Dynamic Website Project Documentation

# Next.js Frontend + Strapi CMS Backend

# Table of Contents

1. Project Overview

2. Technology Stack

3. Project Structure

4. Data Flow & Architecture

5. Local Development Setup

6. Content Management Workflow

7. Dynamic Content System

8. API Integration

9. Caching & Performance

10. Troubleshooting

# 1. Project Overview

This is a dynamic website project built with Next.js as the frontend framework and Strapi v5 as the headless CMS backend. The project implements a sophisticated content management system where content creators can manage website content through Strapi's admin panel, and changes are automatically reflected on the Next.js frontend with intelligent caching and revalidation.

## Key Features

* Dynamic Content Management: Content is managed through Strapi and dynamically rendered on the frontend
* Component-Based Architecture: Modular, reusable components for different content types
* Real-time Updates: Content changes in Strapi trigger automatic frontend updates via webhook revalidation
* SEO Optimization: Dynamic meta tags and SEO components
* Performance Optimized: Intelligent caching with ISR (Incremental Static Regeneration)
* Responsive Design: Modern, mobile-first design with Tailwind CSS

# 2. Technology Stack

## Frontend (Next.js)

* Framework: Next.js 15.3.4 (App Router)
* React Version: 19.0.0
* Styling: Tailwind CSS 4.x
* Animations: GSAP 3.13.0
* Content Rendering: Strapi Blocks React Renderer
* Image Optimization: Next.js Image component
* State Management: React built-in state (useState, useCallback)

## Backend (Strapi)

* CMS: Strapi v5.16.1
* Database: PostgreSQL (configurable)
* Plugins:
* Deep Populate plugin for complex data fetching
* SEO plugin for meta management
* Users & Permissions for authentication
* Nodemailer for email functionality

## Additional Technologies

* Database: PostgreSQL (production), SQLite (development)
* Authentication: Strapi JWT tokens
* Image Hosting: Multiple domains supported (localhost, staging, production)
* Package Management: npm
* Version Control: Git

# 3. Project Structure

## Frontend Directory Structure (/Frontend)

Frontend/  
├── app/ # Next.js App Router  
│ ├── globals.css # Global styles and Tailwind imports  
│ ├── layout.js # Root layout with header/footer  
│ ├── page.js # Home page  
│ ├── not-found.js # 404 error page  
│ ├── blocksMap.js # Component mapping for dynamic content  
│ ├── api/ # API routes  
│ │ ├── revalidate/ # Webhook revalidation endpoint  
│ │ └── sitemap/ # Dynamic sitemap generation  
│ ├── who-we-are/ # About page  
│ ├── what-we-do/ # Services page  
│ ├── article/ # Blog/article pages  
│ ├── contact/ # Contact page  
│ ├── privacy-policy/ # Privacy policy page  
│ ├── strategic-partnership/ # Partnership page  
│ └── think-forward/ # Blog listing page  
├── components/ # Reusable React components  
│ ├── Header/ # Navigation components  
│ ├── Footer/ # Footer components  
│ ├── Home/ # Homepage-specific components  
│ ├── WhoWeAre/ # About page components  
│ ├── WhatWeDo/ # Services page components  
│ ├── Contact/ # Contact form components  
│ ├── Article/ # Blog/article components  
│ └── [SharedComponents] # Button, Container, inputs, etc.  
├── lib/ # Utility functions and configurations  
│ ├── strapiApi.js # Strapi API fetch functions  
│ ├── populateMap.js # Data population configurations  
│ ├── parseStrapiRichText.js # Rich text content parser  
│ └── data.js # Static data and configurations  
├── public/ # Static assets  
├── package.json # Dependencies and scripts  
├── next.config.mjs # Next.js configuration  
├── tailwind.config.js # Tailwind CSS configuration  
└── jsconfig.json # JavaScript configuration

