CampusCare Backend – Step-by-Step TODO List

Phase 1: Setup & Configuration

- Install dependencies (mysql2, bcryptjs, jsonwebtoken, express-validator, helmet, morgan)
- Create .env file with database credentials and JWT_SECRET

Phase 2: Models (Database Layer)

- Create User model (models/User.js) — CRUD for user_profiles
- Create Blog model (models/Blog.js) — CRUD for blog
- Create Comment model (models/Comment.js) — operations for blog_comments
- Create Reaction model (models/Reaction.js) — operations for blog_likes
- Create Resource model (models/Resource.js) — operations for academic_resources
- Create LocalGuide model (models/LocalGuide.js) — operations for places and place_rating

Phase 3: Middleware

- Authentication middleware (middleware/auth.js) — JWT verification
- Authorization middleware (middleware/authorize.js) — role based access
- Validation middleware (middleware/validate.js) — input validation
- Error handler middleware (middleware/errorHandler.js) — global error handling

Phase 4: Controllers (Business Logic)

- Auth controller — register, login, logout
- Blog controller — CRUD blogs
- Comment controller — create/read comments
- Reaction controller — like/unlike blogs
- Resource controller — student-only resources
- Local guide controller — places by category with ratings

Phase 5: Routes (API Endpoints)

- Auth routes — /api/auth/register, /api/auth/login
- Blog routes — GET, POST, PUT, DELETE /api/blogs
- Comment routes — /api/blogs/:id/comments
- Reaction routes — /api/blogs/:id/like

- Resource routes — /api/resources (student only)
- Local guide routes — /api/local-guide, /api/local-guide/:category

Phase 6: Integration & Testing

- Integrate routes, helmet, morgan, error handler in app.js
- Test APIs using Postman and fix issues

Now starting the project.

Phase 1: Setup & Configuration — Explanation

What We Did

---

1. Updated `package.json` with Required Dependencies

Added the core packages needed to build and secure the backend:

- `mysql2` — MySQL driver for Node.js
- `bcryptjs` — Used for hashing user passwords securely
- `jsonwebtoken` — Handles JWT-based authentication
- `express-validator` — Validates and sanitizes user input
- `helmet` — Adds security-related HTTP headers
- `morgan` — Logs HTTP requests for debugging and monitoring

---

2. Created `config/database.js`

- Sets up a **MySQL connection pool**
- Reads database credentials from **environment variables**
- Provides a **reusable database connection** across the application

This avoids opening a new database connection for every request.

---

3. Created `config/constants.js`

Centralizes application-wide constants such as:

- User roles: `student`, `guest`, `moderator`, `admin`
- JWT expiration time
- Pagination defaults (page size, limits)
- Local guide categories

This makes the app easier to maintain and update.

---

4. Created `env.template`

- Acts as a reference template for required environment variables
- Helps ensure `.env` contains all necessary values
- Prevents runtime errors caused by missing configuration

---

5. Created `.gitignore`

- Ensures sensitive files like `.env` are **not committed to Git**
- Protects database credentials and secret keys
- Follows security best practices

Why This Matters

- **Dependencies:** Installed all essential tools required to build the backend
- **Database Connection:** Backend is ready to connect to MySQL reliably
- **Configuration Management:** Centralized settings improve readability and maintenance
- **Security:** `.gitignore` prevents accidental exposure of secrets
-

Helmet and Morgan (NPM Packages) — Explanation

Helmet

What It Is

Helmet is a **security middleware for Express.js**.

What It Does

Helmet helps secure your backend by **setting HTTP security headers** that protect your application from common web vulnerabilities.

Examples of Protection

- Prevents **XSS (Cross-Site Scripting)** attacks
- Prevents **clickjacking**
- Hides the fact that your app is using Express
- Helps enforce **HTTPS** in production
- Sets **Content Security Policies (CSP)**

How It Works

Helmet is added as middleware in your Express app.
Once added, it **automatically applies security headers to all HTTP responses**.

```
app.use(helmet());
```

That’s it — all responses now include security headers.

Morgan

What It Is

Morgan is an **HTTP request logger middleware** for Express.js.

What It Does

Morgan logs every HTTP request made to your server.
This is extremely useful for **debugging**, **monitoring**, and **development**.

What It Logs

- Request method (GET, POST, PUT, DELETE)
- Request URL
- Response status code
- Response time
- Client IP address

Example Console Output

```
GET /api/blogs 200 45ms

POST /api/auth/login 200 120ms

GET /api/resources 401 5ms
```

How It Works

Morgan automatically logs **every incoming request** once added as middleware.

```
app.use(morgan('dev'));
```

The `'dev'` format shows logs in a clean, readable format.

Why We Use Them

- **Helmet:** Adds essential security headers with minimal setup
- **Morgan:** Provides real-time request logging for debugging and monitoring

Phase 2: Models (Database Layer) — Explanation

What Are Models?

Models are **JavaScript modules/classes** that handle all database-related operations. They:

- Encapsulate database queries
- Provide reusable methods for **CRUD operations**
- Keep database logic separate from business logic
- Make the code easier to maintain and test

In simple terms, models act as a **bridge between controllers and the database**.

