INFINITE SCROLL GALLERY

PROJECT OVERVIEW&OBJECTIVES:

Problem statement:

Traditional image galleries often require:

- **Pagination** (clicking next/previous page), which interrupts the flow.
- **Preloaded heavy image sets**, causing long loading times and poor performance.

Users today expect **smooth, endless scrolling** for media browsing, like in Instagram, Pinterest, and Unsplash. The challenge is to **design a lightweight gallery** that continuously loads content on demand without overwhelming the browser or requiring manual navigation.

KEY FEATURES:

1. **Endless Scrolling** – Automatically loads new images when the user reaches the bottom of the page, removing the need for pagination.
2. **Dynamic Image Loading** – Images are fetched and displayed on-demand, ensuring faster load times and efficient memory usage.
3. **Responsive Design** – Works smoothly across different screen sizes (desktop, tablet, mobile).
4. **Smooth User Experience** – Seamless browsing without interruptions like page reloads or clicks on “next/previous.”
5. **Lazy Loading Support** – Only loads images when they come into the user’s view, improving performance and reducing bandwidth usage.
6. **Customizable Layout** – Grid or masonry-style gallery arrangement for better visual appeal.
7. **Scalable** – Can handle a small or very large collection of images without slowing down.
8. **Interactive UI** – Option to add hover effects, image captions, or lightbox preview for better user engagement.

EXPECTED OUTCOME:

1. **Seamless Browsing Experience** – Users can continuously explore images without interruptions such as page reloads or navigation clicks.
2. **Improved Performance** – By dynamically and lazily loading images, the gallery ensures faster loading times and reduced memory/bandwidth usage.
3. **Responsive and Modern UI** – The gallery adapts to different devices and screen sizes, providing a smooth experience on desktop, tablet, and mobile.
4. **Scalability** – The application can handle a large collection of images without affecting user experience, making it suitable for real-world scenarios like social media feeds or e-commerce product listings.
5. **Practical Learning Output** – The project demonstrates knowledge of **HTML, CSS, and JavaScript**, particularly in event handling (scroll detection), DOM manipulation, and responsive web design.
6. **Engaging User Interaction** – With features like hover effects or image previews, users stay engaged while exploring the gallery.

TECHNOLOGY STACK & ENVIRONMENT SETUP:

NODE.JS:

Technology Stack

1. **Frontend**
    - **HTML5** → Structure of the gallery (image containers, layout).
    - **CSS3** → Styling, responsive design, grid/masonry layout.
    - **JavaScript (Vanilla JS/ES6)** → Scroll detection, dynamic content loading, lazy loading.

2. **Backend** (Optional if images are served dynamically)
    - **Node.js** → Backend runtime environment for serving image data.
    - **Express.js** → Web framework to create REST APIs for fetching images.
3. **Database** (Optional, if you want to store image URLs)
    - **MongoDB / JSON file / Static folder** → To store and retrieve image URLs or metadata.
4. **Tools & Packages**
    - **NPM (Node Package Manager)** → For managing project dependencies.
    - **Axios / Fetch API** → To request image data from the backend.
    - **Multer (Optional)** → If you want to add image upload functionality.

Environment Setup using Node.js

Install Node.js

- Download and install from Node.js official site.
- Verify installation:
- node -v

npm –v

Initialize the Project

mkdir infinite-scroll-gallery

cd infinite-scroll-gallery

npm init –y

Install Dependencies

bash

npm install express cors

Project Structure

1. infinite-scroll-gallery/
2. ├── backend/
3. | ├── server.js   # Node.js + Express server
4. | └── images.json   # (or DB for storing image URLs)
5. ├── frontend/
6. | ├── index.html
7. | ├── style.css
8. | └── script.js
9. ├── package.json
10. Setup Backend (server.js)
    Example code to serve image data:
11. const express = require("express");
12. const cors = require("cors");
13. const app = express();
14. const PORT = 5000;
15. app.use(cors());
16. // Sample image URLs
17. const images = [
18.   "https://picsum.photos/300/200?random=1",
19.   "https://picsum.photos/300/200?random=2",

```
20.       "https://picsum.photos/300/200?random=3"
21.     ];
22.     app.get("/api/images", (req, res) => {
23.       res.json(images);
24.     });
25.     app.listen(PORT, () => {
26.       console.log(`Server running on http://localhost:${PORT}`);
27.     });
```

28. Run the Server

```
29.     node backend/server.js
```

Visit → http://localhost:5000/api/images to see the image data.

