**Listing Model (Core Model)**

**What is the Listing model?**

> **The Listing model represents a travel destination or stay that users can add to the platform.**

**Key fields**

- **Title**
- **Description**
- **Price**
- **Location**
- **Country**
- **Image (URL + filename from ImageKit)**
- **Owner (reference to User)**
- **Reviews (array of Review IDs)**

**Why this model is important**

- **It is the main entity of the application**
- **All CRUD operations revolve around listings**

**Relationships**

- **One listing → one owner (User)**
- **One listing → many reviews**

**Interview-style answer**

> **“The Listing model stores all travel-related information like title, price, location, and images. Each listing is linked to a user as the owner and can have multiple reviews.”**

## 1️⃣ Index Route (Read – All Listings)

### What it does

The **index route** is used to **show all listings** available on the platform.

### Route

```
GET /listings
```

### Flow (simple)

- User opens the home page
- Frontend sends a GET request
- Backend fetches all listings from MongoDB
- Data is sent back and displayed as cards

### Interview-style answer

> “The index route fetches all travel listings from the database and displays them on the homepage. It uses a GET request and returns an array of listings.”

---

## 2️⃣ Show Route (Read – Single Listing)

### What it does

The **show route** displays **details of a single listing**.

### Route

```
GET /listings/:id
```

### Flow

- User clicks on a listing
- Listing ID is sent in URL

- Backend finds the listing by ID
- Also fetches related reviews and owner info

**Interview-style answer**

> “The show route is used to display a single listing in detail. It takes the listing ID from the URL, fetches the data from MongoDB, and sends it to the frontend.”

---

### 3️⃣ Create Route (New + Create)

🔹 **New Route (Form)**

```
GET /listings/new
```

**Purpose**

> Shows the **form** to add a new listing.

**Security**

- Protected route
- Only logged-in users can access it

---

🔹 **Create Route (Submit Form)**

```
POST /listings
```

**Flow**

- User fills the form
- Image is uploaded using ImageKit
- Backend validates data
- Listing is saved in MongoDB with owner ID

**Interview-style answer**

> “The create route allows authenticated users to add a new listing. It handles form data, uploads images using ImageKit, validates input, and stores the listing in MongoDB.”

---

### 4️⃣ Update Route (Edit + Update)

🔹 **Edit Route (Form)**

```
GET /listings/:id/edit
```

**Purpose**

> Shows the edit form with **pre-filled data**.

**Security**

- Only the **owner of the listing** can access it

---

🔹 **Update Route (Submit Changes)**

```
PUT /listings/:id
```

**Flow**

- User edits details
- Backend checks ownership
- Updates listing in MongoDB

**Interview-style answer**

> “The update route allows the owner of a listing to modify its details. Before updating, ownership is verified to prevent unauthorized changes.”

---

### 5️⃣ Delete Route

**What it does**

Deletes a listing from the platform.

**Route**

`DELETE /listings/:id`

**Flow**

- User clicks delete
- Backend verifies user is owner
- Listing is removed from MongoDB
- Associated reviews are also handled

**Interview-style answer**

> “The delete route removes a listing from the database. It is protected so only the listing owner can delete it, ensuring data security.”

## 1️⃣ EJS-Mate (Template Engine)

**What is EJS-Mate?**

> **EJS-Mate** is a layout engine built on top of EJS that helps avoid code repetition.

**Why did you use it?**

> In WanderLust, many pages share common UI elements like the **navbar, footer, and Bootstrap links**.
> EJS-Mate allows me to define a **base layout** and reuse it across all pages.

**How it works**

- I created a `layout.ejs`
- Common components go there
- Individual pages inject content using `block` or `body`

**Interview-style answer**

> “I used EJS-Mate to maintain a consistent layout across all pages. It helps in reusing common components like navbar and footer and keeps the code clean and maintainable.”

---

## 2️⃣ Navbar Styling (Bootstrap + Custom CSS)

### Purpose

> The navbar provides **easy navigation** across the application.

### Features in your project

- Links: Home, Add Listing, Login, Register, Logout
- Dynamic rendering based on authentication
- Responsive for mobile

### Styling approach

- Used **Bootstrap navbar classes**
- Added custom CSS for spacing and branding

### Interview-style answer

> “The navbar is styled using Bootstrap to ensure responsiveness. I used conditional rendering in EJS to show different options based on whether the user is logged in or not.”