## Backend Directory Structure (/Backend\_strapi)

Backend\_strapi/  
├── src/ # Source code  
│ ├── api/ # Content types and API endpoints  
│ │ ├── home/ # Home page content type  
│ │ ├── article/ # Blog article content type  
│ │ ├── who-we-are/ # About page content type  
│ │ ├── what-we-do/ # Services page content type  
│ │ ├── contact/ # Contact page content type  
│ │ ├── page/ # Dynamic page content type  
│ │ ├── global/ # Global site settings  
│ │ └── [other-types]/ # Additional content types  
│ ├── components/ # Reusable content components  
│ │ ├── layout/ # Layout components (hero, content-block, etc.)  
│ │ ├── elements/ # UI elements (buttons, cards, etc.)  
│ │ └── shared/ # Shared components (SEO, etc.)  
│ ├── extensions/ # Strapi extensions  
│ └── admin/ # Admin panel customizations  
├── config/ # Configuration files  
│ ├── database.ts # Database configuration  
│ ├── server.ts # Server configuration  
│ ├── admin.ts # Admin panel configuration  
│ ├── api.ts # API configuration  
│ ├── middlewares.ts # Middleware configuration  
│ └── plugins.ts # Plugin configuration  
├── types/ # TypeScript type definitions  
├── database/ # Database files (SQLite in development)  
├── package.json # Dependencies and scripts  
├── tsconfig.json # TypeScript configuration  
└── README.md # Strapi documentation

# 4. Data Flow & Architecture

## High-Level Architecture

Content Creator → Strapi Admin Panel → PostgreSQL Database  
 ↓  
 Strapi API Endpoints  
 ↓  
Next.js Frontend ← API Calls ← Dynamic Component Rendering  
 ↓  
 End User Website

## Detailed Data Flow

1. Content Creation:

* Content creators log into Strapi admin panel
* Create/edit content using dynamic zones and components
* Publish content (draft/publish workflow)

1. API Data Retrieval:

* Next.js pages call fetchStrapi() function
* API requests include populate parameters for related data
* Authentication via Bearer token for secure endpoints

1. Dynamic Rendering:

* Content is structured as dynamic zones with component types
* blocksMap.js maps component types to React components
* Components receive data props and render accordingly

1. Cache Management:

* Next.js ISR provides automatic caching (3600s revalidation)
* Strapi webhooks trigger immediate cache invalidation
* Tag-based revalidation for granular cache control

# 5. Local Development Setup

## Prerequisites

* Node.js (18.0.0 to 22.x.x)
* npm (6.0.0+)
* PostgreSQL (recommended) or SQLite (development)
* Git

## Backend Setup (Strapi)

1. Navigate to backend directory:

cd Backend\_strapi

1. Install dependencies:

npm install

1. Environment Configuration:

Create a .env file in the Backend\_strapi directory:

# Database Configuration  
DATABASE\_CLIENT=postgres  
DATABASE\_HOST=localhost  
DATABASE\_PORT=5432  
DATABASE\_NAME=strapi\_db  
DATABASE\_USERNAME=strapi\_user  
DATABASE\_PASSWORD=your\_password  
DATABASE\_SSL=false  
  
# Server Configuration  
HOST=0.0.0.0  
PORT=1337  
APP\_KEYS=your\_app\_keys\_here  
  
# JWT Configuration  
JWT\_SECRET=your\_jwt\_secret  
API\_TOKEN\_SALT=your\_api\_token\_salt  
ADMIN\_JWT\_SECRET=your\_admin\_jwt\_secret  
  
# Email Configuration (optional)  
SMTP\_HOST=your\_smtp\_host  
SMTP\_PORT=587  
SMTP\_USERNAME=your\_email  
SMTP\_PASSWORD=your\_email\_password

1. Start development server:

npm run develop

1. Create admin user:

* Navigate to http://localhost:1337/admin
* Create your first admin user
* Configure content types and create sample content

## Frontend Setup (Next.js)

1. Navigate to frontend directory:

cd Frontend

1. Install dependencies:

npm install

1. Environment Configuration:

Create a .env.local file in the Frontend directory:

# Strapi Configuration  
NEXT\_PUBLIC\_STRAPI\_API\_URL=http://localhost:1337  
STRAPI\_API\_TOKEN=your\_strapi\_api\_token  
  