30. Connect Frontend to Backend
In script.js, fetch images from backend dynamically:

```
31.     async function loadImages() {
32.       const res = await fetch("http://localhost:5000/api/images");
33.       const data = await res.json();
34.       const gallery = document.getElementById("gallery");
35.       data.forEach(url => {
36.         let img = document.createElement("img");
37.         img.src = url;
38.         gallery.appendChild(img);
39.       });
40.     }
41.     loadImages();
```

---

1. **Setup Backend (server.js)**
   Example code to serve image data:
2. const express = require("express");
3. const cors = require("cors");
4. const app = express();
5. const PORT = 5000
6. app.use(cors());
7. // Sample image URLs
8. const images = [
9. "https://picsum.photos/300/200?random=1",
10. "https://picsum.photos/300/200?random=2",
11. "https://picsum.photos/300/200?random=3"
12.];
13. app.get("/api/images", (req, res) => {
14. res.json(images);
15. })
16. app.listen(PORT, () => {
17. console.log(`Server running on http://localhost:${PORT}`);
18. });
19. **Run the Server**
20. node backend/server.js

Visit → http://localhost:5000/api/images to see the image data.

Connect Frontend to Backend

In script.js, fetch images from backend dynamically:

```
async function loadImages() {
  const res = await fetch("http://localhost:5000/api/images");
  const data = await res.json();
  const gallery = document.getElementById("gallery");
  data.forEach(url => {
    let img = document.createElement("img");
    img.src = url;
    gallery.appendChild(img);
  });
}
loadImages();
```

Frontend Framework Options for Infinite Scroll Gallery

Since this project is mainly **UI-driven (scroll, image rendering, responsiveness)**, the following frameworks can be used:

1. React.js (Recommended)

- **Why?**
  - Component-based (makes the gallery modular).
  - Easy state management (to store loaded images).
  - Smooth integration with APIs (fetching new images).
  - Libraries like **React Infinite Scroll Component** or **React Virtualized** make implementation simple.
- **Usage Example:**

  - Create a <Gallery /> component.
  - Detect scroll position with onScroll.
  - Fetch and render images dynamically using useState and useEffect.

---

2. Angular

- Strong structure with TypeScript.
- Built-in services and dependency injection make API handling clean.
- Good for larger, more complex projects (but may feel heavy for a mini-project).

---

3. Vue.js

- Lightweight and beginner-friendly.
- Supports two-way data binding for handling image lists easily.
- Can use Vue Infinite Loading plugin for scroll-based fetching.

---

4. Plain JavaScript with Bootstrap / Tailwind CSS

- If the project is small and you don't want the complexity of frameworks, you can simply:
  - Use **vanilla JavaScript** for scroll detection.
  - Use **Bootstrap** or **Tailwind CSS** for responsive layout and styling.
- This is enough for a **mini-project submission** while still looking modern.

Suggested Frontend Stack for Your Project

React.js + Tailwind CSS

- **React.js** → For dynamic component-based UI and API integration.
- **Tailwind CSS** → For quick, responsive, and modern design without writing heavy CSS.

Database Options

1. MongoDB (Recommended)

- NoSQL, document-based, great for flexible JSON-like structures.
- Perfect for storing image metadata like { url, title, description }.
- Easy to integrate with **Node.js + Express**.