Why We Need Models

Instead of writing raw SQL queries inside controllers, we:

- Write SQL logic **once** in models
- Reuse model methods across multiple controllers
- Keep controllers clean and readable
- Improve maintainability and scalability
- Make unit testing easier

This follows the **separation of concerns** principle.

What We Created

---

1. User Model (`models/User.js`)

Purpose:
Manage user accounts and profiles.

Key Methods:

- `findByEmail(email)` — Find user by email (used during login)
- `findById(userId)` — Fetch user details without exposing password
- `create(userData)` — Register a new user
- `update(userId, updateData)` — Update user profile
- `findByIdWithDetails(userId)` — Fetch user with college/course info
- `findCollegeByEmailDomain(email)` — Validate college email domain

Example Usage:

```
// In a controller

const user = await User.findByEmail('student@college.edu');


if (user) {

  // User exists

}
```

---

2. Blog Model (`models/Blog.js`)

Purpose:
Manage blog posts.

Key Methods:

- `findAll(collegeId, page, limit)` — Fetch all blogs with pagination
- `findById(blogId)` — Fetch single blog with author info and like count
- `create(blogData)` — Create a new blog
- `update(blogId, userId, updateData)` — Update blog (only by owner)
- `delete(blogId, userId)` — Delete blog (only by owner)
- `isOwner(blogId, userId)` — Verify blog ownership

Special Features:

- Includes **like count** and **comment count**
- Includes **author information**
- Supports **pagination**

---

3. Comment Model (`models/Comment.js`)

Purpose:
Manage blog comments (single-level replies).

Key Methods:

- `findByBlogId(blogId)` — Fetch all comments for a blog
- `create(commentData)` — Add a new comment
- `delete(commentId, userId)` — Delete comment (only by owner)

- `isOwner(commentId, userId)` — Verify ownership

Note:
Comments are **single-level only** (no nested replies), as per project requirements.

---

4. Reaction Model (`models/Reaction.js`)

Purpose:
Manage blog likes (no dislikes).

Key Methods:

- `hasLiked(blogId, userId)` — Check if user has liked the blog
- `like(blogId, userId)` — Add a like
- `unlike(blogId, userId)` — Remove a like
- `getLikeCount(blogId)` — Get total likes
- `getLikedBy(blogId)` — Get list of users who liked the blog

Note:
Only **likes** are supported; dislikes are intentionally excluded.

---

5. Resource Model (`models/Resource.js`)

Purpose:
Manage academic resources (student-only access).

Key Methods:

- `findByCollegeId(collegeId, page, limit)` — Fetch resources for a college
- `findById(resourceId)` — Fetch a single resource

- `create(resourceData)` — Create new resource (admin/moderator)
- `update(resourceId, updateData)` — Update resource
- `delete(resourceId)` — Delete resource

Note:
Resources are **college-specific** and accessible only to **students**.

---

6. LocalGuide Model (`models/LocalGuide.js`)

Purpose:
Manage local places and user ratings.

Key Methods:

- `findByCollegeId(collegeId, categoryId)` — Fetch places for a college
- `findByCategory(collegeId, categoryName)` — Filter places by category
- `getCategories()` — Fetch all categories
- `addRating(placeId, userId, rating)` — Add/update rating (1–5 stars)
- `getUserRating(placeId, userId)` — Fetch user's rating
- `create(placeData)` — Add new place (admin only)

Special Features:

- Calculates **average ratings**
- Groups places by category
- Supports categories:
    - healthcare

    - tech
    - clothing
    - food
    - logistics

How Models Work Together

```
Controller → Model → Database

     ↓         ↓         ↓

 Business     SQL      MySQL

  Logic     Queries    Tables
```

Example Flow

1. Controller receives request: *"Get all blogs"*
2. Controller calls: `Blog.findAll(collegeId, page, limit)`

Model executes SQL query:

```
SELECT * FROM blog WHERE college_id = ?
```

3. Model returns data to controller
4. Controller sends response to client

Security Features in Models

Parameterized Queries (SQL Injection Protection)

```
// ✅ Safe
pool.execute(
  'SELECT * FROM users WHERE email = ?',
  [email]
);

// ❌ Dangerous (SQL injection risk)
pool.execute(
  `SELECT * FROM users WHERE email = '${email}'`
);
```

- Prevents SQL injection attacks

Other Security Measures

- Input validation (expected data types only)
- Ownership checks using `isOwner()` methods
- Password field never exposed in fetch methods

Benefits of Using Models

- **Reusability:** Same methods used across controllers
- **Maintainability:** SQL changes made in one place
- **Testability:** Models can be tested independently
- **Organization:** Clear separation of concerns
- **Performance:** Optimized queries using joins and aggregations

Phase 3: Middleware — Detailed Explanation

📌 What is Middleware?

Middleware are functions that execute **between the incoming request and the outgoing response** in an Express.js application.

