**1. Serverless Image Resizer (with Simple Web Page)**

**Purpose**  
This project is designed to **simplify and automate the process of optimizing images** as soon as they are uploaded, ensuring uniform sizing or reduced file size. It tackles a common issue faced by bloggers, small businesses, and marketing teams: manually resizing or compressing images can be time-consuming and prone to errors. By letting a function handle these tasks immediately upon upload, you maintain consistent image quality and dimensions, improve website load times, and streamline your workflow.

Additionally, requiring a **small web page** for image uploads introduces basic front-end functionality, making the project more user-friendly. End users see how effortlessly they can upload pictures and retrieve processed versions without diving into code or command-line tools.

**Use Case**

* Bloggers who want to upload large photos but need them optimized for quick loading.
* E-commerce sites aiming for uniform product images.
* Social media managers needing fast, standardized banner image resizing.

**Difficulty Bonus**  
0

**Basic Rubric (How to Start)**

1. **Front-End (Required)**
   * Create a minimal HTML page featuring an <input type="file" /> element (or drag-and-drop area) and a button to submit.
   * Include basic JavaScript to handle file selection and send it to the back-end.
2. **Choose a Language & Image Processing Library**
   * **Python** with pillow or **Node.js** with sharp/jimp.
   * Ensure the library supports resizing, compressing, or format conversion.
3. **Serverless or Local Function**
   * **Local**: Use serverless-offline, OpenFaaS, or a container running your script.
   * **Free Tier Cloud**: AWS Lambda, Azure Functions, or GCP Cloud Functions if you want real cloud usage.
4. **Storage Setup**
   * **Local**: MinIO (S3-like) or a simple folder watch approach.
   * **Cloud**: A free-tier AWS S3 bucket or equivalent.
5. **Trigger & Process**
   * When an image is uploaded, the function (or script) resizes/compresses it and stores the new version.
   * Return a link or immediate preview of the optimized image.
6. **Validation & Testing**
   * Try JPG, PNG, and different file sizes.
   * Inspect output dimensions and file size for correctness.

**Extra Credit Opportunity (Optional)**

* **Additional Image Operations**: Crop, rotate, apply watermarks, or convert file formats (e.g., JPG to WebP). (at least 2)
* **Multi-Image Batch**: Support uploading several images simultaneously, then provide a ZIP of all processed images.

**2. Static Web App + Serverless Contact Form**

**Purpose**  
This project addresses the **need for a simple “Contact Us” page** on a static website, which can collect inquiries and store or forward them via email. Many small websites don’t have dynamic back-ends, yet they still need user communication. This approach uses a serverless function or local function to process form submissions, removing the burden of maintaining a dedicated server. It demonstrates how front-end and back-end logic can be decoupled while keeping operational costs and complexity low.

**Use Case**

* A small bakery or café’s website collecting cake orders or booking requests.
* A freelance graphic designer’s portfolio site with a client inquiry form.
* A community event sign-up page capturing volunteer details.

**Difficulty Bonus**  
0

**Basic Rubric (How to Start)**

1. **Front-End Creation**
   * Use plain HTML/CSS (or a small React/Vue page) with fields for name, email, and message.
   * Add simple JavaScript to POST form data to your back-end endpoint.
2. **Serverless Function**
   * **Local**: serverless-offline in Node.js/Python or an OpenFaaS function.
   * **Free Cloud**: Netlify Functions, AWS Lambda, or Azure Functions.
   * The function receives form data and optionally performs validations.
3. **Email or Database Handling**
   * For email: Use nodemailer (Node.js), AWS SES, or SendGrid to forward the message to site owners.
   * For data storage: Insert form submissions into a local DB or a free-tier NoSQL table.
4. **Deployment**
   * **Static Site**: Host on GitHub Pages, Netlify, or a local container environment.
   * **Function**: Deploy to the chosen environment so it’s accessible via an endpoint.
5. **Test & Validate**
   * Submit test messages. Confirm an email is received or data is stored.

**Extra Credit Opportunity (Optional)**

* **Spam Protection**: Add client-side validation, server-side checks, or a captcha solution (e.g., Google reCAPTCHA).
* **Auto-Responder**: Send an automated “Thank you for contacting us” email back to the user.

**Why Cloud Is Beneficial**

* **No Dedicated Server**: You only spin up resources when the form is submitted, reducing overhead.
* **Scalable**: If many users decide to send inquiries at the same time, the function seamlessly scales.
* **Easy Maintenance**: Minimal code updates and no patching of servers.

**3. Personal Portfolio with Dynamic Blog (Firebase or Netlify)**

**Purpose**  
A personal portfolio site allows you to **showcase projects, highlight skills**, and create a **dynamic blog** section for sharing news or reflections. This project helps you learn how to manage static content (your portfolio, about page) alongside dynamic entries (blog posts). It’s ideal for individuals looking to establish an online presence without paying hosting fees or running an entire CMS themselves.