Example Schema (Mongoose):

```
const mongoose = require("mongoose");


const ImageSchema = new mongoose.Schema({
  url: { type: String, required: true },
  title: { type: String },
  description: { type: String },
  uploadedAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("Image", ImageSchema);
```

---

2. MySQL / PostgreSQL

- Relational database (rows and tables).

- Good if you want structured data with relationships (e.g., users, albums, likes).
- Example table:

id	url	title	description	uploaded_at
1	https://picsum.photos/200	Nature 1	Forest view	2025-10-04 10:00
2	https://picsum.photos/201	Nature 2	Lake view	2025-10-04 10:10

---

3. Firebase Firestore (Cloud DB)

- NoSQL cloud database by Google.
- Easy for frontend-heavy projects (React, Vue, Angular).
- Auto-scales, great for mobile/web apps.

Example Firestore Document:

```
{
  "url": "https://firebasestorage.googleapis.com/.../image1.jpg",
  "title": "Mountain View",
  "description": "Snow covered mountains",
  "uploadedAt": "2025-10-04T10:00:00Z"
}
```

How It Works with Infinite Scroll

1. Frontend sends a request like:
   - GET /api/images?skip=20&limit=10
   - Skip = number of images already loaded
   - Limit = number of new images to load
2. Backend queries the database with **pagination** (skip + limit).

3. Database returns the next batch of image metadata.
4. Frontend appends those images into the gallery.

Tools for Infinite Scroll Gallery

◆ Frontend Tools

1. **HTML5** – To structure the gallery and containers.
2. **CSS3 / Tailwind CSS / Bootstrap** – For styling, responsive layouts, and grid/masonry effects.
3. **JavaScript (Vanilla JS or ES6)** – For scroll detection and DOM manipulation.
4. **Frontend Frameworks (Optional)** –
    - **React.js** → Component-based UI with libraries like react-infinite-scroll-component.
    - **Vue.js** → Lightweight and flexible with plugins like vue-infinite-loading.
    - **Angular** → Strongly structured, TypeScript support.

◆ Backend Tools

1. **Node.js** – Runtime environment for building the server.
2. **Express.js** – Web framework for handling API requests.
3. **Database** –
    - **MongoDB (with Mongoose)** → For storing image metadata.
    - **MySQL / PostgreSQL** → If you want relational structure.
    - **Firebase Firestore** → Cloud NoSQL DB for real-time syncing.

◆ Development & Build Tools

1. **NPM (Node Package Manager)** – To manage dependencies.
2. **Git & GitHub** – For version control and project collaboration.
3. **VS Code** – Popular IDE with extensions for Node.js and frontend frameworks.
4. **Postman / Thunder Client** – To test backend APIs (fetching image data in batches).

◆ Optional Enhancement Tools

1. **Cloudinary / Firebase Storage / AWS S3** – To host and optimize images.
2. **Webpack / Vite** – For bundling frontend code.
3. **ESLint / Prettier** – For maintaining clean and consistent code.
4. **Heroku / Vercel / Netlify** – For deploying the project online.

API Design & Data Model for Infinite Scroll Gallery

Data Model

The gallery stores **image metadata** (not the actual image file).

Example Schema (MongoDB / Mongoose):

```
const mongoose = require("mongoose");


const ImageSchema = new mongoose.Schema({
  url: { type: String, required: true },        // Image link
  title: { type: String },                      // Optional caption
  description: { type: String },                // Optional details
  tags: [String],                               // e.g., nature, travel
  uploadedAt: { type: Date, default: Date.now } // Timestamp
});

module.exports = mongoose.model("Image", ImageSchema);
```

Planned REST Endpoints

1. Get Images (Paginated for Infinite Scroll)

- **Endpoint:** GET /api/images
- **Query Params:**
  - page → page number (default = 1)
  - limit → number of images per request (default = 10)
- **Response:**

```
{
  "page": 1,
  "limit": 10,
  "total": 100,
  "images": [
    {
      "id": "652c1a",
      "url": "https://picsum.photos/300/200?random=1",
      "title": "Sunset",
      "description": "Orange sky over hills",
      "tags": ["nature", "sunset"],
      "uploadedAt": "2025-10-04T07:30:00Z"
    }
  ]
}
```

Supports **infinite scroll** → frontend keeps calling /api/images?page=2&limit=10 when user scrolls down.

