Technical Requirements Document for Project "IT-RAYS website Page Builder"

Presented to	**IT-RAYS**
Project Name	IT-RAYS Page Builder Technical Requirements
Document Number	IRTRD001
Document Type	Technical Requirements
Date	11-July-2025
Version	1.0.0
Prepared By	Mohamed Montaser (CEO)

Project Overview

This document outlines the core technical requirements for the development of the Page Builder application. It defines the technology stack, architectural patterns, coding standards, development practices, and deployment considerations for both the Next.js frontend and Laravel/MySQL backend. Adherence to these guidelines is mandatory for all development team members to ensure a cohesive, high-quality, scalable, and maintainable product.

1. IT-RAYS

- **About IT-RAYS:**
  A company where you can find every beautiful mind and talented resource. We specialize in helping other developers make it easier to code and to live. We provide several services free of charge Web Services for many things on the web passing through Web and Mobile applications development with the best existing User Experience and ending to Creating hand-crafted WordPress themes on Envato marketplace. We also provide custom solutions on a large scale for corporate levels. Our products have the highest level of Quality ever existing.

- **IT-RAYS History:**
  IT-RAYS was established in 2012 and believe in changing the world to go better. The company originally worked all over the world and has established many successful projects in the Middle East and the globe.
  The owner of the company has more than 19 years of experience in the IT sector and has led many businesses to grow exponentially. He believes that the open market is the key value that can take you to the remarkable success as well as the worldwide popularity.

- **IT-RAYS Vision:**
  We believe that we can deliver a value to all our customers in a way that really matches every client's needs. Not only delivering a value but also a high-quality value in the key value for our success stories. This we believe can make everyone's life better by changing the quality of services for every client to the highest level.

- **IT-RAYS Services:**
  We provide many services on the market, below are some examples of the service (but not limited to)
  1- Custom professional software solutions.
  2- User experience, UX review and UX design.
  3- Hand-crafted unique WordPress themes.
  4- Mobile applications development.
  5- Website and Web Portals development.

2. Summary

Why use Our Custom Framework?

Unlike many providers that simply offer standard Laravel PHP solutions, we dive deep into the core of Laravel to craft exceptional, elegant software that elevates the customer's needs and provide unique, stable, and secure software for our customers in one place.

Our dedication to quality and innovation has resulted in software that is not only just functional as per customer requirements, but also highly dynamic, re-usable systems and our commitment to excellence has made our products top-level. With a focus on customization, performance, and responsive design, we go the extra mile to ensure that our software meets the diverse needs of our clients, setting their websites apart in a crowded online space.

Objectives:

- Create a modern, visually appealing website to promote the IT-RAYS using the latest trends and standards.
- Ensure ease of use for both administrators (through CMS dashboard) and end-users (visitors through Frontend).
- Incorporate a responsive design for seamless accessibility across devices (desktop, mobile, tablet, etc..).
- Ensure multilingual support (Arabic & English) with a potential for further language expansion.

3. Frontend Technical Requirements (Next.js)

3.1. Core Stack & Libraries

- **Framework:** Next.js (LTS version, latest stable v15.x.x)
- **Language:** TypeScript.
- **Package Manager:** npm
- **UI Library (Optional):** None initially, focus on custom components. Consider using (e.g., rSuite, shadcn) for accessibility and styled components if needed later.
- **Styling:** Tailwind CSS (v4.x.x) with PostCSS and Autoprefixer. Utilize tailwind.config.js for custom themes, colors, and design tokens.
- **State Management:**
  - **Local Component State:** useState, useReducer
  - **Global Client State:** React Context API for simple global states (e.g., theme, authentication status). For more complex scenarios, consider Zustand as an example.
  - **Server State/Data Fetching:** React Query (TanStack Query v5.x.x) for data fetching, caching, invalidation, and synchronization.
- **Form Handling:** React Hook Form (v7.x.x) for performance and flexible form management.