They can:

- Execute code
- Modify the request (`req`) or response (`res`)
- End the request–response cycle
- Pass control to the next middleware using `next()`

🔁 Request Flow

Request → Middleware 1 → Middleware 2 → Controller → Response

Think of middleware as **checkpoints** that every request must pass through.

❓ Why Do We Need Middleware?

🔐 Security

- Protect routes
- Verify JWT tokens
- Enforce permissions

✅ Validation

- Ensure request data is correct
- Block malformed or malicious input

⚠️ Error Handling

- Catch errors centrally
- Return consistent error responses

♻️ Code Reuse

- Write logic once
- Use it across multiple routes

🧩 What We Created

1️⃣ Authentication Middleware (`middleware/auth.js`)

🎯 Purpose

To **verify JWT tokens** and **authenticate users**.

⚙️ How It Works

1. Checks for `Authorization: Bearer <token>` header
2. Extracts the token
3. Verifies token using `JWT_SECRET`
4. Fetches the user from the database
5. Attaches user data to `req.user`
6. Calls `next()` or returns an error

🔹 Functions

a) `authenticate` — Required Authentication

- ❌ No token → `401 Unauthorized`
- ❌ Invalid token → `401 Unauthorized`
- ✅ Valid token → Adds user to `req.user` and continues

b) `optionalAuth` — Optional Authentication

- ❌ No token → Continues without error
- ✅ Valid token → Adds user to `req.user`
- Used for routes that work **with or without login**

📍 Example Usage

// Protected route (login required)

router.get('/profile', authenticate, getProfile);

// Public route (login optional)

router.get('/blogs', optionalAuth, getBlogs);

👤 Data Attached to `req.user`

{

"userId": 123,

"email": "student@college.edu",

"collegeId": 5,

"isModerator": false,

  "isAdmin": false

}

2️⃣ Authorization Middleware (`middleware/authorize.js`)

🎯 Purpose

To **control access based on user roles** (Role-Based Access Control).

🔹 Functions

a) `requireStudent`

- Ensures the user has a `collegeId`
- Used for **student-only features** (resources, creating blogs)

b) `requireModerator`

- Allows **moderators or admins**
- Used for moderation tasks

c) `requireAdmin`

- Allows **only admins**
- Used for admin-only operations

d) `requireModeratorOrAdmin`

- Allows **moderators and admins**

📍 Example Usage

// Only students can access resources

```
router.get('/resources',
  authenticate,
  requireStudent,
  getResources
);

// Only admins can delete users
router.delete('/users/:id',
  authenticate,
  requireAdmin,
  deleteUser
);
```

🚫 Error Responses

- ❌ Not logged in → `401 Unauthorized`
- ❌ Wrong role → `403 Forbidden`

3️⃣ Validation Middleware (`middleware/validate.js`)

🎯 Purpose

To **handle validation errors** from `express-validator`.

⚙️ How It Works

1. Reads validation results
2. If errors exist → returns formatted error response
3. If no errors → passes control to controller

📍 Example Usage

```
const { body } = require('express-validator');

const { handleValidationErrors } = require('../middleware/validate');



router.post('/register',

  [

    body('email').isEmail().withMessage('Invalid email'),

    body('password').isLength({ min: 6 })

      .withMessage('Password must be at least 6 characters')

  ],

  handleValidationErrors,

  registerController

);
```

❌ Validation Failure Response

```
{

  "success": false,

  "message": "Validation failed",

  "errors": [
  {
   "field": "email",
   "message": "Invalid email",
   "value": "invalid-email"
  },
  {
   "field": "password",
   "message": "Password must be at least 6 characters",
   "value": "123"
  }
 ]
}
```

4️⃣ Error Handler Middleware (`middleware/errorHandler.js`)

🎯 Purpose

To provide **centralized, consistent error handling**.

🔹 Functions

a) `errorHandler`

- Catches all application errors
- Handles known error types
- Logs errors for debugging
- Returns consistent error responses

b) `notFound`

- Handles undefined routes
- Returns `404 Route not found`

🧠 Error Types Handled

- MySQL duplicate entry → `409 Conflict`
- Foreign key errors → `400 Bad Request`
- Database connection errors → `500 Internal Server Error`
- JWT errors → `401 Unauthorized`
- Validation errors → `400 Bad Request`
- Invalid ID format → `400 Bad Request`

📍 Setup in `app.js`

```
app.use('/api', routes);   // All routes

app.use(notFound);         // 404 handler

app.use(errorHandler);     // Global error handler (must be last)
```

❌ Error Response Format

```
{

  "success": false,

  "message": "Error message here",

  "stack": "..."  // Only in development

}
```

🔄 How Middleware Works Together (Example: Create Blog)

1. **Request**: `POST /api/blogs`
    - Header: `Authorization: Bearer <token>`
    - Body: `{ title, content }`
2. **authenticate**
    - ✅ Token valid
    - ✅ User found
    - → Adds `req.user`
3. **requireStudent**
    - ✅ User is a student
4. **handleValidationErrors**
    - ✅ Title and content valid
5. **Controller**
    - Blog created successfully
6. **If error occurs**
    - Caught by `errorHandler`

