Full-Stack Guideline – Domain/Category/Product Catalog Website

Full-Stack Guideline – Domain/Category/Product Catalog Website

1 Overview

The planned application is a catalog-style web site for sharing resources on **Claude Code agents**, **MCP servers and references**, **AI use-cases**, **AI-based platforms** (chat, IDE, document generation etc.) and a **blog**. Each subject area is treated as a **domain**. Within each domain there are **categories**, and each category contains one or more **products**. Users should be able to browse domains, drill down into categories, view products in a grid of cards (thumbnail, name, subtitle) and open a detailed product page with description and downloadable attachments. An authenticated admin will have a dashboard to create, update or delete domains, categories and products and to upload attachments.

The stack consists of:

- **Backend:** Node.js with Express for REST APIs and **MongoDB** for the database. W3Schools highlights that Express routers organize routes and controllers handle request logicw3schools.com, and models encapsulate data accessw3schools.com. The API will expose CRUD endpoints for domains, categories, products and attachments. JWT-based authentication protects admin endpoints.
- **Frontend: Next.js** for the user interface. A Next.js example project can be bootstrapped with the `with-mongodb` template to configure a MongoDB connectionblog.openreplay.com. Next.js API routes are serverless functions under `/pages/api` that act as endpointsblog.openreplay.com, but in this design the main REST API is handled by the Express backend; Next.js will call it via fetch/axios. Next.js pages will implement the home page, domain pages, category pages, product pages and admin pages. Data can be fetched at request time using `getServerSideProps` blog.openreplay.com to ensure fresh information.

2 Features

2.1 Domain list (Home page)

- **Card grid:** The home page lists all domains in a grid. Each card shows the domain's name, a short description and possibly an icon. Clicking a card navigates to `/domain/[slug]`.
- **Search/filter:** Provide text search or filters to quickly locate a domain.
- **Blog preview:** Optionally include a section with recent blog posts.

2.2 Domain page

- Shows the domain's full description and lists the **categories** belonging to that domain. Each category appears as a card or list item.
- Clicking a category navigates to `/category/[slug]`.

2.3 Category page

- Displays all products under the selected category in a **responsive grid**. Each product card contains a thumbnail image, the product name and a subtitle. Use CSS Grid or a UI library (e.g., Tailwind CSS, Material UI) to create the grid layout.
- The card is clickable and links to the product detail page.

2.4 Product detail page

- Shows detailed information: product title, subtitle, full description, tags, published date, author, pricing (if applicable), etc.
- **Attachments:** A list of downloadable attachments (PDF, DOCX, ZIP, etc.). Each attachment links to a protected `/api/download/:id` endpoint that validates permission and streams the file.
- **Images/videos:** Additional media such as screenshots or demo videos (optional). Provide a gallery slider.

2.5 Blog section

- A separate blog domain with categories and posts. Use a **Markdown** or rich-text editor in admin to write blog posts.
- Posts support tags, cover images and attachments.

2.6 Admin dashboard

- **Authentication:** Admins must log in via username/password and receive a JWT. Authenticated requests must include the token in the `Authorization` header.
- **Domain management:** Create, list, update and delete domains. Each domain has a name, slug (URL-friendly identifier), description, hero image and sort order.
- **Category management:** Within a domain, admins create categories with name, slug, description and sort order.
- **Product management:** For each category, admins create products with fields such as name, slug, subtitle, description (Markdown), thumbnails, attachments, tags and SEO metadata. Products can be activated/deactivated.
- **Attachment upload:** Use a file upload form to upload attachments. Files are stored in a `uploads/` folder or cloud storage (e.g., AWS S3). Metadata (file name, mime type, size, product reference) is stored in MongoDB.
- **Blog management:** Manage blog categories and posts using similar CRUD pages.
- **Dashboard UI:** Use Next.js pages (`/admin` , `/admin/domains` , `/admin/categories` , `/admin/products`) with forms and tables. Use client-side libraries like React Hook Form and SWR/React Query for data fetching and state management.