---

## 3️⃣ Footer Styling

### Purpose

> Footer gives a professional finish and displays static information.

### Implementation

- Common footer included via EJS-Mate layout
- Styled using Bootstrap utility classes

- Fixed spacing and alignment

**Interview-style answer**

> "The footer is implemented as a reusable component using EJS-Mate. It ensures consistency across pages and improves the overall UI."

---

## 4️⃣ Styling Index Page (All Listings)

**What is shown**

- Grid of listings
- Image, title, price, location
- Clickable cards

**Styling method**

- Bootstrap **cards and grid system**
- Responsive layout using rows and columns
- Hover effects using CSS

**Interview-style answer**

> "The index page uses Bootstrap's grid and card system to display listings in a clean, responsive layout that works well across devices."

---

## 5️⃣ Styling New Listing Page (Form)

**Purpose**

> Allows users to add new travel listings.

**Styling approach**

- Bootstrap form controls
- Input validation feedback

- Proper spacing and labels

**Interview-style answer**

> “The new listing form is styled using Bootstrap form components to maintain usability and validation consistency.”

---

## 6️⃣ Styling Edit Page (Update Listing)

**Similar to New Page**

- Pre-filled data
- Same Bootstrap form structure
- Cleaner UX for updates

**Interview-style answer**

> “The edit page reuses the same form structure as the create page, with pre-filled data to improve user experience.”

---

## 7️⃣ Styling Show Page (Single Listing)

**What it includes**

- Large image
- Listing details
- Reviews section
- Edit/Delete buttons (owner only)

**Styling**

- Bootstrap grid layout
- Buttons styled using Bootstrap utilities
- Conditional rendering

**Interview-style answer**

> “The show page is designed to clearly display all listing details using Bootstrap layout utilities and conditionally rendered actions.”

---

## 8️⃣ Bootstrap (Overall Usage)

**Why Bootstrap?**

> Bootstrap helps create a **responsive, clean UI quickly** without writing excessive CSS.

**Where you used it**

- Navbar & footer
- Cards
- Forms
- Buttons
- Grid system

**Interview-style answer**

> “I used Bootstrap to ensure responsiveness and faster UI development while adding custom CSS for branding and layout control.”

## 1️⃣ Custom Error Handling (In Your Project)

**What it means (simple):**

Instead of letting the server **crash or send confusing errors**, you created **your own error system** so errors are:

- Clean
- Meaningful
- Easy to debug

**Example from WanderLust:**

If a user opens:

```
/listings/12345
```

but that listing doesn't exist.

Instead of:
❌ App crashes
❌ Random error message

You show:
✅ "Listing not found"

**Why interviewers like this:**

It shows you care about **user experience** and **backend stability**.

---

### 2️⃣ wrapAsync (warpasync)

**Problem it solves:**

In Express, **async errors are not caught automatically**.

Without `wrapAsync`:

```
app.get("/listings", async (req, res) => {
  const data = await Listing.find(); // error here crashes app
});
```

**What `wrapAsync` does:**

It **wraps async functions** and sends errors to the error handler.

**Simple meaning:**

> "wrapAsync catches errors inside async routes and forwards them safely."

**Interview line:**

> “I used wrapAsync to handle async route errors without writing try-catch everywhere.”

---

### 3️⃣ ExpressError (Custom Error Class)

**What it is:**

A **custom error object** that contains:

- Error message
- HTTP status code

**Why you created it:**

So you can throw errors like this:

```
throw new ExpressError(404, "Listing not found");
```

**Simple meaning:**

> “ExpressError helps me send proper status codes with meaningful messages.”

**Interview line:**

> “Instead of generic errors, I use ExpressError to control status codes like 404 or 400.”

---

### 4️⃣ error.ejs (Error Page)

**What it is:**

A **UI page** that shows errors nicely.

**Why it’s needed:**

Instead of showing:
❌ White screen
❌ JSON error dump

You show:
✅ Friendly message
✅ Same UI as rest of app

**Example:**

If someone visits an invalid URL:

`/listings/99999`

They see:

> ❝Sorry, this listing does not exist❞

**Simple meaning:**

> "error.ejs displays errors in a user-friendly way."

