow('', null),
   ```

   - Must be a **string**.
   - **Optional**: Can be an empty string `''` or `null` if no image is provided.
   - This is useful for providing a default image in the Mongoose schema if none is given.

4. `price`

   ```javascript
   ```

```
price: Joi.number().required().min(0),
```

- Must be a **number**.
- **Required**: The validation will fail if `price` is missing.
- **Min Value**: Must be greater than or equal to `0`.

5. `location`

```javascript
location: Joi.string().required(),
```

- Must be a **string**.
- **Required**: The validation will fail if `location` is missing.

6. `country`

```javascript
country: Joi.string().required(),
```

- Must be a **string**.
- **Required**: The validation will fail if `country` is missing.

Wrapping the Schema

```javascript
}).required();
```

- The entire object is wrapped in `.required()`, ensuring that the request body itself is present during validation.

Example of Valid Data

```json
{
  "title": "Beautiful Apartment",
  "description": "A cozy apartment in the city center.",
  "image": "",
  "price": 100,
  "location": "Downtown",
  "country": "USA"
}
```

Example of Invalid Data

1. **Missing `title`:**

   ```json
   {
     "description": "Cozy apartment.",
     "price": 100,
     "location": "Downtown",
     "country": "USA"
   }
   ```

   - Validation Error: **"title" is required.**

2. **Negative `price`:**

   ```json
   {
     "title": "Beautiful Apartment",
     "description": "A cozy apartment.",
     "price": -10,
     "location": "Downtown",
     "country": "USA"
   }
   ```

- Validation Error: **"price" must be greater than or equal to 0.**

Integration in the App

This schema is used in routes, such as the `POST` and `PUT` routes, via the middleware `validateListing`:

```javascript
const validateListing = (req, res, next) => {
  let { error } = listingSchema.validate(req.body);
  if (error) {
    let errMsg = error.details.map((el) => el.message).join(',');
    throw new ExpressError(404, errMsg);
  } else {
    next();
  }
};
```

- Ensures that:
  - Data sent in `req.body` matches the schema.
  - Errors are thrown if validation fails, preventing invalid data from being saved.

Let me know if you want further examples or clarification!

Explanation of the Code

Code Snippet:

```javascript
module.exports = (fn) => {
  return (req, res, next) => {
    fn(req, res, next).catch(next);
  };
};
```

This is a **utility function** used for handling asynchronous errors in **Express.js** routes. Let's break it down:

Purpose

- Wraps an asynchronous function `fn` to handle potential errors without needing to write repetitive `try-catch` blocks in every route.
- Automatically passes any errors to the next middleware (like an error handler).

How It Works

1. **Input**:
   - Takes an asynchronous function `fn` as its parameter.
   - Example: `async (req, res, next) => { /* logic */ }`.
2. **Returns**:
   - A new function `(req, res, next)` that:
     - Calls the original function `fn` with the same arguments (`req`, `res`, `next`).
     - Uses `.catch(next)` to catch any errors from the promise returned by `fn` and pass them to the `next` middleware (error handler).
3. **Error Handling**:
   - If the `fn` function throws an error or rejects a promise, `.catch(next)` ensures that the error is passed to Express's built-in error handling mechanism.

Why It's Useful

1. **Simplifies Code**:
   - Instead of manually wrapping every route in a `try-catch`, you can use this wrapper.
   - Example (without the utility):

     ```javascript
     app.get('/example', async (req, res, next) => {
       try {
         // Asynchronous logic
       } catch (err) {
         next(err);
       }
     });
     ```

   - Example (with the utility):

     ```javascript
     const wrapAsync = require('./utils/wrapAsync');

     app.get('/example', wrapAsync(async (req, res, next) => {
       // Asynchronous logic
     }));
     ```

2. **Centralized Error Management**:
   - Keeps routes clean by offloading error handling to this utility function.

Example Usage

Code:

```javascript
const wrapAsync = require('./utils/wrapAsync');

app.get(
  '/listings',
  wrapAsync(async (req, res) => {
    let listings = await Listing.find();
```