3 Database Design (MongoDB)

Use **Mongoose** as an ODM. Collections and sample schemas are:

3.1 Domain

```
1 js
```

CopyEdit

```
const domainSchema = new mongoose.Schema({   name: { type: String, required: true },
slug: { type: String, unique: true, required: true },   description: String,
heroImage: String, // URL or path   createdAt: { type: Date, default: Date.now } });
```

3.2 Category

```
1 js
```

CopyEdit

```
const categorySchema = new mongoose.Schema({   domain: { type: mongoose.Schema.Types.ObjectId, ref: 'Domain', required: true },   name: { type: String, required: true },   slug: { type: String, unique: true, required: true },   description: String,   sortOrder: Number });
```

3.3 Product

```
1 js
```

CopyEdit

```
const productSchema = new mongoose.Schema({   category: { type: mongoose.Schema.Types.ObjectId, ref: 'Category', required: true },   name: { type: String, required: true },   slug: { type: String, unique: true, required: true },   subtitle: String,   description: String, // markdown or HTML   thumbnail: String, // image path or URL   tags: [String],   attachments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Attachment' }],   isActive: { type: Boolean, default: true },   createdAt: { type: Date, default: Date.now },   updatedAt: { type: Date, default: Date.now } });
```

3.4 Attachment

```
1 js
```

CopyEdit

```
const attachmentSchema = new mongoose.Schema({   product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product', required: true },   filename: String,   originalName: String,   mimeType: String,   size: Number,   url: String, // file location or signed URL   uploadedAt: { type: Date, default: Date.now } });
```

3.5 BlogPost and BlogCategory (optional)

Define similar schemas for blog categories and blog posts with fields like title, slug, content, tags, cover image, attachments, and author.

4 Backend Implementation (Node.js/Express)

4.1 Project Structure

Use the organization recommended by W3Schoolsw3schools.com:

```
1 bash
```

CopyEdit

```
backend/   app.js               # Main entry point   routes/             # Express routers     domains.js     categories.js     products.js     attachments.js     auth.js     blog.js   controllers/        # Request handlers     domainController.js     categoryController.js     productController.js     attachmentController.js     authController.js     blogController.js   models/             # Mongoose models     Domain.js     Category.js     Product.js     Attachment.js     User.js     BlogPost.js     BlogCategory.js   middleware/         # Custom middleware     auth.js           # JWT verification     validation.js    # Request validation     upload.js         # File
```

```
upload via multer   utils/     errorHandler.js   # Centralized error handler   config/
db.js             # MongoDB connection     env.js            # Environment variables
```

This separation keeps routes, controllers and models decoupledw3schools.com. Each router file defines URL paths and delegates to controller functions. Controllers implement the business logic and call the Mongoose models.

4.2 Database Connection

Create `config/db.js` to connect to MongoDB using Mongoose. Use environment variables (e.g., `MONGODB_URI`) loaded via `dotenv`. Test the connection at startup.

4.3 Routes and Controllers

Use **Express Router** to organize endpoints. W3Schools demonstrates defining a router and mapping HTTP verbs to controller functionsw3schools.com. For example, `routes/products.js` may look like:

```
1 js
```

CopyEdit

```
const express = require('express'); const router = express.Router(); const
productController = require('../controllers/productController'); const auth =
require('../middleware/auth');  // Public endpoints router.get('/',
productController.listProducts); router.get('/:id', productController.getProduct);  //
Admin endpoints - require authentication router.post('/', auth,
productController.createProduct); router.put('/:id', auth,
productController.updateProduct); router.delete('/:id', auth,
productController.deleteProduct);  module.exports = router;
```

Controllers implement functions such as `listProducts`, `createProduct`, etc. They catch errors and return JSON responses. Use `async/await` and try/catch. W3Schools illustrates a controller that retrieves and creates users using model methods and sends JSON responsesw3schools.com.

4.4 Request Validation

Validate request bodies using libraries like Joi or express-validator. W3Schools shows an example of validating a request body with Joi to ensure data integrityw3schools.com. Create a `middleware/validation.js` that defines schemas for each endpoint and returns 400 responses when validation fails.

4.5 Authentication Middleware

Implement JWT authentication in `middleware/auth.js`. When a request includes an `Authorization` header containing a valid token, the middleware attaches the user to `req.user`. Unauthorized requests result in 401 errors. Only admin routes use the middleware.

4.6 File Uploads

Use `multer` to handle multipart/form-data uploads. Define a `storage` configuration with a destination (`uploads/attachments/`) and a filename generator. In `routes/attachments.js`, create endpoints:

- `POST /attachments/upload` – Accepts a file and product ID, stores the file and metadata.
- `GET /attachments/:id/download` – Streams the file. Ensure the user has permission.