---

2. Upload New Image *(Optional)*

- **Endpoint:** POST /api/images
- **Body (JSON or Form-Data):**

```
{
  "url": "https://picsum.photos/300/200?random=55",
  "title": "Beach",
  "description": "Relaxing seashore",
  "tags": ["beach", "travel"]
}
```

- **Response:**

```
{ "message": "Image uploaded successfully", "id": "653f23" }
```

---

3. Get Image by ID

- **Endpoint:** GET /api/images/:id
- **Response:**

```
{
  "id": "652c1a",
  "url": "https://picsum.photos/300/200?random=1",
  "title": "Sunset",
  "description": "Orange sky over hills",
  "tags": ["nature", "sunset"],
```

  "uploaded At": "2025-10-04T07:30:00Z"

}

---

4. Delete Image *(Optional, for Admin)*

- **Endpoint:** DELETE /api/images/:id
- **Response:**

{ "message": "Image deleted successfully" }

---

5. Search / Filter Images *(Optional Enhancement)*

- **Endpoint:** GET /api/images/search?tag=nature
- **Response:** List of images filtered by keyword/tag.

API Workflow for Infinite Scroll

1. Frontend loads first batch → GET /api/images?page=1&limit=10.
2. User scrolls near bottom → fetch next batch → GET /api/images?page=2&limit=10.
3. Repeat until all images are fetched.

Request / Response Format

1. Get Images (Paginated for Infinite Scroll)

- **Endpoint:**
  GET /api/images?page=1&limit=10
- **Request (Example):**

GET /api/images?page=1&limit=10 HTTP/1.1

Host: localhost:5000

Accept: application/json

- **Response (Success):**

```
{
  "page": 1,
  "limit": 10,
  "total": 100,
  "images": [
    {
      "id": "652c1a",
      "url": "https://picsum.photos/300/200?random=1",
      "title": "Sunset",
      "description": "Orange sky over hills",
      "tags": ["nature", "sunset"],
      "uploadedAt": "2025-10-04T07:30:00Z"
    },
    {
      "id": "652c1b",
      "url": "https://picsum.photos/300/200?random=2",
      "title": "Mountain",
      "description": "Snow covered peaks",
      "tags": ["mountain", "nature"],
      "uploadedAt": "2025-10-04T07:32:00Z"
    }
  ]
}
```

- **Response (Error Example):**

```json
{
  "error": "Invalid page number"
}
```

---

2. Upload New Image *(Optional)*

- **Endpoint:**
  POST /api/images
- **Request (JSON body):**

```json
{
  "url": "https://picsum.photos/300/200?random=55",
  "title": "Beach",
  "description": "Relaxing seashore",
  "tags": ["beach", "travel"]
}
```

- **Response (Success):**

```json
{
  "message": "Image uploaded successfully",
  "id": "653f23"
}
```

- **Response (Error Example):**

```json
{
  "error": "URL field is required"
}
```

---

3. Get Image by ID

- **Endpoint:**
  GET /api/images/:id
- **Request Example:**

```
GET /api/images/652c1a HTTP/1.1

Host: localhost:5000

Accept: application/json
```

- **Response (Success):**

```
{
  "id": "652c1a",
  "url": "https://picsum.photos/300/200?random=1",
  "title": "Sunset",
  "description": "Orange sky over hills",
  "tags": ["nature", "sunset"],
  "uploadedAt": "2025-10-04T07:30:00Z"
}
```