---

### 5️⃣ Validation Using Joi (Schema Validation)

**Problem it solves:**

Users can send **wrong or empty data** from forms or Postman.

**Example:**

User submits:

- Title = empty
- Price = "abc"

**What Joi does:**

Before saving to database, Joi **checks the data format**.

**Simple meaning:**

> "Joi validates user input before it reaches the database."

**Interview line:**

> "I used Joi to prevent invalid data from being stored."

---

## 6️⃣ Joi Validation as Middleware

**Why middleware?**

So validation happens:

- **Before** route logic
- Automatically
- For every request

**How it works (flow):**

```
Request → Joi Validation → Route Logic → Database
```

If data is invalid:
❌ Route is NOT executed
❌ DB is NOT touched

**Example:**

```
router.post(
  "/listings",
  validateListing,    // Joi middleware
  wrapAsync(createListing)
);
```

**Simple meaning:**

> "Middleware stops bad data early."

**Interview line:**

> "I used Joi validation as middleware to ensure clean and secure data flow."

## 1️⃣ Create Review Model

**What it means (simple):**

A **review model** is a blueprint for how a review is stored in the database.

**In your website:**

When a user writes:

- ⭐ Rating
- 📝 Comment

You need to store it properly.

**What your Review model contains:**

- Rating (number)
- Comment (text)
- Author (user who wrote it)
- Listing reference (which place it belongs to)

**Simple meaning:**

> "The review model defines what a review looks like in my database."

---

## 2️⃣ Create Reviews (Add Review)

**What happens when user submits a review:**

1. User opens a listing
2. Writes a comment and rating
3. Clicks **Submit Review**

**Backend flow:**

- You create a new review
- Attach it to that listing

- Save it in the database

**Simple meaning:**

> “Creating a review means saving the user’s feedback and linking it to the correct listing.”

**Interview line:**

> “Each listing can have multiple reviews, so I store reviews separately and link them using references.”

---

### 3️⃣ Validation for Reviews (Very Important)

**Why validation is needed:**

Users can send:

- Empty comment
- Rating = 10 (invalid)
- Fake request from Postman

**What you validate:**

- Rating must be between 1–5
- Comment should not be empty

**Where validation happens:**

Before saving review → **Joi middleware**

**Simple meaning:**

> “Review validation ensures only meaningful and valid feedback is stored.”

**Interview line:**

> “I use schema validation to prevent invalid reviews from entering the database.”

---

### 4️⃣ Render Reviews (Show Reviews on Website)

**What it means:**

Displaying reviews under each listing.

**How it works (simple):**

- When loading a listing
- Fetch its reviews from database
- Show:
    - Rating stars
    - Comment text
    - Reviewer name

**UI Example:**

⭐ ⭐ ⭐ ⭐
"Nice place, very clean!"
– Rahul

**Simple meaning:**

> "Rendering reviews means showing all reviews related to a listing on its detail page."

---

### 5️⃣ Delete Review

**Who can delete a review?**

- Only the user who created it
- Or admin (if any)

**What happens on delete:**

1. Review is removed from Review collection

2. Its reference is removed from Listing

**Why both are needed:**

If you delete only the review:
❌ Listing still points to it (bug)

**Simple meaning:**

> "Deleting a review means removing it safely from both the review collection and its parent listing."

**Interview line:**

> "I ensure database consistency while deleting reviews."

---

## 6️⃣ Handling Delete Listing (Very Important)

**Problem:**

A listing has:

- Images
- Reviews

If listing is deleted:
❌ Reviews remain in DB (orphan data)

**Your solution:**

When a listing is deleted:

- Automatically delete all related reviews

**How (conceptually):**

- Use middleware or logic to remove dependent reviews

**Simple meaning:**

> "When a listing is deleted, all its reviews are cleaned up automatically."

**Interview line:**

> “I handled cascading delete to avoid orphan records.”

## 1️⃣ Express Router

**What it is (simple):**

`express.Router()` helps you **organize routes** instead of putting everything in `app.js`.