4.7 Error Handling

Create an error handling utility (e.g., `utils/errorHandler.js`) that throws custom errors with status codes. Use a centralized error-handling middleware to capture thrown errors and send consistent responses[w3schools.com](w3schools.com).

4.8 Sample API Endpoints

Resource	Endpoint	Method	Description
Domains	`/api/domains`	GET	List all domains.
	`/api/domains/:id`	GET	Get one domain by ID or slug.
	`/api/domains`	POST (admin)	Create a domain.
	`/api/domains/:id`	PUT (admin)	Update a domain.
	`/api/domains/:id`	DELETE (admin)	Delete a domain.
Categories	`/api/domains/:domainId/categories`	GET	List categories for a domain.
	`/api/categories/:id`	GET	Get one category.
	`/api/domains/:domainId/categories`	POST (admin)	Create a category within a domain.
	`/api/categories/:id`	PUT (admin)	Update a category.
	`/api/categories/:id`	DELETE (admin)	Delete a category.
Products	`/api/categories/:categoryId/products`	GET	List products for a category.
	`/api/products/:id`	GET	Get a product's detail (with

			attachments).
	`/api/categories/:categoryId/products`	POST (admin)	Create a product.
	`/api/products/:id`	PUT (admin)	Update a product.
	`/api/products/:id`	DELETE (admin)	Delete a product.
Attachments	`/api/products/:productId/attachments`	GET	List attachments for a product.
	`/api/products/:productId/attachments`	POST (admin)	Upload an attachment.
	`/api/attachments/:id/download`	GET	Download attachment file.
Authentication	`/api/auth/login`	POST	Admin login (returns JWT).
	`/api/auth/register`	POST	(optional) Register new admin.
Blog	`/api/blog/posts`	CRUD endpoints similar to products.	

5 Frontend Implementation (Next.js)

5.1 Project Setup

Start with the `with-mongodb` example when creating the Next.js app. Running `npx create-next-app --example with-mongodb myapp` bootstraps a project with MongoDB configured[blog.openreplay.com](). This template includes a `lib/mongodb.ts` file that connects to MongoDB using the connection string from `.env.local` [blog.openreplay.com]().

In this project the backend is separate, but you can still reuse the template's structure (the library file can be removed if not needed). Next.js pages reside in the `pages` directory. Use the new **App Router** or Pages Router depending on your Next.js version.

5.2 Directory Structure

```bash
```

CopyEdit

```
frontend/   pages/     index.tsx                # Home page - list domains
domain/[slug].tsx        # Domain page - list categories     category/[slug].tsx
# Category page - list products     product/[slug].tsx       # Product details
blog/index.tsx           # Blog home     blog/[slug].tsx          # Blog post details
admin/login.tsx          # Admin login page     admin/domains.tsx        # Manage
domains     admin/categories.tsx     # Manage categories     admin/products.tsx
# Manage products     admin/blog.tsx           # Manage blog posts     api/
# If you choose to use Next.js API routes (optional)   components/     Layout.tsx
Navbar.tsx     DomainCard.tsx     CategoryCard.tsx     ProductCard.tsx
AttachmentList.tsx     AdminForm.tsx   utils/     fetcher.ts               # Wrapper
around fetch/axios     auth.ts                  # JWT handling   styles/     ... CSS
or Tailwind classes
```

5.3 Data Fetching

Use **getServerSideProps** or the new `fetch` with `cache: 'no-store'` to fetch data from the backend at request time. The OpenReplay tutorial shows using `getServerSideProps` to fetch posts from an API and return them to the page's propsblog.openreplay.com. This approach ensures the page is rendered with up-to-date data on each request. For example, in `pages/index.tsx` you can fetch domains:

```ts
```

CopyEdit

```
export async function getServerSideProps() {   const res = await
fetch(process.env.API_URL + '/api/domains');   const domains = await res.json();
return { props: { domains } }; }
```

Inside the component, render the domains:

```tsx
```

CopyEdit

```
export default function Home({ domains }) {   return (     <Layout>
<h1>Domains</h1>       <div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3
gap-4">         {domains.map((d) => (           <DomainCard key={d._id} domain={d} />
))}       </div>     </Layout>   ); }
```

For the admin pages you may use client-side data fetching with SWR or React Query to provide real-time updates and caching.

