PHOTOGRAPHY PORTFOLIO

FUNCTIONAL SPECIFICATIONS - 16/8/2025

GLOSSARY

Term	Definition
SPA	Single Page Application - A web application that loads a single HTML page and dynamically updates content
API	Application Programming Interface - Set of protocols for building and integrating application software
JSON	JavaScript Object Notation - Lightweight data storing format
Lazy Loading	Delays loading of non-critical resources until they are needed
Responsive Design	Dynamic approach that allows websites to be displayed on different screen sizes.
Kubernetes	Container orchestration platform for automating deployment and scaling
Actix-Web	Rust web framework for building web applications
React	JavaScript library for building user interfaces
Vite	Fast build tool for modern web projects

1. INTRODUCTION

1.1 Purpose

The objective of this document is to offer a detailed functional specification for my photographic portfolio web application. This portfolio system is intended to promote photographic work through a minimalist, high-performance online interface that emphasises visual presentation and user experience.

This portfolio aims to:

- Provide a professional online presence for showcasing photographic work
- Deliver an optimal viewing experience across all devices and screen sizes
- Ensure fast load times and smooth navigation between photographs
- Maintain simplicity in both user interface and content management

1.2 Scope

This functional specification describes the entire photographic portfolio web application, including both the user-facing frontend interface and the backend system that serves content and manages data.

1.2.1 In Scope:

- Display system for three photography categories: People, Landscapes, and Abstract
- Individual photo viewing with detailed information (title, description, date)
- Category-based filtering and navigation
- Responsive design for desktop, tablet, and mobile devices
- Static content serving with JSON-based data management
- Image optimization and lazy loading for performance
- Single-service deployment architecture
- Local development environment using Kubernetes
- Data including camera model, lens, ISO, aperture, shutter speed, and focal length will be displayed when viewing singular photos.

1.2.2 Out of Scope:

- User authentication or login functionality
- Photo upload interface (content managed via JSON file)
- Comments or user interaction features.

2. USE CASES

2.0 Use Case Model

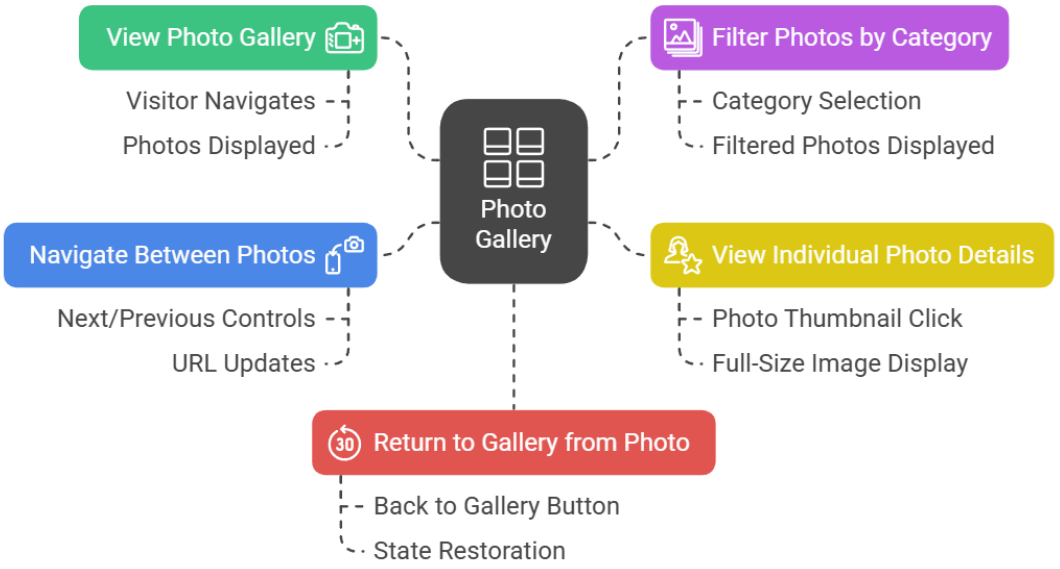

Figure 1 - Use Case Model

View Photo Gallery

- A visitor navigates to the gallery.
- The system displays a grid of available photos.