🔐 Security Features

- **Token Verification**
    - Validates signature and expiration
- **Role-Based Access Control**
    - Students → create content
    - Guests → read-only
    - Admins → full control
- **Input Validation**
    - Blocks invalid and malicious data
- **Centralized Error Handling**
    - No sensitive data leaks
    - Proper HTTP status codes

✅ Benefits

- 🔐 **Security** — Protected routes and access control
- ♻️ **Code Reuse** — Shared logic across routes

- 📏 **Consistency** — Same error and validation format everywhere
- 🛠️ **Maintainability** — Update logic in one place
- 🐞 **Debugging** — Centralized error logging

🧠 Real-World Analogy

Think of middleware as **security checkpoints at an airport**:

- 🪪 **Authentication** → ID check (Who are you?)
- 🎟️ **Authorization** → Permission check (What can you do?)
- 🎒 **Validation** → Bag check (Is your data safe?)
- 🚨 **Error Handler** → Emergency response (What if something goes wrong?)

Phase 4: Controllers (Business Logic Layer) — Detailed Explanation

---

📌 What Are Controllers?

Controllers handle the **core business logic** of your application. They act as the bridge between **routes** and **models**.

Controllers:

- Receive requests from routes
- Use models to interact with the database
- Process business rules and logic
- Return responses to clients

🧠 How Controllers Fit in the Flow

Route → Controller → Model → Database

```
        ↓
    Process Logic
```

↓

Send Response

Think of controllers as the **brain of your API**.

❓ Why Do We Need Controllers?

🧩 Separation of Concerns

- Routes handle **URL mapping**
- Controllers handle **logic**

♻️ Reusability

- Same controller can be used by multiple routes

🛠️ Maintainability

- Business logic is centralized in one place

🧪 Testability

- Controllers can be tested independently from routes

🧩 What We Created

1️⃣ Auth Controller (`controllers/authController.js`)

🎯 Purpose

Handle **user authentication and profile management**.

🔹 Functions

a) `register` — Register New User

Input

{ "email", "password", "reg_no", "first_name", ... }

Process

1. Check if email already exists
2. Validate college email domain
3. Hash password using `bcrypt`
4. Create user in database
5. Generate JWT token

Output

{ "success": true, "token": "...", "user": { ... } }

---

b) `login` — Login User

Input

{ "email", "password" }

Process

1. Find user by email
2. Compare password with hashed password
3. Generate JWT token
4. Fetch user details

Output

{ "success": true, "token": "...", "user": { ... } }

---

c) `getProfile` — Get Current User Profile

Input

- `req.user` (from authentication middleware)

Process

1. Fetch user details from database
2. Include college and course information

Output

{ "success": true, "user": { ... } }

2️⃣ Blog Controller (`controllers/blogController.js`)

🎯 Purpose

Manage **blog posts** (CRUD operations).

🔹 Functions

a) `getAllBlogs` — Get All Blogs

Input

- Query params: `page`, `limit`, `collegeId`

Process

1. Fetch blogs with pagination
2. Include like and comment counts

3. Include author information

Output

{ "success": true, "blogs": [...], "pagination": { ... } }

---

b) `getBlogById` — Get Single Blog

Input

- Blog ID from URL params

Process

1. Fetch blog details
2. Fetch associated images
3. Include author and statistics

Output

{ "success": true, "blog": { ... } }

---

c) `createBlog` — Create New Blog

Input

- `{ blog_title, blog_content }`
- `req.user`

Process

1. Validate input
2. Create blog with user's `collegeId`
3. Save blog to database

Output

{ "success": true, "blog": { ... } }

---

d) `updateBlog` — Update Blog

Input

- Blog ID
- Updated content
- `req.user`

Process

1. Verify blog exists
2. Verify ownership
3. Update blog

Output

{ "success": true, "blog": { ... } }

---

e) `deleteBlog` — Delete Blog

Input

- Blog ID
- `req.user`

Process

1. Verify blog exists
2. Verify ownership
3. Delete blog (cascades to likes/comments)

Output

{ "success": true }

3️⃣ Comment Controller (`controllers/commentController.js`)

🎯 Purpose

Manage **blog comments** (single-level replies).

🔹 Functions

a) `getCommentsByBlogId`

Process

1. Verify blog exists
2. Fetch all comments
3. Include author details

Output

{ "success": true, "comments": [...], "count": n }

---

b) `createComment`

Process

1. Validate content
2. Verify blog exists
3. Create comment

Output

{ "success": true, "comment": { ... } }

---

c) `deleteComment`

Process

1. Verify comment exists
2. Verify ownership
3. Delete comment

Output

{ "success": true }

4️⃣ Reaction Controller (`controllers/reactionController.js`)

🎯 Purpose

Handle **blog likes** (no dislikes).

🔹 Functions

a) `likeBlog`

Process

1. Verify blog exists
2. Check if already liked
3. Add like
4. Return updated like count

Output

{ "success": true, "likeCount": n }

---

b) `unlikeBlog`

Process

1. Verify blog exists
2. Remove like
3. Return updated like count

Output

{ "success": true, "likeCount": n }

---

c) `getLikeStatus`

Process

1. Check if user liked blog
2. Fetch total like count

Output

{ "success": true, "hasLiked": true, "likeCount": n }

5️⃣ Resource Controller (`controllers/resourceController.js`)

🎯 Purpose

Manage **academic resources** (student-only access).