5.4 Routing and Dynamic Pages

Next.js uses file names as routes. Dynamic routes like `pages/domain/[slug].tsx` capture the slug parameter. In the page's `getServerSideProps` fetch the domain by slug and its categories. Similarly, `pages/category/[slug].tsx` fetches products for that category. `pages/product/[slug].tsx` fetches the product details and attachments.

5.5 Grid Layout and Cards

Use **CSS Grid** or a framework like Tailwind to create responsive grids. Each product card component displays the thumbnail, product name and subtitle. Cards should have hover effects and clickable area to navigate. Use `next/image` for optimized images and set `priority` for above-the-fold content.

5.6 Admin Forms

Implement forms for domain, category and product creation using React Hook Form or Formik. Use controlled inputs and display validation errors. On submission, send POST/PUT requests to the backend with JSON data or multipart/form-data for file uploads.

5.7 Authentication Handling

Store the JWT returned from the login endpoint in a cookie or localStorage. Wrap admin pages with a higher-order component that checks authentication. Include the JWT in the `Authorization` header when calling admin APIs.

5.8 File Download

To download attachments, provide anchor tags or buttons linking to the `download` endpoint. The browser will prompt the user to save the file. Ensure you send proper `Content-Disposition` and `Content-Type` headers from the backend.

6 Interaction Between Frontend and Backend

1. **HTTP requests:** The Next.js front-end uses fetch or axios to call the Express API. Each call should include necessary headers and handle errors.
2. **JSON responses:** Backend endpoints return JSON objects containing data or error messages. The front-end parses JSON and updates the UI.
3. **Authentication:** The login page calls `/api/auth/login` with credentials and receives a JWT. This token is stored and added to subsequent admin requests.
4. **Real-time updates:** For the admin dashboard, use SWR or React Query to keep lists of domains, categories and products in sync. After create/update/delete operations, refetch the list or optimistically update the cache.

7 Security Considerations

- **Input validation:** Validate all user input on the server side using Joi or express-validator as recommendedw3schools.com. Sanitize strings to prevent injection attacks.
- **Authentication and authorization:** Secure admin routes using JWT. Hash passwords using bcrypt and store salted hashes in the User collection. Use HTTPS in production.
- **File upload security:** Restrict allowed file types and file sizes. Store files outside the web root. Validate attachment references before serving downloads.
- **Rate limiting and CORS:** Implement rate limiting for public endpoints and configure CORS to allow only the front-end domain.
- **Error handling:** Do not expose stack traces to clients. Use centralized error handling middleware to return generic error messagesw3schools.com.

8 Deployment

- **Environment variables:** Store secrets (database URI, JWT secret) in environment variables. In development, use `.env.local`. In production, configure them via your hosting platform.
- **Backend deployment:** Deploy the Node.js/Express server on a platform such as **Heroku**, **DigitalOcean**, **AWS EC2** or **Railway**. Use a process manager (e.g., PM2) to run the app and enable automatic restarts. Configure

HTTPS (via Nginx reverse proxy or hosting provider).

- **Frontend deployment:** Deploy the Next.js front-end on **Vercel**, **Netlify** or another static hosting provider. Set `NEXT_PUBLIC_API_URL` to the backend API base URL. Build the project with `npm run build` and deploy the `.next` directory.
- **Database:** Use managed MongoDB such as **MongoDB Atlas** or self-hosted MongoDB. Ensure network access is restricted to the backend server.
- **CI/CD:** Configure continuous deployment to automate builds and tests. Use GitHub Actions or GitLab CI to run linting, tests and deployments when changes are pushed.

9 Conclusion

This guideline outlines a full-stack architecture for a domain/category/product catalog website using Node.js, Express and MongoDB on the backend and Next.js on the frontend. Following best practices—such as separating routes, controllers and modelsw3schools.com, using server-side data fetchingblog.openreplay.com, validating requestsw3schools.com and handling errors centrallyw3schools.com—will help you build a maintainable application. Implementing features such as dynamic routing, a responsive grid layout, secure file uploads and an admin dashboard will deliver a rich user experience and allow easy management of domains, categories and products.