**In your project:**

You have:

- Listings routes
- Reviews routes
- Users routes

Instead of writing all routes in one file, you separate them.

**Example (conceptual):**

```
/listings
/listings/:id
/listings/:id/reviews
/login
/register
```

**Simple meaning:**

> “Express Router helps me split routes into clean, manageable files.”

**Interview line:**

> “I use Express Router to keep my backend modular and readable.”

---

## 2️⃣ Web Cookies

**What cookies are:**

Cookies are **small data stored in the user's browser** by the server.

**In your project:**

Cookies are used to:

- Remember logged-in users
- Store session ID
- Show flash messages

**Simple example:**

When a user logs in:

- Server sends a cookie
- Browser stores it
- Browser sends it with every request

**Simple meaning:**

> "Cookies help the server remember the user."

---

### 3️⃣ Sending Cookies

**What it means:**

Server **creates and sends cookies** to the browser.

**In your project:**

After login:

- You send a cookie with user/session info

**Example (conceptual):**

```
res.cookie("user", "loggedIn");
```

**Simple meaning:**

“Sending cookies means saving small information in the user’s browser.”

**Interview line:**

“I send cookies to maintain user sessions.”

---

### 4️⃣ Cookie Parser

**Problem without cookie-parser:**

Express **cannot read cookies** sent by the browser.

**What cookie-parser does:**

- Reads cookies from requests
- Converts them into a usable object

**In your project:**

You access cookies like:

```
req.cookies
```

**Simple meaning:**

“cookie-parser lets Express understand cookies sent by the browser.”

**Interview line:**

“I use cookie-parser to read cookies safely in my backend.”

---

### 5️⃣ Signed Cookies (VERY IMPORTANT)

**Problem with normal cookies:**

Users can:

- Edit cookies manually
- Fake values

**Solution: Signed cookies**

Signed cookies:

- Are **tamper-proof**
- Use a secret key

**In your project:**

For:

- Session security
- Authentication

If user changes cookie:
❌ Server detects it
❌ Rejects it

**Simple meaning:**

> "Signed cookies ensure data integrity and prevent tampering."

**Interview line:**

> "I use signed cookies to secure session-related data."

1️⃣ **What is *State* (in your project)**

**State = user-related information that should be remembered between requests**

In **WanderLust**, examples of state:

- User is **logged in or not**
- User ID after login
- Success / error messages (flash)
- Which user created a listing or review

👉 HTTP is **stateless**, so Express uses **sessions** to maintain state.

---

## 2️⃣ What is `express-session`

**express-session** helps us:

- Store user data **on the server**
- Identify the user using a **session ID stored in a cookie**

📌 In WanderLust:

- After login → user stays logged in
- You don't need to login again on every page

---

## 3️⃣ How Session Works (Simple Flow)

1. User logs in
2. Server creates a **session**
3. Session ID is stored in **browser cookie**
4. On next request:
    - Browser sends session ID
    - Server finds session data
    - User is recognized

---

## 4️⃣ Session Setup in WanderLust Project

**app.js**

```
import session from "express-session";

const sessionOptions = {
  secret: "mysupersecretkey",
```

```
  resave: false,
  saveUninitialized: true,
  cookie: {
    httpOnly: true,
    expires: Date.now() + 7 * 24 * 60 * 60 * 1000,
    maxAge: 7 * 24 * 60 * 60 * 1000
  }
};

app.use(session(sessionOptions));
```

---

### 5️⃣ Cookie inside `sessionOptions` (Very Important)

**Why cookies?**

- Cookie stores **session ID**
- Session data stays on **server**

**Cookie properties explained:**

| Option | Meaning |
|---|---|
| `httpOnly` | JS can't access cookie (security) |
| `expires` | When cookie will expire |
| `maxAge` | Cookie life (7 days) |

📌 In WanderLust:

- User stays logged in for **7 days**

---

### 6️⃣ Storing & Using Session Information

**Store data in session**

```
req.session.user = user._id;
```

**Use session data**

```
if (req.session.user) {
   // user is logged in
}
```

📌 Used for:

- Login check
- Authorization
- Protected routes

---

## 7️⃣ What is `connect-flash`

**connect-flash** is used to show:

- Success messages
- Error messages

📌 Example:

- “Logged in successfully”
- “You must be logged in”
- “Listing created successfully”

Flash messages use **session internally**.