🔹 Functions

a) `getResources`

Process

1. Fetch resources for user's college

2. Apply pagination

---

b) `getResourceById`

Process

1. Fetch resource
2. Verify college access

---

c) `createResource` (Admin/Moderator)

Process

1. Validate input
2. Create resource for college

---

d) `updateResource` / `deleteResource`

- Similar to blog update/delete logic

6️⃣ Local Guide Controller (`controllers/localGuideController.js`)

🎯 Purpose

Manage **local places and ratings**.

🔹 Functions

a) `getPlaces`

Process

1. Fetch places for college

2. Filter by category (optional)
3. Calculate average ratings

---

b) `getPlacesByCategory`

Process

1. Fetch places by category
2. Include ratings

---

c) `addRating`

Process

1. Validate rating (1–5)
2. Add or update rating
3. Recalculate average rating

---

d) `getCategories`

Output

{ "success": true, "categories": [...] }

🔄 How Controllers Work Together (Example: Create Blog)

1. **Request**

POST /api/blogs

Authorization: Bearer <token>

Body: { blog_title, blog_content }

2. **Route**

router.post('/blogs', authenticate, requireStudent, createBlog);

3. **Middleware**
- `authenticate` → verifies token, sets `req.user`
- `requireStudent` → ensures student access
4. **Controller Execution**
- Reads `req.user`
- Validates input
- Calls `Blog.create()`
5. **Model → Database**
- SQL executed
- Blog created
6. **Response**

{ "success": true, "blog": { ... } }

🔐 Security Features in Controllers

- **Authentication Checks**
  - Uses `req.user`
  - No direct access without token
- **Authorization Checks**
  - Ownership verification
  - Role-based checks
- **Input Validation**
  - Required fields
  - Data type checks
  - Content sanitization
- **Error Handling**
  - Try-catch blocks
  - Consistent error responses
  - No sensitive data leaks

📦 Controller Response Format

✅ Success Response

{ "success": true, "message": "Operation successful", "data": { ... } }

❌ Error Response

{ "success": false, "message": "Error message here" }

✅ Benefits

- 📁 Organized code structure
- ♻️ Reusable business logic
- 🧪 Easy unit testing
- 🛠️ Simple maintenance
- 🔐 Built-in security checks

🍽️ Real-World Analogy

Think of controllers like **restaurant staff**:

- **Routes** → Host (directs customers)
- **Controllers** → Waiter (takes and processes orders)
- **Models** → Kitchen (prepares food)
- **Database** → Pantry (stores ingredients)

The Waiter (Controller):

- Takes the order (request)
- Checks permissions (authorization)
- Sends order to kitchen (model)
- Brings food back (response)

Phase 5: Routes (API Endpoints) — Explanation

What are routes?

Routes map HTTP requests to controller functions. They:

- **Define API endpoints (URLs)**
- **Specify HTTP methods (GET, POST, PUT, DELETE)**
- **Apply middleware (authentication, validation, authorization)**
- **Connect URLs to controller functions**

Think of routes as the "receptionist" of your API:
Client Request → Route → Middleware → Controller → Response

Why we need routes

- **URL structure: Define clean, RESTful endpoints**
- **Request handling: Route requests to the right controller**
- **Middleware chain: Apply auth, validation, etc.**
- **Organization: Separate routing from business logic**

What we created

1. Auth Routes (`routes/authRoutes.js`)

Purpose: Handle user authentication endpoints

Routes:

a) POST /api/auth/register — Register new user

- **URL: POST `/api/auth/register`**
- **Body:**

```
{
  "email": "student@college.edu",
  "password": "password123",
  "reg_no": "2024CS001",
  "first_name": "John",
  "last_name": "Doe",
  "course_id": 1,
  "graduation_year": 2027,
  "date_of_birth": "2000-01-01"
}
```

- **Middleware Chain:**
    1. **Validation (express-validator)**
        - **Email must be valid**
        - **Password min 6 characters**
        - **Required fields checked**
    2. **handleValidationErrors — Returns errors if validation fails**
    3. **register (controller) — Creates user and returns token**

b) POST /api/auth/login — Login user

- **URL:** POST `/api/auth/login`
- **Body:**

```
{
  "email": "student@college.edu",
  "password": "password123"
}
```

- **Middleware Chain:**
  1. **Validation** — Email format, password required
  2. **handleValidationErrors**
  3. **login (controller)** — Verifies credentials, returns token

c) GET /api/auth/profile — Get user profile

- **URL:** GET `/api/auth/profile`
- **Headers:** Authorization: Bearer `<token>`
- **Middleware Chain:**
  1. **authenticate** — Verifies JWT token, adds `req.user`
  2. **getProfile (controller)** — Returns user details