```
    res.render('listings/index', { listings });
  })
);
```

- **What Happens**:
  - The route logic is asynchronous.
  - If an error occurs (e.g., `Listing.find()` fails), it is caught by `.catch(next)` and passed to the error-handling middleware.

Flow Overview

1. **Request Handling**:
   - Request reaches the route.
   - The wrapped function (`fn`) is executed.
2. **Error Occurrence**:
   - If `fn` encounters an error (e.g., DB query fails), `.catch(next)` captures it.
3. **Passing Error**:
   - The error is passed to the next middleware (usually a centralized error handler).
4. **Error Handling Middleware**:
   - The app's error handler processes the error and sends an appropriate response.

Example Error Handler

Code:

```javascript
app.use((err, req, res, next) => {
  const { statusCode = 500, message = 'Something went wrong' } = err;
  res.status(statusCode).render('error', { message });
});
```

- Ensures that all errors (caught by `wrapAsync`) are handled properly, providing a user-friendly response.

Key Takeaway

- `wrapAsync` reduces boilerplate code, simplifies error handling in asynchronous routes, and ensures a consistent way to manage errors in an Express.js application.

Explanation of the Code

Code Snippet:

```javascript
class ExpressError extends Error {
  constructor(statusCode, message) {
    super();
    this.statusCode = statusCode;
    this.message = message;
  }
}

module.exports = ExpressError;
```

Purpose

This code defines a custom error class called `ExpressError`, which is an extension of JavaScript's built-in `Error` class. It is specifically designed for use in Express.js applications to handle and customize errors consistently.

How It Works

1. **Inherits the `Error` Class:**
   - The `ExpressError` class uses the `extends` keyword to inherit properties and methods from the built-in `Error` class.
   - This means that `ExpressError` objects will have the same behavior as regular error objects, but with additional custom properties (`statusCode` and `message`).
2. **Constructor Parameters**:
   - `statusCode`: The HTTP status code (e.g., 404 for "Not Found", 500 for "Internal Server Error").
   - `message`: A descriptive error message.
3. **Initialization**:
   - Calls the parent `Error` constructor with `super()`.
   - Sets `statusCode` and `message` on the error instance, so they can be accessed later when handling errors.
4. **Export**:
   - The class is exported using `module.exports`, so it can be imported and used throughout the application.

Why Use It?

1. **Simplifies Error Management**:
   - Instead of manually creating error objects with `statusCode` and `message` everywhere, you can use this class to standardize error creation.
2. **Improves Code Readability**:
   - Makes it clear that the error is related to Express.js (hence the name `ExpressError`).
3. **Customizable Errors**:
   - Allows you to easily add other custom properties if needed in the future.

Example Usage

Throwing an Error:

```javascript
const ExpressError = require('./utils/ExpressError');

app.all('*', (req, res, next) => {
  throw new ExpressError(404, 'Page Not Found');
});
```

- **What Happens**:
  - When a request is made to an undefined route, this middleware throws an `ExpressError` with:
    - `statusCode`: `404`
    - `message`: `"Page Not Found"`

Catching Errors:

```javascript
app.use((err, req, res, next) => {
  const { statusCode = 500, message = 'Something went wrong' } = err;
  res.status(statusCode).render('error', { message });
});
```

- **How It Works**:
  - The error-handling middleware catches the `ExpressError` instance.
  - Accesses the `statusCode` and `message` properties to determine the appropriate response.
  - Renders an error page with the provided message.

Flow Overview

1. **Error Creation**:
    - When something goes wrong (e.g., invalid user input or a missing resource), an `ExpressError` is created with a specific `statusCode` and `message`.
2. **Error Throwing**:
    - The `ExpressError` instance is thrown using `throw new ExpressError()`.
3. **Error Handling**:
    - The error-handling middleware catches the error.
    - The `statusCode` and `message` properties are used to send a user-friendly error response.

Example Scenario

Invalid Data in a Form

- **Validation Middleware**:

    ```javascript
    const validateListing = (req, res, next) => {
      const { error } = listingSchema.validate(req.body);
      if (error) {
        const errMsg = error.details.map((el) => el.message).join(',');
        throw new ExpressError(400, errMsg);
      }
      next();
    };
    ```

- **What Happens**:
    - If the `listingSchema` validation fails, an `ExpressError` is thrown with:
        - `statusCode`: `400`
        - `message`: A list of validation error messages.
- **Error Response**:
    - The error-handling middleware uses the `statusCode` and `message` to send a meaningful response to the client.

Benefits

1. Consistent error handling throughout the application.
2. Makes error messages and HTTP status codes more predictable and easier to debug.
3. Encourages a clean separation between business logic and error handling.

Let me know if you'd like examples or further clarifications!

Explanation of the HTML Template

This is a template file written in **EJS (Embedded JavaScript)**. It is often used in Node.js applications with the Express framework to render dynamic HTML content.

Code Breakdown

HTML Boilerplate