---

## 8️⃣ Flash Setup in WanderLust

```
import flash from "connect-flash";

app.use(flash());
```

---

## 9️⃣ `res.locals` (VERY IMPORTANT)

`res.locals` makes data available to **all EJS files**

### Middleware

```
app.use((req, res, next) => {
  res.locals.success = req.flash("success");
  res.locals.error = req.flash("error");
  next();
});
```

📌 Now `success` and `error` are available in **every EJS file**

---

## 🔟 Implement Flash in Project (Backend)

### Success Flash

```
req.flash("success", "Listing created successfully!");
res.redirect("/listings");
```

### Error Flash

```
req.flash("error", "You must be logged in!");
res.redirect("/login");
```

---

## 1️⃣1️⃣ Flash Success Include (EJS)

**`views/includes/success.ejs`**

```
<% if (success.length) { %>
  <div class="alert alert-success">
    <%= success %>
  </div>
<% } %>
```

---

### 1️⃣2️⃣ Flash Failure Include (EJS)

**views/includes/error.ejs**

```
<% if (error.length) { %>
  <div class="alert alert-danger">
    <%= error %>
  </div>
<% } %>
```

---

### 1️⃣3️⃣ Use Flash Includes in Layout

```
<%- include("../includes/success") %>
<%- include("../includes/error") %>
```

📌 Now messages appear automatically on all pages

### 1️⃣ Authentication vs Authorization (Most Important)

#### 🔐 Authentication – *Who are you?*

- Checks **username & password**
- Happens during **login**

📌 In WanderLust:

- User logs in using email & password
- Passport verifies credentials

Example:

> “Is this user really Samarth?”

---

#### 🚫 Authorization – *What can you do?*

- Checks **permissions**

- Happens **after login**

📌 In WanderLust:

- Only logged-in user can:
    - Create listing
    - Add review
- Only listing owner can:
    - Edit / delete listing

Example:

> “Can Samarth delete this listing?”

---

### 2️⃣ How Passwords Are Stored (Very Important)

❌ **Wrong way (Never used):**

```
password: "samarth123"
```

✅ **Correct way (Used in project):**

- Password is **hashed**
- Original password is **never stored**

📌 Database stores something like:

```
$2a$10$KXj8k1s...
```

---

### 3️⃣ Hashing (Simple)

**Hashing = converting password into unreadable string**

Example:

```
Password: samarth123
```

```
Hashed:   $2a$10$AbCDeFg...
```

- One-way process
- Cannot get original password back

📌 In WanderLust:

- Hashing is done automatically by **passport-local-mongoose**

---

### 4️⃣ Salting (Extra Security)

**Salting = adding random value before hashing**

Why?

- Same password → different hash
- Protects against rainbow table attacks

📌 You didn't manually add salt
👉 `passport-local-mongoose` handles it internally

---

### 5️⃣ What is Passport (In Simple Words)

**Passport = authentication library for Express**

Passport helps with:

- Login
- Signup
- Session handling
- Password hashing & verification

📌 In WanderLust:

- Passport checks credentials
- Passport stores logged-in user in session

---

### 6️⃣ User Model (Your Project)

**models/user.js**

```
import mongoose from "mongoose";
import passportLocalMongoose from "passport-local-mongoose";

const userSchema = new mongoose.Schema({
  email: String
});

userSchema.plugin(passportLocalMongoose);

export default mongoose.model("User", userSchema);
```

**What this plugin does automatically:**

- Adds username & password fields
- Hashes password
- Adds `authenticate()`, `register()`

---

### 7️⃣ Configuring Passport Strategy

**app.js**

```
import passport from "passport";
import LocalStrategy from "passport-local";
import User from "./models/user.js";

app.use(passport.initialize());
app.use(passport.session());

passport.use(new LocalStrategy(User.authenticate()));
```

```
passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

📌 Meaning:

- Passport knows **how to login**
- Passport knows **how to store user in session**

---

## 8️⃣ Signup (GET & POST)

### 🔹 Signup GET