# Revalidation Secret  
REVALIDATE\_SECRET=your\_revalidation\_secret

1. Start development server:

npm run dev

1. Access the application:

* Frontend: http://localhost:3000
* Strapi Admin: http://localhost:1337/admin

## Production Deployment

### Frontend Deployment

cd Frontend  
npm run build  
npm start

### Backend Deployment

cd Backend\_strapi  
npm run build  
npm start

# 6. Content Management Workflow

## Creating Content in Strapi

1. Access Admin Panel:

* Navigate to Strapi admin (http://localhost:1337/admin)
* Log in with admin credentials

1. Content Types Available:

* Home: Single type for homepage content
* Article: Collection type for blog posts
* Page: Collection type for service pages
* Who We Are: Single type for about page
* What We Do: Single type for services overview
* Contact: Single type for contact page
* Global: Single type for site-wide settings

1. Dynamic Zone Components:

Each content type uses dynamic zones with these components:

* Hero: Main page header with title and CTA
* Content Block: Text content with optional background
* Feature Card: Highlight cards with icons/images
* Feature Item: List items with descriptions
* Step Item: Process steps or numbered items
* Target Audience Section: Service offerings grid

1. Content Creation Process:

* Select content type from sidebar
* Add components to dynamic zone
* Fill in component fields (text, images, buttons)
* Configure SEO settings
* Save as draft or publish immediately

## SEO Management

Each content type includes SEO components:

* Meta Title: Page title for search engines
* Meta Description: Page description for search results
* Keywords: Relevant keywords for SEO
* Open Graph: Social media sharing configuration
* Canonical URL: Preferred URL for search engines

# 7. Dynamic Content System

## How Dynamic Zones Work

Dynamic zones in Strapi allow content creators to build pages by combining different component types. The frontend renders these components dynamically based on their type.

### Example: Home Page Structure

Strapi returns this structure:

{  
 "Home": [  
 {  
 "\_\_component": "layout.hero",  
 "id": 1,  
 "text": "Welcome to Our Website",  
 "smallText": "Leading solutions provider",  
 "Button": { "text": "Learn More", "url": "/about" }  
 },  
 {  
 "\_\_component": "layout.content-block",  
 "id": 2,  
 "leftText": "Our Services",  
 "rightText": "We provide comprehensive solutions...",  
 "Background": false  
 }  
 ]  
}

### Frontend Rendering

HomeClient.js processes this data:

{blocks.map((block) => {  
 switch (block.\_\_component) {  
 case "layout.hero":  
 return <HomeHero key={block.id} data={block} />;  
 case "layout.content-block":  
 return <ServicesBlock key={block.id} data={block} />;  
 // ... other components  
 }  
})}

## Rich Text Processing

The project uses Strapi's blocks format for rich text content:

// parseStrapiRichText.js  
export function parseStrapiRichText(textArray) {  
 if (!Array.isArray(textArray)) return [];  
 const parts = [];  
 textArray.forEach((paragraph) => {  
 paragraph.children.forEach((child) => {  
 if (child.code) {  
 parts.push({ span: true, text: child.text });  
 } else {  
 parts.push({ text: child.text });  
 }  
 });  
 parts.push({ br: true });  
 });  
 return parts;  
}

# 8. API Integration

## Strapi API Functions

The strapiApi.js file provides the main interface between frontend and backend:

export async function fetchStrapi(endpoint, { populate, tag, revalidate } = {}) {  
 const queryString = populate  
 ? "?" + qs.stringify({ populate }, { encodeValuesOnly: true })  
 : "";  
  
 const res = await fetch(  
 `${process.env.NEXT\_PUBLIC\_STRAPI\_API\_URL}/api/${endpoint}${queryString}`,  
 {  
 headers: {  
 Authorization: `Bearer ${process.env.STRAPI\_API\_TOKEN}`,  
 },  
 next: {  
 tags: tag ? [tag] : undefined,  
 revalidate: revalidate || undefined,  
 },  
 }  
 );  
  
 return res.ok ? (await res.json()).data : null;  
}

## Population Strategies

Content relationships are populated using the populate parameter:

