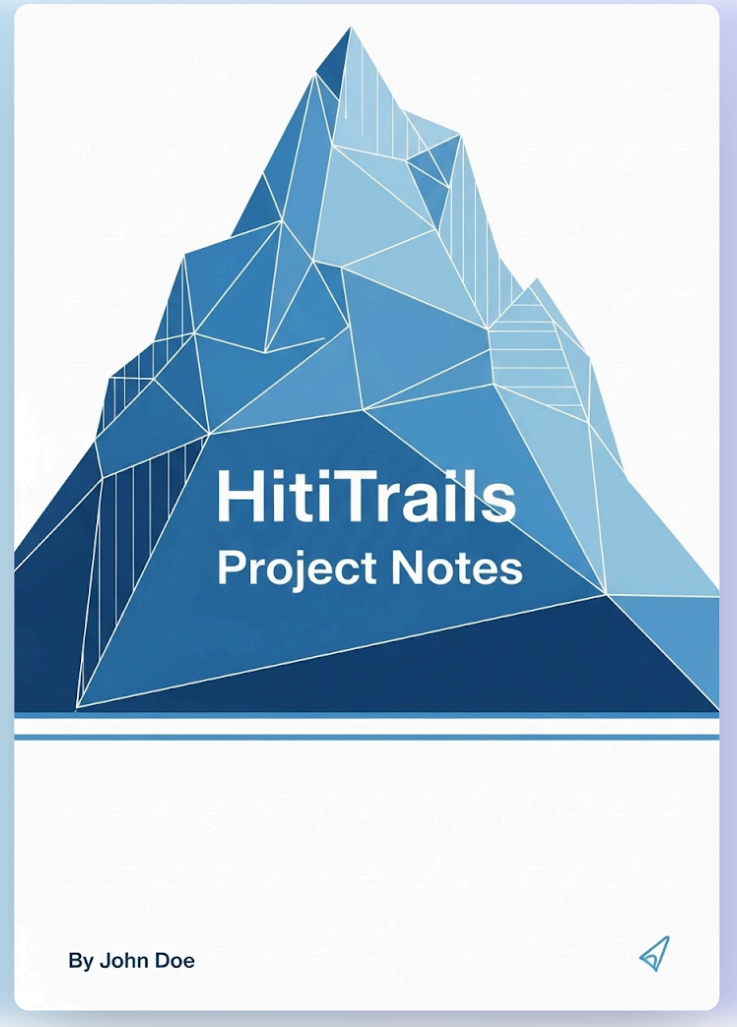
HitiTrails Docs

FULL STACK REVISION

HitiTrails
Project Guide

A comprehensive chapter-wise revision note blending theoretical concepts with practical code implementation for your Node.js & Express application.

`Node.js` `Express` `MongoDB` `Passport`



CHAPTER 01

Basic Setup & Architecture

I→ The Entry Point (app.js)

This is the heart of your application. It initializes the server, connects to the database, and sets up global middleware (like sessions and parsing).

app.js DB Connection

```
async function main() {
  await mongoose.connect(DBURL);
}
main().then(() => console.log("DB connected"));
```

MVC Architecture (Model - View - Controller)

Separation of concerns is key. You moved logic out of the main file into specialized directories.

Models

Data Structure

`models/listing.js`

Views

User Interface

`views/listings/`

Controllers

Business Logic

`controller/listings.js`

CHAPTER 02

Database Modeling

Schemas & Models

Defined in `models/listing.js`. Defines the "shape" of your data.

```
const listingSchema = new Schema({
    title: { type: String, required: true },
    price: Number,
    image: { url: String, filename: String },
    // ...
});
```

1:N Relationship Strategy

A Listing has **Many** Reviews. We store references (ObjectIds) in the Listing array to keep the document size efficient.

```
reviews: [
  {
    type: Schema.Types.ObjectId,
    ref: "Review",
  },
],
```

CHAPTER 03

CRUD Operations & Routing

Route Name	HTTP Verb	Path	Description
Index	GET	`/listings`	Show all listings
New	GET	`/listings/new`	Show creation form
Create	POST	`/listings`	Save to DB
Show	GET	`/listings/:id`	Show detailed view
Edit	GET	`/listings/:id/edit`	Show edit form
Update	PUT	`/listings/:id`	Save changes
Delete	DELETE	`/listings/:id`	Remove listing

> **Key Concept: method-override**
>
> HTML forms only support GET and POST. To use PUT and DELETE, you installed `method-override`.
> Usage: `app.use(methodOverride("_method"));` allows URLs like `/listings/:id?_method=DELETE`.

CHAPTER 04

Error Handling & Middlewares

Custom ExpressError

Extends the standard JS Error class to include a Status Code (e.g., 404, 500).

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

wrapAsync Utility

Replaces try-catch blocks in async routes. Passes errors to `next()`.

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

Schema Validation with Joi

You defined `validateListing` middleware. It validates incoming data against a Joi schema *before* it hits your database logic. If data is invalid (e.g. negative price), it throws an error immediately, saving DB resources.

CHAPTER 05

Authentication & Authorization

Authentication (Who are you?)

- **Passport.js:** Handling the complexity of login sessions.
- **Local Strategy:** Used `passport-local-mongoose` in `models/user.js` to add username/salt/hash fields automatically.
- **Session:** `passport.session()` maintains the user's login state across pages.

Authorization (What can you do?)

isLoggedIn
Checks `req.isAuthenticated()`. If false, redirects to login and saves `redirectUrl`.

isOwner
Checks if `currUser._id` matches `listing.owner`. Protects Edit/Delete routes.

CHAPTER 06

Advanced Features: Images & Maps

Image Uploads

Stored in Cloudinary, not your server DB.

1. **Multer:** Parses `multipart/form-data` from the form.
2. **Cloud Config:** `cloudConfig.js` connects to your account.
3. **Storage:** DB only saves the URL and Filename.

```
upload.single("image")
```

Geocoding & Maps

Used MapTiler SDK for both geocoding and rendering.

```
const geoResults = await
maptilerClient.geocoding.forward(newListing.location);
```

This converts "Jaipur" into `{ type: 'Point', coordinates: [75.78, 26.91] }`.

CHAPTER 07

UI/UX Improvements

Flash Messages

Temporary pop-up messages (e.g., "Listing Created!") using `connect-flash`.

```
res.locals.success = req.flash("success");
```

Star Rating System

Implemented using `starability-basic` CSS library. It ingeniously uses standard HTML Radio Buttons (for form submission validity) styled as Stars.

CHAPTER 08

Booking System

EXTRA FEATURE

Booking Logic Implementation

Calculating costs on the server side.

Since this isn't in standard tutorials, you created a specific `Booking` model linking a User (Booker) and a Listing.

```
// controller/booking.js
const nights = Math.ceil((checkOutDate -
checkInDate) / (1000 * 60 * 60 * 24));
const totalPrice = nights * listing.price * guests;
```