3.2. Project Structure

- **Root Layout:** Utilize Next.js 15+ App Router for structured routing (/{slug}).
- **Component Organization:**
  - /builder/elements: Reusable UI components (e.g., Button.tsx, Modal.tsx)

  - /hooks: Custom React hooks.
  - /utils: Utility functions.
  - /lib: External library integrations or API clients.
  - /styles: Global styles or Tailwind base configuration.
- **Naming Conventions:** PascalCase for components, camelCase for variables/functions, kebab-case for CSS classes (handled by Tailwind).

3.3. Styling & UI Components

- **Atomic Design Principles:** Encourage building components from atoms (buttons, inputs) to molecules (forms) to organisms (sections).
- **Responsiveness:** Mobile-first approach for all UI components using Tailwind's responsive utilities.
- **Design System:** Gradually build a reusable component library within the /elements directory, enforcing consistency in look and feel.
- **Accessibility (A11y):** Prioritize semantic HTML, ARIA attributes when necessary, and keyboard navigation.

3.4. State Management

- **Data Flow:** Unidirectional data flow (React's philosophy).
- **Server State (React Query):**
  - Use useQuery for GET requests, useMutation for POST/PUT/DELETE requests.
  - Implement query invalidation strategies (e.g., queryClient.invalidateQueries) to keep UI in sync with backend changes.
  - Utilize query caching for performance.
- **Avoid Prop Drilling:** Use Context API or global state managers for props needed by deeply nested components.

3.5. Data Fetching & Caching

- **API Client:** Use a lightweight HTTP client like axios or native fetch API wrapped in custom hooks/functions for consistent error handling and request configuration.
- **Error Handling:** Implement robust error boundaries and display user-friendly error messages for API failures.
- **Loading States:** Provide clear loading indicators for asynchronous operations.

3.6. Error Handling & UI Feedback

- **Error Boundaries:** Use React Error Boundaries for catching UI rendering errors.
- **Toast Notifications:** Implement a consistent notification system (e.g., react-hot-toast) for success, error, and warning messages.
- **Form Validation:** Client-side validation using React Hook Form, complemented by server-side validation messages.

3.7. Performance Optimization

- **Code Splitting:** Leverage Next.js automatic code splitting.
- **Image Optimization:** Use next/image component for optimized images.
- **Lazy Loading:** Lazy load components that are not immediately visible.
- **Data Fetching:** Optimize data fetching (e.g., select only necessary fields, use pagination where appropriate).

3.8. Accessibility (A11y)

- **WCAG Guidelines:** Strive for WCAG 2.1 AA compliance.
- **Semantic HTML:** Use appropriate HTML5 semantic elements.
- **Keyboard Navigation:** Ensure all interactive elements are keyboard accessible.
- **ARIA Attributes:** Use ARIA roles and attributes judiciously when semantic HTML isn't enough.
- **Contrast Ratios:** Ensure sufficient color contrast.

3.9. Testing Strategy

- **Unit Tests:** Jest + React Testing Library for individual components and utility functions. Aim for high component test coverage.
- **Integration Tests:** Test interactions between multiple components or with mock APIs.
- **End-to-End (E2E) Tests:** Cypress or Playwright for critical user flows (e.g., login, create page, save page). Initial focus on core flows.
- **Linting & Type Checking:** Enforce strict ESLint rules (Airbnb/Standard config adapted for Next.js/TS) and TypeScript checks (tsconfig.json). Pre-commit hooks for linting.

3.10. Build & Deployment

- **Static Assets:** Leverage Next.js static asset serving for optimal performance.
- **Build Process:** Use next build command.
- **Environment Variables:** All configurations should be managed via environment variables (.env.local, .env.production).