// Simple population  
const homeData = await fetchStrapi("home", {  
 populate: "\*"  
});  
  
// Complex population for nested relationships  
const complexData = await fetchStrapi("article", {  
 populate: {  
 SEO: { populate: "\*" },  
 author: { populate: "\*" },  
 categories: { populate: "\*" }  
 }  
});

## API Endpoints

Common endpoints used in the project:

* GET /api/home - Homepage content
* GET /api/articles - Blog articles list
* GET /api/articles/:slug - Individual article
* GET /api/who-we-are - About page content
* GET /api/what-we-do - Services content
* GET /api/global - Site-wide settings

# 9. Caching & Performance

## ISR (Incremental Static Regeneration)

Next.js pages use ISR for optimal performance:

// Automatic revalidation every hour  
export const revalidate = 3600;  
  
// Or per-request revalidation  
const data = await fetchStrapi("home", {  
 populate: "\*",  
 revalidate: 3600  
});

## Webhook-Based Revalidation

When content is updated in Strapi, webhooks trigger immediate cache invalidation:

// app/api/revalidate/route.js  
export async function POST(request) {  
 const { model, entry } = await request.json();  
   
 switch (model) {  
 case 'home':  
 revalidatePath('/');  
 break;  
 case 'article':  
 const slug = entry?.slug;  
 revalidatePath('/think-forward');  
 if (slug) revalidatePath(`/article/${slug}`);  
 break;  
 // ... other content types  
 }  
   
 return NextResponse.json({ revalidated: true });  
}

## Tag-Based Caching

Granular cache control using Next.js cache tags:

// Fetch with cache tag  
const globalData = await fetchStrapi("global", {  
 populate: "\*",  
 tag: "global"  
});  
  
// Invalidate specific tag  
revalidateTag('global');

# 10. Troubleshooting

## Common Issues

### 1. API Connection Issues

Problem: Frontend can't connect to Strapi

Solutions:

* Check NEXT\_PUBLIC\_STRAPI\_API\_URL in .env.local
* Ensure Strapi is running on correct port (1337)
* Verify API token is correct and has proper permissions

### 2. Content Not Updating

Problem: Changes in Strapi don't appear on frontend

Solutions:

* Check if content is published (not draft)
* Verify webhook revalidation is configured
* Manually clear Next.js cache: delete .next folder and restart
* Check browser cache (hard refresh with Ctrl+F5)

### 3. Image Loading Issues

Problem: Images from Strapi not displaying

Solutions:

* Verify image domains in next.config.mjs
* Check image permissions in Strapi
* Ensure proper image upload settings

### 4. Build Errors

Problem: Next.js build fails

Solutions:

* Check for TypeScript errors (set ignoreBuildErrors: true temporarily)
* Ensure all environment variables are set
* Verify all imports are correct

### 5. Database Connection Issues

Problem: Strapi can't connect to database

Solutions:

* Verify database credentials in .env
* Ensure PostgreSQL is running
* Check database exists and user has permissions

## Environment Variables Checklist

### Frontend (.env.local)

NEXT\_PUBLIC\_STRAPI\_API\_URL=http://localhost:1337  
STRAPI\_API\_TOKEN=your\_api\_token  
REVALIDATE\_SECRET=your\_secret

### Backend (.env)

DATABASE\_CLIENT=postgres  
DATABASE\_HOST=localhost  
DATABASE\_PORT=5432  
DATABASE\_NAME=strapi\_db  
DATABASE\_USERNAME=strapi\_user  
DATABASE\_PASSWORD=your\_password  
JWT\_SECRET=your\_jwt\_secret  
API\_TOKEN\_SALT=your\_salt  
ADMIN\_JWT\_SECRET=your\_admin\_secret

# Conclusion

This documentation provides a comprehensive guide to understanding, setting up, and maintaining the Next.js + Strapi dynamic website project. The architecture enables content creators to manage website content efficiently while providing developers with a flexible, performant, and maintainable codebase.

For additional support or questions, refer to the official documentation:

* Next.js Documentation: https://nextjs.org/docs
* Strapi Documentation: https://docs.strapi.io
* React Documentation: https://react.dev