- **Response (Error Example):**

```
{
  "error": "Image not found"
}
```

---

4. Delete Image *(Optional)*

- **Endpoint:**
  DELETE /api/images/:id
- **Request Example:**

```
DELETE /api/images/652c1a HTTP/1.1
Host: localhost:5000
Accept: application/json
```

- **Response (Success):**

```
{
  "message": "Image deleted successfully"
}
```

- **Response (Error Example):**

```
{
  "error": "Image not found"
}
```

Database Schema for Infinite Scroll Gallery

Option 1: MongoDB (NoSQL – Recommended for Node.js)

Collection: Images

Each document represents one image entry.

```
{
  "_id": ObjectId("652c1a..."),
  "url": "https://picsum.photos/300/200?random=1",
  "title": "Sunset",
  "description": "Orange sky over hills",
  "tags": ["nature", "sunset"],
  "uploadedAt": ISODate("2025-10-04T07:30:00Z")
}
```

Schema (Mongoose)

```
const mongoose = require("mongoose");

const ImageSchema = new mongoose.Schema({
  url: { type: String, required: true },          // Image link
  title: { type: String },                        // Optional caption
  description: { type: String },                  // Optional details
  tags: [String],                                 // Keywords for search
  uploadedAt: { type: Date, default: Date.now }   // Timestamp
});

module.exports = mongoose.model("Image", ImageSchema);
```

Option 2: MySQL / PostgreSQL (Relational)

Table: images

Column Name	Data Type	Constraints	Description
id	INT (PK)	AUTO_INCREMENT, NOT NULL	Unique identifier
url	VARCHAR(255)	NOT NULL	Image link
title	VARCHAR(100)	NULL	Image title/caption
description	TEXT	NULL	Extra details about image
tags	VARCHAR(255)	NULL	Comma-separated tags

Column Name	Data Type	Constraints	Description
uploaded_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Upload timestamp

SQL Create Table Example

```
CREATE TABLE images (
  id INT PRIMARY KEY AUTO_INCREMENT,
  url VARCHAR(255) NOT NULL,
  title VARCHAR(100),
  description TEXT,
  tags VARCHAR(255),
  uploaded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Schema Relationships (if extended)

- **users** table/collection → If you want to track which user uploaded the image.
- **likes/comments** table/collection → If you extend the project into a social gallery.

Frontend UI/UX Plan & Wireframes

UI/UX Plan

1. **Homepage Layout**
    - **Header / Navbar**: Logo, search bar, and optional filter tags (e.g., "Nature", "Travel").

   - **Gallery Section**: Grid/Masonry layout displaying image thumbnails.
   - **Infinite Scroll Mechanism**: As user scrolls, more images load automatically.
   - **Footer (Optional)**: Simple footer with copyright/info.

2. **User Interaction Flow**
   - **Landing**: User enters the gallery → first batch of images loads.
   - **Scroll Detection**: When user nears the bottom, frontend requests the next set of images via API.
   - **Image Interaction**: Hover effect (zoom-in or title overlay).
   - **Click Action (Optional)**: Clicking an image opens a lightbox/modal with larger preview and description.

3. **Design Considerations (UX)**
   - **Responsive** → Works on mobile, tablet, desktop.
   - **Minimalist UI** → Focus on images, clean background.
   - **Lazy Loading** → Load images only when visible to improve performance.
   - **Feedback** → Loading spinner while fetching new images.

Wireframes

1. Homepage Wireframe (Gallery View)

```
 -------------------------------------------------
| LOGO       | [ Search... ]    | [ Filter Tags ] |
 -------------------------------------------------
|  Img1   |  Img2   |  Img3   |  Img4   |
| Img5   | Img6   | Img7   | Img8   |
| Img9   | Img10  | Img11  | Img12  |
|--------------------------------------------|
|         Loading Spinner (when scrolling)       |
|--------------------------------------------|
```

2. Image Preview (Lightbox/Modal) Wireframe