---

2. Blog Routes (`routes/blogRoutes.js`)

Purpose: Manage blog posts

Routes:

a) GET /api/blogs — Get all blogs

- **URL: GET** `/api/blogs?page=1&limit=10&collegeId=5`
- **Query Params: page (default: 1), limit (default: 10), collegeId (optional)**
- **Middleware: optionalAuth (public)**
- **Controller: getAllBlogs**

b) GET /api/blogs/:id — Get single blog

- **URL: GET** `/api/blogs/123`
- **Params: id = 123**
- **Middleware: optionalAuth**
- **Controller: getBlogById**

c) POST /api/blogs — Create blog

- **URL: POST** `/api/blogs`
- **Headers: Authorization: Bearer** `<token>`
- **Body:**

```
{
  "blog_title": "My First Blog",
  "blog_content": "This is my blog content..."
}
```

- **Middleware Chain:**
    1. **authenticate**
    2. **requireStudent**
    3. **Validation — blog_title required (max 128 chars), blog_content required**
    4. **handleValidationErrors**
    5. **createBlog (controller)**

d) PUT /api/blogs/:id — Update blog

- **URL: PUT `/api/blogs/123`**
- **Headers: Authorization: Bearer `<token>`**
- **Body:**

```
{
  "blog_title": "Updated Title",
  "blog_content": "Updated content..."
}
```

- **Middleware Chain:**
    1. **authenticate**
    2. **requireStudent**
    3. **Validation (optional fields)**
    4. **handleValidationErrors**

5. updateBlog (controller — checks ownership)

e) DELETE /api/blogs/:id — Delete blog

- URL: DELETE `/api/blogs/123`
- Headers: Authorization: Bearer `<token>`
- Middleware Chain:
    1. authenticate
    2. requireStudent
    3. deleteBlog (controller — checks ownership)

---

3. Comment Routes (`routes/commentRoutes.js`)

Purpose: Manage blog comments

Routes:

a) GET /api/blogs/:id/comments — Get comments

- URL: GET `/api/blogs/123/comments`
- Middleware: optionalAuth (public)
- Controller: getCommentsByBlogId

b) POST /api/blogs/:id/comments — Add comment

- URL: POST `/api/blogs/123/comments`
- Headers: Authorization: Bearer `<token>`
- Body:

```
{
  "comment_content": "Great blog post!"
}
```

- **Middleware Chain:**
   1. **authenticate**
   2. **requireStudent**
   3. **Validation — comment_content max 1000 chars**
   4. **handleValidationErrors**
   5. **createComment (controller)**

c) DELETE /api/comments/:commentId — Delete comment

- **URL: DELETE** `/api/comments/456`
- **Headers: Authorization: Bearer** `<token>`
- **Middleware Chain:**
   1. **authenticate**
   2. **requireStudent**
   3. **deleteComment (controller — checks ownership)**

---

4. Reaction Routes (`routes/reactionRoutes.js`)

Purpose: Handle blog likes

Routes:

a) GET /api/blogs/:id/like-status — Check like status

- **URL: GET `/api/blogs/123/like-status`**
- **Headers: Authorization: Bearer `<token>`**
- **Middleware: authenticate**
- **Controller: getLikeStatus**
- **Returns:**

```
{ "hasLiked": true, "likeCount": 5 }
```

b) POST /api/blogs/:id/like — Like blog

- **URL: POST `/api/blogs/123/like`**
- **Headers: Authorization: Bearer `<token>`**
- **Middleware Chain:**
    1. **authenticate**
    2. **requireStudent**
    3. **likeBlog (controller)**

c) DELETE /api/blogs/:id/like — Unlike blog

- **URL: DELETE `/api/blogs/123/like`**
- **Headers: Authorization: Bearer `<token>`**
- **Middleware Chain:**
    1. **authenticate**

2. requireStudent
3. unlikeBlog (controller)

---

5. Resource Routes (`routes/resourceRoutes.js`)

Purpose: Manage academic resources (student-only)

Routes:

a) GET /api/resources — Get resources

- **URL: GET `/api/resources?page=1&limit=10`**
- **Headers: Authorization: Bearer `<token>`**
- **Middleware Chain: authenticate → requireStudent → getResources**

b) POST /api/resources — Create resource

- **URL: POST `/api/resources`**
- **Headers: Authorization: Bearer `<token>`**
- **Body:**

```
{
  "resource_title": "CS101 Notes",
  "resource_description": "Introduction to CS",
  "resource_link": "https://example.com/notes"
}
```

- **Middleware Chain:**
    1. **authenticate**
    2. **requireModeratorOrAdmin**
    3. **Validation (title, URL)**
    4. **handleValidationErrors**
    5. **createResource (controller)**