4. Backend Technical Requirements (Laravel)

4.1. Core Stack & Libraries

- **Framework:** Laravel (LTS version, e.g., v12.x)
- **Language:** PHP (compatible with Laravel LTS, e.g., v8.3+)
- **Database:** MySQL (v8.0+)
- **ORM:** Eloquent
- **Package Manager:** Composer (v2.x)
- **Authentication:** Laravel Sanctum (for SPA API token authentication)
- **Request Validation:** Laravel's built-in Validation.
- **API Resources:** Laravel API Resources for transforming Eloquent models into JSON responses.
- **Multi-lingual:** yes, recommended mcamara Laravel-localization package.

4.2. Project Structure & Design Patterns

- **Modular Architecture:** Organize code by feature like the existing structure, feature targeted directory that contains (Models, Views, Controllers, etc…).
- **Service Layer (Optional but Recommended):** For complex business logic, introduce a service layer (app/Services) between controllers and models.
- **Repository Pattern (Optional):** Consider a repository pattern for abstracting database interactions if complexity demands it.
- **SOLID Principles:** Adhere to SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion).
- **Dependency Injection:** Utilize Laravel's IoC container for dependency injection.

4.3. API Design Principles

- **RESTful API:** Design APIs following RESTful conventions (resources, HTTP verbs, status codes).
  - GET /api/pages - Retrieve all pages.
  - GET /api/pages/{slug} - Retrieve a specific page.
  - POST /api/pages - Create a new page.
  - PUT /api/pages/{slug} - Update an existing page.
  - DELETE /api/pages/{slug} - Delete a page.
- **Stateless:** API should be stateless.
- **Content Type:** application/json for requests and responses.
- **API Versioning:** Implement API versioning (e.g., /api/v1/...) from the start to allow for future changes without breaking existing clients.
- **Pagination & Filtering:** Implement pagination, sorting, and filtering for collection endpoints from the beginning.

4.4. Database Interaction

- **Migrations:** Use Laravel Migrations for all database schema changes. Rollback capability should be verified.
- **Seeders:** Use Seeders for populating initial data (e.g., admin users, default templates).
- **Eloquent:** Utilize Eloquent for database interactions. Avoid raw SQL queries unless absolutely necessary (and justified).
- **N+1 Problem:** Address N+1 query issues using eager loading (with()).
- **Transactions:** Use database transactions for atomic operations that involve multiple writes.

4.5. Authentication & Authorization

- **Authentication:** Laravel Sanctum for API token-based authentication for the Next.js SPA.

- **Authorization:** Implement Laravel Gates or Policies for granular access control (e.g., only authenticated admins can create/update/delete pages).

4.6. Validation

- **Server-Side Validation:** All incoming request data **must** be validated on the server using Laravel's Validation features.
- **Custom Validation Rules:** Create custom validation rules for complex validation logic.

4.7. Error Handling & Logging

- **Standardized Error Responses:** Implement a consistent JSON error response structure (e.g., {"message": "...", "errors": {...}, "code": ...}).
- **HTTP Status Codes:** Return appropriate HTTP status codes (e.g., 200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Unprocessable Entity, 500 Internal Server Error).
- **Logging:** Use Laravel's logging facilities (Monolog). Log errors, critical events, and suspicious activities. Configure logging channels (e.g., daily or stack).

4.8. Security Best Practices

- **Input Sanitization:** Beyond validation, sanitize user inputs to prevent XSS, SQL injection (Eloquent helps here).
- **CSRF Protection:** Laravel handles this by default for web routes; for API, ensure proper Sanctum token handling.
- **Rate Limiting:** Implement rate limiting on API endpoints (especially login) to prevent brute-force attacks.
- **CORS:** Explicitly configure CORS policies in Laravel (config/cors.php) to allow requests only from the frontend domain(s).
- **Sensitive Data:** Never expose sensitive data in API responses or logs. Encrypt sensitive data at rest.

- **Dependency Updates:** Regularly update Composer dependencies to mitigate security vulnerabilities.
- **Environment Variables:** All secrets and configuration should be managed via .env files and never committed to Git.