Filter Photos by Category

- The user selects a category (e.g., people, landscapes).
- Only matching photos are displayed.

View Individual Photo Details

- The user clicks a photo thumbnail.
- A full-size image and its metadata are shown.

Navigate Between Photos

- The user moves to the next/previous photo.
- The system updates the displayed image and URL.

Return to Gallery from Photo

- The user clicks a “back to gallery” button.
- The gallery view is restored with the previous filter/state intact.

2.1 Use Case 1 - View Photo Gallery

Use Case	View Photo Gallery
Description	Visitors can browse through the complete collection of photographs
User Story	As a visitor, I want to view all available photographs in the portfolio so that I can see the photographer's complete body of work
Precondition	• Visitor has navigated to the gallery page • Photos data is loaded in the application
Postcondition	All photos are displayed and accessible to the visitor
Main Flow	1. Visitor navigates to `/gallery` route 2. Frontend already has photos data from initial app load 3. System displays all photos in a responsive layout 4. Images lazy-load as visitor scrolls 5. Visitor can continue scrolling to see all photos
Alternative flow	• If no photos exist, display "No photos available" message • If images fail to load, show placeholder with retry option

2.1 Use Case 2 - Filter Photos by Category

Use Case	Filter Photos by Category
Description	Visitor filters the photo gallery to view only photos from a specific category (People, Landscapes, or Abstract)
User Story	As a visitor, I want to filter photos by category so that I can view only the type of photography I'm interested in.
Precondition	• The visitor is on the gallery page. • Photos are loaded and displayed.
Postcondition	Only photos from selected category are displayed
Main Flow	1. The visitor clicks on a category filter button (People/Landscapes/Abstract). 2. Frontend filters the stored photos array by selected category. 3. Page instantly updates to show only filtered photos. 4. Visitor can select a different category or "All" to reset
Alternative flow	• If category has no photos, display "No photos in this category" Error • Visitors can clear filters to return to all photos.

2.1 Use Case 3 - View Individual Photo Details

Use Case	View Individual Photo Details
Description	Visitor clicks on a photo thumbnail to view the full-size image with detailed information
User Story	As a visitor, I want to view a photo in full size with its details so that I can learn more about it
Precondition	• Visitor is viewing the gallery • Photo data is loaded in React Context
Postcondition	Full photo and details are displayed
Main Flow	1. Visitor clicks on a photo thumbnail in gallery 2. React Router navigates to `/photo/{id}` route 3. PhotoDetail component retrieves photo data from context using ID 4. Full-size image loads from `/images/{filename}` endpoint 5. Title, description, and date are displayed alongside image
Alternative flow	• If ID doesn't exist, show 404 message with return to gallery option

2.1 Use Case 4 - Navigate Between Photos

Use Case	Navigate Between Photos
Description	Visitor clicks on a photo thumbnail to view the full-size image with detailed information including camera settings
User Story	As a visitor viewing a photo, I want to navigate to the next or previous photo without returning to the gallery
Precondition	• Visitor is on a photo detail page • Multiple photos exist in the collection
Postcondition	Visitor is viewing the adjacent photo with updated URL
Main Flow	1. Visitor is viewing a photo at `/photo/{id}` 2. Visitor clicks "Next" arrow or presses right arrow key 3. Frontend finds next photo in the photos array 4. React Router updates URL to `/photo/{nextId}` 5. New photo and details are displayed 6. Navigation arrows update based on position
Alternative flow	• At last photo, "Next" loops to the first photo • At first photo, "Previous" loops to the last photo