---

6. Local Guide Routes (`routes/localGuideRoutes.js`)

Purpose: Manage local places and ratings

Routes:

a) GET /api/local-guide/categories — Get categories

- **URL: GET `/api/local-guide/categories`**
- **Public: No authentication required**
- **Controller: getCategories**

b) GET /api/local-guide/places — Get places

- **URL: GET `/api/local-guide/places?collegeId=5`**
- **Query: collegeId (required)**
- **Public: optionalAuth**
- **Controller: getPlaces**

c) GET /api/local-guide/places/:category — Get by category

- URL: GET `/api/local-guide/places/healthcare?collegeId=5`
- Params: category = "healthcare"
- Public: optionalAuth
- Controller: getPlacesByCategory

d) POST /api/local-guide/places/:id/rating — Rate place

- URL: POST `/api/local-guide/places/789/rating`
- Headers: Authorization: Bearer `<token>`
- Body:

```
{ "rating": 5 }
```

- Middleware Chain: authenticate → requireStudent → Validation (1-5) → handleValidationErrors → addRating

---

How routes work together

Example: Creating a blog post

1. Client sends request:

```
POST /api/blogs

Headers: Authorization: Bearer <token>

Body: { blog_title: "...", blog_content: "..." }
```

2. **Express matches route in `blogRoutes.js`**
3. **Middleware executes in order:**
   - **authenticate → verify JWT → add `req.user`**
   - **requireStudent → checks access**
   - **Validation → checks required fields and lengths**
   - **handleValidationErrors → returns errors if any**
4. **Controller executes → createBlog(req, res) → creates blog**
5. **Response sent to client:**

```
{
  "success": true,
  "blog": { "blog_id": 456, ... }
}
```

Route patterns

- **Public routes:** `router.get('/categories', getCategories)`
- **Optional auth:** `router.get('/blogs', optionalAuth, getAllBlogs)`
- **Authenticated routes:** `router.get('/profile', authenticate, getProfile)`
- **Student-only routes:** `router.post('/blogs', authenticate, requireStudent, createBlog)`

- **Admin/moderator routes:** `router.post('/resources', authenticate, requireModeratorOrAdmin, createResource)`

Validation in routes

Express-validator examples:

```
body('email')

  .isEmail()

  .withMessage('Invalid email')

  .normalizeEmail(),

body('password')

  .isLength({ min: 6 })

  .withMessage('Password too short'),

body('rating')

  .isInt({ min: 1, max: 5 })
```

If validation fails:

```
{

  "success": false,

  "message": "Validation failed",

  "errors": [

    { "field": "email", "message": "Invalid email", "value":
"invalid-email" }

  ]
}
```

RESTful design principles

- GET: Read data (no side effects)
- POST: Create new resource
- PUT: Update existing resource
- DELETE: Remove resource

URL examples:

- `/api/blogs` — Collection of blogs
- `/api/blogs/:id` — Single blog
- `/api/blogs/:id/comments` — Comments for a blog
- `/api/blogs/:id/like` — Like action for a blog

Benefits

- Clear API structure
- Security via middleware
- Input validation
- Maintainability
- Scalability

Real-world analogy

Think of routes like a restaurant menu:

- **Route: Menu item (what you can order)**
- **Middleware: Kitchen rules (who can order, when)**
- **Controller: Chef (prepares the order)**
- **Response: Food (what you get back)**

Example:

- **Menu item: "Create Blog Post"**
- **Rules: Must be logged in, must be student**
- **Chef: Prepares blog post**
- **Result: Blog post created**

Phase 6: App Integration & Security — Explanation

What is app.js?

`app.js` **is the main Express application file. It:**

- **Configures the Express app**
- **Sets up middleware**
- **Mounts all routes**
- **Handles errors globally**

Think of it as the "control center" that connects everything.

What we did in Phase 6

1. Added security middleware (Helmet)

```
app.use(helmet());
```

Purpose: Sets HTTP security headers

What it does:

- **Prevents XSS attacks**
- **Prevents clickjacking**
- **Hides Express version**
- **Enforces HTTPS in production**
- **Sets content security policies**

Example headers added:

```
X-Content-Type-Options: nosniff

X-Frame-Options: DENY

X-XSS-Protection: 1; mode=block
```

---

2. Added logging middleware (Morgan)

```
if (process.env.NODE_ENV === 'development') {

  app.use(morgan('dev'));

} else {
  app.use(morgan('combined'));
}
```

Purpose: Logs HTTP requests

What it logs:

- **Request method (GET, POST, etc.)**
- **Request URL**
- **Response status code**
- **Response time**
- **IP address**

Example output:

```
GET /api/blogs 200 45ms
POST /api/auth/login 200 120ms
GET /api/resources 401 5ms
```

Why different modes:

- **Development:** `dev` **— concise, colored output**
- **Production:** `combined` **— detailed logs for analysis**

---

3. Imported all routes

```
const authRoutes = require('./routes/authRoutes');
const blogRoutes = require('./routes/blogRoutes');
const commentRoutes = require('./routes/commentRoutes');

const reactionRoutes = require('./routes/reactionRoutes');

const resourceRoutes = require('./routes/resourceRoutes');

const localGuideRoutes = require('./routes/localGuideRoutes');
```

Purpose: Loads all route modules

What this does:

- **Imports route definitions**
- **Makes routes available to mount**
- **Keeps code organized**