```
app.get("/signup", (req, res) => {
  res.render("users/signup.ejs");
});
```

➡️ Shows signup form

---

### 🔹 Signup POST

```
app.post("/signup", async (req, res) => {
  try {
    let { username, email, password } = req.body;
    const newUser = new User({ username, email });
    const registeredUser = await User.register(newUser, password);
    req.login(registeredUser, err => {
      if (err) return next(err);
      req.flash("success", "Welcome to WanderLust!");
      res.redirect("/listings");
    });
  } catch (e) {
    req.flash("error", e.message);
    res.redirect("/signup");
  }
});
```

📌 What happens:

1. User submits form
2. Password is hashed
3. User saved in DB
4. User logged in automatically

---

### 9️⃣ Login (GET & POST)

🔹 **Login GET**

```
app.get("/login", (req, res) => {
  res.render("users/login.ejs");
});
```

---

🔹 **Login POST**

```
app.post("/login",
  passport.authenticate("local", {
    failureRedirect: "/login",
    failureFlash: true
  }),
  (req, res) => {
    req.flash("success", "Welcome back!");
    res.redirect("/listings");
  }
);
```

📌 What happens:

- Passport checks username & password
- If correct → user logged in
- If wrong → error flash shown

---

### 🔁 How Everything Works Together (Flow)

1. User signs up → password hashed
2. User logs in → passport authenticates
3. Session created → stored using cookie
4. Authorization middleware checks permissions
5. Flash messages show success/error

---

### 🧠 Interview-Friendly Summary

> “In my WanderLust project, authentication is handled using Passport.js with a local strategy. Passwords are securely hashed and salted using passport-local-mongoose. Sessions maintain login state, and authorization ensures only authenticated users can create or modify listings.”

### 1️⃣ Connecting Login Route (How login works in WanderLust)

**Login GET (Show form)**

```
router.get("/login", (req, res) => {
  res.render("users/login.ejs");
});
```

➡️ Shows login page

---

**Login POST (Authenticate user)**

```
router.post(
  "/login",
  passport.authenticate("local", {
    failureRedirect: "/login",
    failureFlash: true
  }),
  (req, res) => {
    req.flash("success", "Welcome back!");
    res.redirect("/listings");
  }
);
```

📌 What happens:

- Passport checks username & password
- If correct → user logged in
- Session is created
- Redirect to listings page

---

### 2️⃣ Logout User

**Logout Route**

```
router.get("/logout", (req, res, next) => {
  req.logout(err => {
    if (err) return next(err);
    req.flash("success", "Logged out successfully!");
    res.redirect("/listings");
  });
});
```

📌 Meaning:

- Session destroyed
- User removed from req.user
- User is logged out

---

### 3️⃣ Login After Signup (Auto Login)

After signup, user should not login again.

```
req.login(registeredUser, err => {
  if (err) return next(err);
  req.flash("success", "Welcome to WanderLust!");
  res.redirect("/listings");
});
```

📌 Result:

- User signs up
- Automatically logged in
- Better user experience

---

### 4️⃣ Post Login Page (Redirect After Login)

**Problem:**

User tries to:

```
/listings/new
```

But is not logged in → redirected to login

**Solution:**

Save original URL in session

```
req.session.redirectUrl = req.originalUrl;
```

After login:

```
const redirectUrl = req.session.redirectUrl || "/listings";
delete req.session.redirectUrl;
res.redirect(redirectUrl);
```

📌 User goes back to the page they wanted

---

### 5️⃣ Listing Owner (Very Important)

Each listing has an **owner**

**Listing Model**

```
owner: {
  type: mongoose.Schema.Types.ObjectId,
  ref: "User"
```

```
}
```

📌 Owner = user who created the listing

---

### 6️⃣ Authorization for Listings

**Middleware: `isOwner`**

```
const isOwner = async (req, res, next) => {
  const { id } = req.params;
  const listing = await Listing.findById(id);

  if (!listing.owner.equals(req.user._id)) {
    req.flash("error", "You do not have permission");
    return res.redirect(`/listings/${id}`);
  }
  next();
};
```

📌 Used for:

- Edit listing
- Delete listing

---

**Using Authorization**