```
 ------------------------------
|           IMAGE            |
|                            |
|   Title: Sunset View        |
|   Description: Hills at dusk |
|   Tags: nature, sunset       |
|                            |
|   [ Close ]                 |
 ------------------------------
```

3. Mobile Wireframe (Responsive View)

```
 ---------------------
|  LOGO            |
|  [ Search... ]     |
 ---------------------
| Img1 | Img2        |
| Img3 | Img4        |
| Img5 | Img6        |
 ---------------------
(Scroll → auto load)
```

Navigation Flow – Infinite Scroll Gallery

1. Entry Point

- User lands on **Homepage (Gallery View)**.
- First batch of images loads automatically.

2. Explore Images (Infinite Scroll)

- User scrolls down → system detects scroll position.
- Backend API is called → next batch of images loads.
- Loading spinner (UX feedback) appears briefly.
- This repeats infinitely until all images are loaded.

3. Search / Filter (Optional Feature)

- User enters keyword in **Search Bar** OR clicks a **Filter Tag**.
- Gallery refreshes → only matching images are shown.
- Infinite scroll continues with filtered results.

4. Image Interaction

- User hovers over image → overlay shows **title/description**.
- User clicks an image → opens in a **Lightbox/Modal** with:
  - Larger view of image
  - Title, description, tags
  - Close button (returns to gallery).

5. Exit / End Navigation

- User can continue scrolling endlessly, perform new searches, or close the page.

Navigation Flow Diagram (Textual)

```
[Homepage]
  ↓ (Initial batch loads)
[Scroll Down]
  ↓ (Fetch more images from API)
[Gallery Updates with New Images]
  ↓
 ┌──────────────┐
 | Search/Tag | → Refresh Gallery
 └──────────────┘
  ↓
[Click Image] → [Lightbox Preview] → [Close] → Back to Gallery
```

State Management Approach for Infinite Scroll Gallery

1. What Needs to be Managed?

In an infinite scroll gallery, the following states must be tracked:

- **Image List** → The array of already loaded images.
- **Pagination State** → Current page number / cursor / offset.
- **Loading State** → Whether new images are being fetched (show spinner).
- **Error State** → Any failure in API response (e.g., "Network Error").
- **Filter/Search State (Optional)** → Current applied search term or tag.

---

2. Approach for State Management

Option A – Vanilla JS (Simple Projects)

- Maintain state as **in-memory JavaScript variables**.

- Example:

```
let images = [];
let page = 1;
let isLoading = false;
```

- Update the DOM whenever new images are appended.

Option B – React.js with Hooks (Recommended for Scalable Projects)

- Use **useState** to store image list, page number, and loading status.
- Use **useEffect** to trigger new API calls when the page changes.
- Example:

```
const [images, setImages] = useState([]);
const [page, setPage] = useState(1);
const [loading, setLoading] = useState(false);