```html
<!DOCTYPE html>
<html lang="en">
```

- **DOCTYPE**: Declares the document type as HTML5.
- `lang="en"`: Sets the language of the document to English, improving accessibility and SEO.

Head Section

The `<head>` contains metadata and links to external stylesheets and fonts.

1. **Character Encoding**:

```html
<meta charset="UTF-8">
```

- Sets the character encoding to UTF-8, allowing the display of special characters.

2. **Viewport Settings**:

   ```html
   <meta name="viewport" content="width=, initial-scale=1.0">
   ```

   - Ensures the page scales properly on different screen sizes (mobile-friendly). The `width=` part is incomplete and should be corrected to `width=device-width`.

3. **Title**:

   ```html
   <title>WandarLust</title>
   ```

   - Sets the title of the web page to "WandarLust."

4. **Stylesheets**:

   - **Custom Stylesheet**:

     ```html
     <link rel="stylesheet" href="/css/style.css">
     ```

     - Links to a local CSS file for custom styles.

   - **Bootstrap**:

     ```html
     <link
     href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
     rel="stylesheet" integrity="...">
     ```

     - Includes the Bootstrap CSS framework for styling and responsive design.

   - **Font Awesome**:

     ```html
     ```

```html
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.7.2/css/all.min.css" integrity="...">
```

  - Adds Font Awesome icons for use in the application.

- **Google Fonts**:

  ```html
  <link href="https://fonts.googleapis.com/css2?family=Plus+Jakarta+Sans:ital,wght@0,200..800;1,200..800&display=swap" rel="stylesheet">
  ```

  - Includes the "Plus Jakarta Sans" font from Google Fonts.

Body Section

1. **Navbar Inclusion**:

   ```html
   <%- include("../includes/navbar") %>
   ```

   - Embeds the `navbar` partial located in the `includes` directory.
   - The `<%-` syntax allows raw HTML from the partial to be included.

2. **Main Content Area**:

   ```html
   <div class="container">
       <%- body %>
   </div>
   ```

   - Wraps the main content (dynamic data passed from the server) in a Bootstrap `.container` class for consistent spacing.
   - `<%- body %>` renders raw HTML content passed by the server (e.g., from a route).

3. **Footer Inclusion**:

```html
<%- include("../includes/footer") %>
```

- Embeds the `footer` partial located in the `includes` directory.

4. **Bootstrap and Custom Scripts**:

```html
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
integrity="..."></script>
<script src="/js/script.js"></script>
```

- **Bootstrap JavaScript**: Adds interactivity (e.g., modals, dropdowns) provided by Bootstrap.
- **Custom JavaScript**: Links to a local JavaScript file for custom functionality.

Dynamic Content with EJS

1. **Partials**:
   - The `navbar` and `footer` are **partials** included in the main layout.
   - Partials are reusable pieces of HTML that keep the code modular and maintainable.
2. **Dynamic `body`**:
   - The `<%- body %>` placeholder renders content dynamically.
   - This is typically used to display different pages within the same layout.

Key Concepts

1. **Template Layout**:
   - This file acts as a layout for the application, defining a consistent structure (head, navbar, footer) across all pages.

- Specific content for each page is injected dynamically into the `<%- body %>` section.

2. **Bootstrap Integration**:
   - The template leverages Bootstrap for responsive design and pre-styled components.
3. **EJS Syntax**:
   - `<%- %>`: Renders raw HTML without escaping.
   - `<%= %>`: Escapes special characters in HTML for security.

Improvement Suggestions

1. **Fix the Viewport Meta Tag**:

   ```html
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   ```

   - Correct the incomplete `width=` attribute.
2. **Favicon**:
   - Add a favicon for better branding:

     ```html
     <link rel="icon" href="/path/to/favicon.ico">
     ```
3. **Error Handling**:
   - Ensure that the included partials (`navbar`, `footer`) handle errors gracefully in case they fail to render.

Let me know if you need help with any of these suggestions!