2.1 Use Case 5 - Return to Gallery from Photo

Use Case	Return to Gallery from Photo
Description	Visitor returns to the gallery view from an individual photo, maintaining their previous filter state
User Story	As a visitor viewing a photo, I want to return to the gallery where I left off so that I can continue browsing
Precondition	• Visitor is viewing an individual photo • Visitor previously came from gallery (with or without filter)
Postcondition	Visitor is back in gallery with previous viewing context maintained
Main Flow	1. Visitor clicks "Back to Gallery" button or browser back 2. React Router navigates back to `/gallery` or `/gallery?category={filter}` 3. Gallery displays with previous filter state intact 4. Scroll position is restored 5. Visitor continues browsing from where they left off
Alternative flow	• If no previous gallery state, default to showing all photos

3. INTERFACES

3.1 System Interfaces

3.1.1 Backend-API Interface

The Backend API Interface acts as the photography portfolio's primary data source and content delivery system. Built with Rust and Actix-Web, it has three core responsibilities: offering photo metadata via JSON APIs, delivering image files from the disc, and delivering the assembled React application to browsers. The API adheres to RESTful principles, with a read-only architecture that reduces complexity while maximising efficiency. At launch, all photo metadata is imported into memory from the photos.json file, allowing for quick and instantaneous responses.

Endpoints

Endpoint	Description	Method	Response Type
/api/photos	Retrieve all photo metadata	Get Request	JSON Array
/images/{filename}	Serve image files	Get Request	Image Binary

3.1.2 Frontend Web Interface

The Frontend Web Interface provides a visually focused experience for viewing the portfolio. It is built as a React Single Page Application with TypeScript to allow for smooth, immediate navigating between multiple views without the need for page refresh. The interface focuses on the photography itself, with a basic design that works effortlessly across devices.

Upon initial load, the application retrieves all photo metadata once and stores it in React Context, allowing for quick filtering and navigation without requiring subsequent server queries. The gallery view displays photographs in a responsive layout with lazy loading for best speed, whereas the individual photo view allows for full-screen viewing of each image and its accompanying details. The UI is stylised with Tailwind CSS, which ensures consistent and maintainable design principles.

Main Components Include:

- **Navigation Bar**: Persistent navigation across all pages
- **Gallery View**: Grid layout for photo thumbnails
- **Photo Detail View**: Full-size photo display with metadata
- **Category Filters**: Interactive buttons for filtering photos
- **Footer**: Contact links and copyright information

Routes Include:

Route	Component	Purpose
/	Home	Home/landing page
/gallery	Gallery	Photo grid with filtering
/gallery?category={type}	Gallery	Filtered photo view
/photo/{id}	PhotoDetail	Individual photo display
/contact	Contact	Contact Information

3.1.3 Kubernetes Configuration Interface

Kubernetes provides a production-like environment for local development, allowing us to test containerised applications before deployment. In this configuration, the backend and frontend run as distinct pods, simulating a microservices architecture that allows for independent development and debugging of each component. The backend pod runs the Rust API server, while the frontend pod runs the Vite development server.The local Kubernetes environment preserves isolation for a better development experience with greater debugging capabilities and faster iteration cycles, even while the production deployment for Railway merges both services into a single container.

3.2 Software Interfaces

3.2.1 Backend Software Stack

Rust, the foundation of the backend software stack, was selected for its outstanding performance, memory security, and small resource footprint. Actix-Web offers a strong, actor-based HTTP server framework that effectively manages several requests at once while preserving consistent performance under load. Non-blocking I/O operations are made possible by the Tokio async runtime, which enables the server to process several requests at once without incurring threading overhead.

Utilising Rust's zero-cost abstractions and compile-time guarantees, the architecture produces a backend that is fast to start and requires little memory In order to eliminate database latency and guarantee consistent quick response times for API queries, all photo metadata is deserialised once during startup using Serde and cached in memory using once_cell.

Backend components include:

Component	Purpose
Rust	Systems programming language
Actix-web	HTTP server framework
Tokio	Async runtime for concurrent operations

3.2.2 Frontend Software Stack

Modern web technologies are combined in the frontend software stack to create a user interface that is responsive and competent. The basis is React 18, which offers a component-based architecture with effective virtual DOM updates and integrated performance enhancements like parallel rendering and automated batching.