useEffect(() => {
  setLoading(true);
  fetch(`/api/images?page=${page}&limit=10`)
    .then(res => res.json())
    .then(data => {
      setImages(prev => [...prev, ...data.images]); // Append new images
      setLoading(false);
    });
}, [page]);
```

- When user scrolls → update page → auto-fetch new images.

Option C – Redux / Context API (For Large Applications)

- If the gallery has **global state** (e.g., user authentication, favorites, likes, comments), use a centralized store.
- **Redux / Context** can handle complex state updates across multiple components.

Development and Deployment Plan – Infinite Scroll Gallery

1. Development Plan

The project will be developed in phases to ensure smooth progress:

1. Requirement Analysis
   - Define project scope, objectives, features, and tech stack.
2. UI/UX Design
   - Create wireframes, navigation flow, and UI mockups.
3. Frontend Development
   - Build responsive gallery layout using HTML/CSS/JavaScript (or React).
   - Implement infinite scroll logic and image rendering.
4. Backend Development
   - Create REST API endpoints using Node.js & Express.
   - Connect with database (e.g., MongoDB/MySQL).
   - Handle pagination, filtering, and error handling.
5. Database Setup
   - Design schema for storing images (id, url, title, tags).
   - Populate with sample images for testing.
6. Integration & Testing
   - Connect frontend with backend API.

    - Test infinite scroll, error handling, and responsiveness.

7. Deployment
    - Deploy backend on Heroku / Render / AWS EC2.
    - Deploy frontend on Netlify / Vercel / GitHub Pages.
    - Connect both for live application access.

---

2. Deployment Plan

- Version Control → GitHub/GitLab for code collaboration.
- CI/CD Pipeline → GitHub Actions or GitLab CI for automated build & test.
- Environment Setup → .env files for API keys and database URLs.
- Hosting:
    - Frontend: Netlify or Vercel (auto-deploy from GitHub).
    - Backend: Heroku, Render, or AWS.
    - Database: MongoDB Atlas (cloud-based) or AWS RDS.

---

3. Team Roles & Responsibilities

1. Project Manager

- Oversees progress, timelines, and deliverables.
- Coordinates between frontend, backend, and testing teams.

2. Frontend Developer(s)

- Build responsive UI (HTML, CSS, JavaScript/React).
- Implement infinite scroll and API integration.
- Ensure cross-browser and mobile compatibility.

3. Backend Developer(s)

- Develop REST API endpoints with Node.js + Express.
- Handle pagination, data retrieval, and image management.
- Ensure scalability and secure API design.

4. Database Engineer

- Design and optimize database schema.
- Manage data storage, indexing, and query optimization.

5. QA/Test Engineer

- Write test cases (unit, integration, UI tests).
- Ensure bug-free deployment and smooth UX.

6. DevOps/Deployment Engineer

- Setup cloud hosting for frontend, backend, and database.
- Manage CI/CD pipeline for continuous delivery.
- Monitor server performance and uptime.

Git Workflow – Infinite Scroll Gallery

1. Version Control Setup

- Repository created on GitHub/GitLab/Bitbucket.
- Team members cloned the repo using:
- git clone <repo-url>

---

2. Branching Strategy

We use a feature-branch workflow based on Git best practices:

- main (or master) Branch
    - Always contains production-ready code.
    - Only merged after testing and approval.

- develop Branch
    - Contains the latest development code.
    - Features are merged here before release.
- feature/* Branches
    - Each new feature (UI, API, database schema, infinite scroll logic) is developed in its own branch.
    - Example: feature/frontend-ui, feature/infinite-scroll, feature/api-endpoints.
- bugfix/* Branches
    - Used for fixing identified issues before merging into develop.
- release/* Branch
    - Created from develop when preparing for deployment.
- hotfix/* Branch
    - For urgent fixes directly on main after release.

---

3. Workflow Steps

1. Create New Branch
2. git checkout -b feature/infinite-scroll
3. Work on Feature
    - Implement code locally.
    - Test changes.
4. Stage and Commit Changes
5. git add .
6. git commit -m "Implemented infinite scroll logic"
7. Push Branch to Remote

8. git push origin feature/infinite-scroll
9. Create Pull Request (PR)
   - Merge request raised to develop.
   - Reviewed by team members.
10. Merge into Develop
    - Approved PR is merged into develop.
11. Merge into Main (Release)
    - Once tested, changes from develop are merged into main.