4.9. Performance & Scalability Considerations

- **Caching:** Utilize Laravel's caching mechanisms (e.g., Redis, file cache) for frequently accessed, static data.
- **Queues:** For long-running tasks (e.g., complex page rendering, image processing), use Laravel Queues.
- **Database Optimization:** Optimize queries, ensure proper indexing, and avoid N+1 problems.
- **Resource Optimization:** Efficient use of server resources.

4.10. Testing Strategy

- **Unit Tests:** PHPUnit for individual classes, methods, and functions.
- **Feature Tests:** PHPUnit for testing API endpoints and application features by making HTTP requests against the application.
- **Database Testing:** Ensure tests can interact with a clean test database.
- **Static Analysis:** PHPStan (level 5+) for static code analysis.
- **Code Style:** Laravel Pint for automatic code style enforcement.

4.11. Deployment

- **Server:** Apache with PHP-FPM.
- **Process Manager:** Supervisor for queue workers and other long-running processes.
- **Environment:** Production, Development. Each environment must be clearly defined and segregated.
- **Composer Install:** Run composer install --no-dev --optimize-autoloader in production.

- **Cache Clear:** Optimize Laravel configuration and routes caching (php artisan optimize).
- **Database Migrations:** Run migrations and seeders (php artisan migrate --force).

5. Database Technical Requirements (MySQL)

5.1. Schema Design Principles

- **Normalization:** Aim for 3NF (Third Normal Form) to minimize data redundancy, unless denormalization is justified for performance reasons (with documentation).
- **Relationships:** Define clear relationships between tables (One-to-One, One-to-Many, Many-to-Many) using foreign keys.
- **Data Types:** Use appropriate data types for columns (e.g., VARCHAR with appropriate length, INT, BIGINT, BOOLEAN, JSON, TEXT).
- **JSON Column Usage:** Utilize MySQL's JSON data type for storing page content structure, elements, and their properties as a single blob. This offers flexibility but requires careful indexing strategies if querying within JSON.

5.2. Naming Conventions

- **Tables:** Plural, snake_case (e.g., pages, page_elements).
- **Columns:** Singular, snake_case (e.g., id, user_id, created_at).
- **Primary Keys:** id (auto-incrementing BIGINT UNSIGNED).
- **Foreign Keys:** related_table_id (e.g., page_id).

5.3. Indexing Strategy

- **Primary Keys:** Automatically indexed.

- **Foreign Keys:** Should be indexed.
- **Frequently Queried Columns:** Index columns used in WHERE clauses, JOIN conditions, and ORDER BY clauses.
- **JSON Columns:** Consider using functional indexes (MySQL 8+) on specific paths within JSON columns if querying within JSON is common and performance critical.

5.4. Backup & Recovery

- **Regular Backups:** Implement automated daily backups of the database.
- **Recovery Plan:** Define a clear disaster recovery plan for restoring the database from backups.

6. Cross-Cutting Concerns

6.1. Environment Management

- **.env Files:** Use .env files for environment-specific configurations (database credentials, API keys, app URL).
- **Environment Variables:** Access sensitive configurations via environment variables (process.env.VAR_NAME in Next.js, env('VAR_NAME') in Laravel).

6.2. CI/CD Pipeline (Initial Setup)

- **Continuous Integration (CI):**
  - Automated execution of linting, formatting, and unit/feature tests on every push to a feature branch and every Pull Request.
  - Build artifacts generation (Frontend: next build, Backend: composer install --no-dev).

- **Continuous Deployment (CD) - Staging:**
    - Automated deployment to a staging environment upon successful merge to main.
- **Tools (Suggested):** GitHub Actions, GitLab CI/CD, or Jenkins.

6.3. Monitoring & Logging (Application Level)

- **Backend Logs:** Configure Laravel logs to be stored securely and rotated. Consider centralized logging solutions (e.g., ELK Stack, Datadog) for production.
- **Frontend Logs:** Client-side error logging to a service (e.g., Sentry) for production.
- **Performance Monitoring:** Basic monitoring of application performance (response times, error rates) in production.

Thanks for reading…