In addition to eliminating runtime mistakes, TypeScript also includes compile-time type checking. Vite is used for its rapid server start, and optimised production builds. Tailwind CSS offers a utility-first styling approach that keeps the final CSS bundle small while facilitating quick UI development with consistent design standards.

Frontend components include:

Component	**Purpose**
React	UI Component Library
Typescript	Type-safe Javascript
Vite	Build tool and dev server
Tailwind CSS	CSS framework

3.3 Communication Interfaces

3.3.1 Client-Server Communication

The client-server communication architecture takes a simplified, performance- and simplicity-optimized approach. The entire React application bundle is served by the Rust backend in response to the browser's initial request to the root URL when a user first views the portfolio. With a single API call to /api/v1/photos, the React application instantly retrieves all photo metadata after loading, saving it in React Context for the duration of the session.

By leveraging the cached data, this single fetch approach removes the need for duplicate API requests when users browse individual photographs, apply filters, and switch between gallery views. Only the loading of real image files, which are requested on-demand when they reach the viewport via lazy loading, requires further server requests.Since most interactions don't require server contact, this architecture minimises network overhead, lowers server load, and offers rapid user interface replies.The backend implements appropriate caching headers, serving images with long-term cache directives (one year) to leverage browser caching, while API responses use no-cache headers to ensure fresh data on page reload.

3.3.2 External Communication

Git Repository:

- Protocol: HTTPS/SSH
- Purpose: Source control and CI/CD trigger
- Flow: Push -> GitHub -> Railway webhook -> Deploy

Railway Deployment:

- Protocol: HTTPS
- Authentication: API token
- Deployment: Docker image via Dockerfile

I plan to host this site on Railway as it works great with github and docker and the free tier is excellent for smaller sites. The deployment uses a single production Dockerfile located at the repository root that combines both the frontend and backend into one optimized container. During the build process, the Dockerfile first compiles the React frontend into static assets, then builds the Rust backend, and finally creates a minimal runtime image where the Rust server serves both the API endpoints and the static React files.

This single-service architecture eliminates CORS complexity, reduces infrastructure costs, and simplifies deployment. When code is pushed to the main branch on GitHub, Railway automatically triggers a new deployment, building the combined container and replacing the running instance with zero-downtime deployment.

4. FUNCTIONAL REQUIREMENTS

4.1 View Photo Gallery

Inputs	Visitor navigates to /gallery
Processing	- Frontend fetches photo list from /api/photos. - Store JSON response in React Context. - Render gallery grid using filename, title, and category. - Images load directly from /images/{filename}.
Outputs	- Grid of photos with title + category. - Fallback message if empty.
Exception	If API returns empty list -> show "No photos available". If fetch fails -> show { "error": "Resource not found", "status": 404 }.

4.2 Filter Photos By Category

Inputs	Visitor selects category (e.g., people, landscapes, abstract).
Processing	- Update selectedCategory state when user selects a category. - Filter photos in state/context by category (or return all if "all"). - Render the filtered list in the gallery. - If no matches, show fallback message.
Outputs	Gallery updates to show only matching categories.
Exception	If no photos in category -> show "No photos in this category".

4.3 View individual Photo Details

Inputs	Visitor clicks a photo in gallery.
Processing	- React Router → /photo/{id}. - Match photo by id from Context. - Display filename, title, description, date, and cameraInfo (if available).
Outputs	- Full-size photo rendered from /images/{filename}. - Metadata details: title, description, date, and fields under cameraInfo.
Exception	If ID is not found -> return { "error": "Resource not found", "status": 404 }.

4.4 Navigate Between Photos

Inputs	Visitor uses next/previous arrows or keyboard.
Processing	- Determine adjacent id in photos array. - Router updates to /photo/{nextId} or /photo/{prevId}.
Outputs	Load and display new photos with all metadata.
Exception	- Last photo -> loop to first. - First photo -> loop to last.

4.5 Return to Gallery

Inputs	Visitor clicks “Back to Gallery” or browser back.
Processing	- Router navigates back to /gallery. - Restore filter + scroll state from Context.
Outputs	Gallery shown with previous state.
Exception	If no saved state → default to full gallery.