```
router.put("/:id", isLoggedIn, isOwner, updateListing);
router.delete("/:id", isLoggedIn, isOwner, deleteListing);
```

---

### 7️⃣ Authorization for Reviews

**Review Owner**

```
author: {
  type: mongoose.Schema.Types.ObjectId,
  ref: "User"
}
```

---

**Review Authorization Middleware**

```
const isReviewAuthor = async (req, res, next) => {
  const { reviewId, id } = req.params;
  const review = await Review.findById(reviewId);

  if (!review.author.equals(req.user._id)) {
    req.flash("error", "You are not allowed to do that");
    return res.redirect(`/listings/${id}`);
  }
  next();
};
```

---

**Using It**

```
router.delete(
  "/:id/reviews/:reviewId",
  isLoggedIn,
  isReviewAuthor,
  deleteReview
);
```

## 1️⃣ MVC Architecture (Most Important)

**MVC = Model – View – Controller**

🔹 **Model (Database Layer)**

- Defines **how data looks**
- Talks to MongoDB

📌 In WanderLust:

- `Listing.js`
- `User.js`

- `Review.js`

Example:

`Listing.find()`

---

🔹 **View (Frontend / UI)**

- What user **sees**
- Written in **EJS**

📌 In WanderLust:

- `listings/index.ejs`
- `listings/show.ejs`
- `users/login.ejs`

---

🔹 **Controller (Logic)**

- Handles **request & response**
- Connects Model and View

📌 In WanderLust:

- `controllers/listings.js`
- `controllers/users.js`
- `controllers/reviews.js`

👉 **Why MVC?**

- Clean code

- Easy to manage
- Industry standard

---

### 2️⃣ `router.route()` (Cleaner Routing)

Instead of writing routes separately:

❌ Old way

```
router.get("/:id", showListing);
router.put("/:id", updateListing);
router.delete("/:id", deleteListing);
```

✅ Used in WanderLust

```
router.route("/:id")
  .get(showListing)
  .put(isLoggedIn, isOwner, updateListing)
  .delete(isLoggedIn, isOwner, deleteListing);
```

📌 Benefits:

- Cleaner code
- All actions of same route in one place

---

### 3️⃣ Image Upload (Cloudinary)

**Why Cloudinary?**

- Images should **not** be stored on server
- Cloudinary stores images online
- Fast & secure

---

**Flow in WanderLust:**

1. User uploads image
2. Image goes to **Cloudinary**
3. Cloudinary returns:
    - `url`
    - `filename`
4. Save these in MongoDB

---

**Image Schema**

```
image: {
  url: String,
  filename: String
}
```

---

**Upload Middleware**

```
import multer from "multer";
import { storage } from "../cloudConfig.js";

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

Used as:

```
router.post("/", upload.single("image"), createListing);
```

---

### 4️⃣ Mapbox (Map Feature)

**Why Mapbox?**

- Shows listing location on map
- Improves user experience

📌 In WanderLust:

- Every listing has a location
- Map shows marker for that location

---

**Location Stored in DB**

```
geometry: {
  type: {
    type: String,
    enum: ["Point"],
    required: true
  },
  coordinates: [Number]
}
```

---

### 5️⃣ Marker & Popup (Mapbox)

**Marker**

- Shows **pin on map**
- Based on coordinates

**Popup**

- Shows:
    - Listing title
    - Location
- Appears when clicking marker

📌 Example:

```
new mapboxgl.Marker()
  .setLngLat(coordinates)
  .setPopup(
    new mapboxgl.Popup().setHTML("<h4>Listing Title</h4>")
```

```
  )
  .addTo(map);
```

---

### 6️⃣ MongoDB Atlas

**What is MongoDB Atlas?**

- Cloud version of MongoDB
- No local database needed
- Accessible from anywhere

📌 In WanderLust:

- Stores:
  - Users
  - Listings
  - Reviews
- Connected using connection string

```
mongoose.connect(process.env.MONGODB_URL);
```

---

### 7️⃣ Render (Deployment)

**What is Render?**

- Hosting platform
- Used to deploy:
  - Backend
  - Frontend
  - Database connection

📌 In WanderLust:

- App deployed on **Render**
- MongoDB Atlas used as database
- Cloudinary used for images

👉 Final app is **live & accessible**