    - Deployment is triggered automatically (CI/CD).

---

4. Benefits of This Workflow

- Parallel Development → Multiple developers can work without conflicts.
- Code Quality → PR reviews ensure clean, tested code.
- Safe Releases → Production (main) remains stable.
- Traceability → Each feature and bugfix tracked separately.

Testing Approach – Infinite Scroll Gallery

1. Objectives of Testing

- Ensure the gallery loads images correctly with infinite scroll.
- Validate the responsiveness and user experience across devices.
- Verify API endpoints return correct data with proper pagination.
- Confirm performance under continuous scrolling and large datasets.
- Detect and fix bugs early for stable deployment.

---

2. Types of Testing

A. Unit Testing

- Scope: Individual components and functions.
- Examples:
  - API function returns correct image batch.
  - Scroll detection triggers the next data fetch.
  - Database query returns correct pagination results.
- Tools: Jest (for JS/React), Mocha/Chai (for Node.js).

B. Integration Testing

- Scope: Interaction between frontend, backend, and database.
- Examples:
  - Frontend scroll → API request → images appended to DOM.
  - Filters/search criteria correctly applied and fetched.
- Tools: Supertest (for Node.js API), Postman for manual checks.

C. Functional Testing

- Scope: User-level scenarios.
- Examples:
  - User scrolls infinitely and new images keep appearing.
  - Clicking an image opens lightbox with correct details.
  - Search bar filters gallery dynamically.

D. Performance Testing

- Scope: App performance under heavy scrolling and large datasets.
- Examples:
  - Test loading speed when fetching 1000+ images.
  - Check memory usage for long browsing sessions.
- Tools: JMeter, Lighthouse.

E. Cross-Browser & Device Testing

- Scope: Ensure UI/UX works consistently on multiple platforms.
- Browsers: Chrome, Firefox, Edge, Safari.
- Devices: Desktop, tablet, mobile.

F. User Acceptance Testing (UAT)

- Scope: End-user validation before deployment.
- Examples:
  - Smooth infinite scroll with no lag.
  - Intuitive navigation and responsiveness.
  - Minimal errors during real usage.

---

3. Test Plan Workflow

1. Write unit tests during development.
2. Test API responses using Postman & automated scripts.
3. Run integration tests after frontend-backend integration.
4. Conduct functional & UI tests with manual and automated tools.
5. Perform load/performance testing with large datasets.
6. Conduct UAT before final deployment.

Hosting & Deployment Strategy – Infinite Scroll Gallery

1. Goals of Deployment

- Make the application accessible online for end-users.
- Ensure scalability, reliability, and security.
- Provide an easy way to update code via CI/CD pipelines.

---

2. Deployment Architecture

- Frontend (UI)
  - Built with HTML, CSS, JavaScript/React.
  - Deployed on Netlify or Vercel for free, fast global CDN distribution.
  - Automatically updates when code is pushed to GitHub.
- Backend (API Layer)
  - Built with Node.js + Express.
  - Deployed on Heroku, Render, or AWS EC2.
  - Exposes REST API endpoints for infinite scroll image retrieval.
- Database (Storage)
  - MongoDB Atlas (cloud-based NoSQL) or MySQL (AWS RDS).
  - Stores images, metadata (title, tags), and pagination data.

---

3. CI/CD Workflow

1. Version Control: Code hosted on GitHub.
2. Continuous Integration:

    - Automated build & test triggered on pull requests (GitHub Actions).

3. Continuous Deployment:
    - Merging to main branch auto-deploys:
        - Frontend → Netlify/Vercel.
        - Backend → Heroku/Render/AWS.
4. Environment Variables:
    - API keys, DB credentials stored securely in .env files or platform secrets.

---

4. Security & Performance Measures

- HTTPS enabled by default on Netlify/Vercel/Heroku.
- CORS policies configured for secure frontend-backend communication.
- Caching and lazy loading for fast gallery rendering.
- Scalability: Backend can scale horizontally (Heroku dynos, AWS autoscaling).

---

5. Deployment Steps (Example: Heroku + Netlify)

Backend (Heroku)

```
# Login to Heroku
heroku login


# Create app
heroku create infinite-scroll-gallery-api
```

```
# Push code
git push heroku main


# Set environment variables
heroku config:set MONGO_URI=<your-db-url>
```

Frontend (Netlify)

- Connect GitHub repo to Netlify.
- Choose build command (npm run build) and deploy directory (/build).
- Automatic deploy on main branch updates.