---

4. Mounted all routes

```
app.use('/api/auth', authRoutes);

app.use('/api/blogs', blogRoutes);

app.use('/api/blogs', commentRoutes);

app.use('/api/blogs', reactionRoutes);

app.use('/api/resources', resourceRoutes);

app.use('/api/local-guide', localGuideRoutes);
```

Purpose: Connects routes to URLs

How it works:

- `app.use('/api/auth', authRoutes)` → Routes in `authRoutes` are prefixed with `/api/auth`
- `POST /register` in `authRoutes` becomes `POST /api/auth/register`

Route mounting overview:

- `/api/blogs` → Blog routes
- `/api/blogs` → Comment routes (nested)
- `/api/blogs` → Reaction routes (nested)
- `/api/resources` → Resource routes
- `/api/local-guide` → Local guide routes

---

5. Added 404 handler

```
app.use(notFound);
```

Purpose: Handles undefined routes

What it does:

- Catches requests to routes that don't exist
- Returns a 404 error
- Must be before error handler

Example:

```
GET /api/invalid-route

→ Returns: { success: false, message: "Route /api/invalid-route
not found" }
```

---

6. Added global error handler

```
app.use(errorHandler);
```

Purpose: Catches all errors

What it does:

- **Catches errors from any middleware or route**
- **Formats error responses consistently**
- **Handles specific error types (MySQL, JWT, validation)**
- **Must be last middleware**

Example errors handled:

- **Database errors → 500 or 400**
- **JWT errors → 401**
- **Validation errors → 400**
- **Not found → 404**

Middleware execution order

```
Request comes in

    ↓

1. helmet()           → Adds security headers
```

```
    ↓
2. cors()            → Handles CORS
    ↓
3. express.json()    → Parses JSON body
    ↓
4. express.urlencoded() → Parses form data
    ↓
5. morgan()          → Logs request
    ↓
6. Routes            → Matches URL to route
      Route middleware → Auth, validation, etc.
      Controller       → Business logic
      Response sent
    ↓
7. notFound()        → If route not found
    ↓
8. errorHandler()    → If error occurred
```

Important: Order matters. Error handlers must be last.

Complete `app.js` structure

```
// 1. Imports
```

```
const express = require('express');
const helmet = require('helmet');
const morgan = require('morgan');
// ... other imports

// 2. Create Express app
const app = express();

// 3. Security & parsing middleware (order matters!)
app.use(helmet());           // Security first
app.use(cors());             // Then CORS
app.use(express.json());     // Then body parsing
app.use(morgan());           // Then logging

// 4. Routes
app.get('/', rootHandler);   // Root route
app.use('/api/auth', authRoutes);
app.use('/api/blogs', blogRoutes);
// ... other routes

// 5. Error handling (must be last!)
app.use(notFound);           // 404 handler
app.use(errorHandler);       // Global error handler
```

Why this structure matters

- **Security first: Helmet sets headers before anything else → protects against common attacks**
- **Parsing before routes: Ensures `req.body` is available**
- **Routes before error handlers: Routes try to match first**
- **Error handlers last: Catch errors from anywhere**

Real-world analogy

Think of `app.js` like a restaurant:

- **Helmet: Security guard (checks everyone)**
- **CORS: Host (decides who can enter)**
- **Body parsing: Waiter (takes your order)**
- **Morgan: Receptionist (logs who came)**
- **Routes: Menu (what you can order)**
- **404 handler: "We don’t serve that"**
- **Error handler: Manager (handles problems)**

Benefits

- **Security: Helmet protects against attacks**
- **Monitoring: Morgan logs all requests**
- **Organization: Routes are centralized**
- **Error handling: Consistent error responses**
- **Maintainability: Easy to add new routes**
- **Scalability: Structure supports growth**

What happens when a request comes in

Example: `POST /api/blogs`

1. **Request arrives → Headers: Authorization: Bearer** `<token>` **→ Body:** `{ blog_title: "...", blog_content: "..." }`
2. **Helmet adds security headers →** `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY`
3. **CORS checks origin → Allows request**
4. `express.json()` **parses body →** `req.body = { blog_title: "...", blog_content: "..." }`
5. **Morgan logs request →** `POST /api/blogs 200 45ms`
6. **Routes match URL → Found:** `POST /api/blogs` **in** `blogRoutes`
7. **Route middleware executes →** `authenticate` **(verifies token),** `requireStudent` **(checks role), Validation (checks input)**
8. **Controller executes →** `createBlog()` **creates blog**
9. **Response sent →** `{ success: true, blog: {...} }`

10. If error occurred → `errorHandler` catches it

Environment-based configuration

```
if (process.env.NODE_ENV === 'development') {
  app.use(morgan('dev'));      // Simple logs
} else {
  app.use(morgan('combined')); // Detailed logs
}
```

Why:

- Development → Easier to read, less verbose
- Production → Detailed logs for debugging and monitoring

Summary

Phase 6 integrated:

- Security (Helmet)
- Logging (Morgan)
- All routes
- Error handling
- 404 handling

Result: The backend is now:

- **Secure**
- **Monitored**
- **Organized**
- **Error-resilient**
- **Production-ready**