**Use Case**

* A student building an online resume and blog for potential employers.
* A freelancer wanting a space to share design mockups, photography, or coding tutorials.
* A hobbyist writing personal reflections or documenting home DIY projects.

**Difficulty Bonus**  
1

**Basic Rubric (How to Start)**

1. **Select a Site Generator or Framework**
   * **Static Generators**: Jekyll, Hugo.
   * **React-Based**: Gatsby, Next.js.
2. **Hosting**
   * **Netlify** or **GitHub Pages** for free static deployments.
   * **Firebase Hosting** (Spark plan) if you want integrated Auth or Firestore usage.
3. **Blog Mechanism**
   * **Markdown Approach**: Store blog posts as Markdown, optionally using a headless CMS like Netlify CMS for easy edits.
   * **Database Approach**: Use Firestore or a local DB if you want a more dynamic solution.
4. **Deployment Pipeline**
   * Connect your repo (GitHub, GitLab) to Netlify or Firebase for auto-builds when you push new content.
5. **Customization**
   * Create separate pages for portfolio sections.
   * Add contact links or an embedded contact form.

**Extra Credit Opportunity (Optional)**

* **Comment System**: Integrate a service like Disqus or build your own with Firestore.
* **Theme Switching**: Offer a dark/light mode or multiple CSS themes.

**4. Firebase (or Local) Chat Room**

**Purpose**  
A real-time chat application demonstrates how to **synchronize user messages instantly** across multiple clients. It educates you on concurrent data handling, user authentication, and potential scaling. Whether you use Firebase’s real-time capabilities or build your own Socket.IO-based solution, you’ll learn to update UI elements live and ensure data consistency.

**Use Case**

* Study groups or class project teams who want a simple chat environment.
* Online communities that need a private, ad-free chat alternative.
* Family or friends wanting a custom space for messaging and staying connected.

**Difficulty Bonus**  
2

**Basic Rubric (How to Start)**

1. **Choose Real-Time Approach**
   * **Firebase**: Realtime Database or Firestore + Firebase Auth.
   * **Local**: Node.js + Socket.IO + a local DB.
2. **Front-End UI**
   * React/Vue or plain HTML/JS for the chat interface (message list, input box).
3. **Authentication**
   * **Firebase Auth** for email/password, Google sign-in, etc.
   * **Local**: Minimal sign-up form or auto-generated user IDs if you want no friction.
4. **Real-Time Message Handling**
   * On every new message, update all connected clients instantly.
5. **Deployment**
   * If using Firebase, hosting the front-end is simple.
   * If local, Docker Compose might help run the Node server + DB.

**Extra Credit Opportunity (Optional)**

* **File/Media Sharing**: Support uploading images or short videos.
* **Typing Indicators & Read Receipts**: Show which users are typing **or** who has seen each message.

**5. Containerized To-Do App with Docker Compose**

**Purpose**  
This project showcases **how microservices architecture** can be applied to a straightforward application: a multi-user to-do list. By splitting the system into front-end, back-end, and database containers, you learn about container orchestration, networking, and how services communicate behind the scenes. It’s a stepping-stone to more complex microservices setups.

**Use Case**

* A personal or small-team tasks manager.
* A stepping-stone for students exploring Docker, container orchestration, and local dev environments.

**Difficulty Bonus**  
2

**Basic Rubric (How to Start)**

1. **Plan the Architecture**
   * **Front-End**: React, Vue, or Angular container.
   * **Back-End API**: Node.js (Express) or Python (Flask/FastAPI) container.
   * **Database**: PostgreSQL, MySQL, or MongoDB container.
2. **Build Dockerfiles**
   * One Dockerfile per service, specifying environment, dependencies, and start commands.
3. **Docker Compose Setup**
   * A docker-compose.yml file describing how containers link, environment variables (DB host, ports), and volume mappings.
4. **Implement CRUD**
   * Basic “Tasks” table or collection.
   * Create, read, update, and delete endpoints.
5. **Test & Verify**
   * docker-compose up, open the front-end in a browser, and try adding tasks.
   * Inspect DB container to confirm data persistence.

**Extra Credit Opportunity (Optional)**

* **User Accounts & Authentication**: Each user has a separate list.
* **Scaling**: Explore Docker Compose “replicas” or switch to a local Kubernetes (Minikube) to see how scaling might work in production.

**6. Trello-Like Kanban Board**

**Purpose**  
A Kanban board organizes tasks into columns (like “To Do,” “In Progress,” “Done”) and allows drag-and-drop card movement. This project introduces **visual project management** concepts, user experience with draggable interfaces, and back-end data storage for boards, lists, and cards. It’s a step beyond a simple to-do list, illustrating how to manage more complex relational data (boards > columns > cards).

**Use Case**

* A small startup or dev team that needs a personalized task board without external dependencies.
* A student group managing project sprints.
* An individual wanting a more feature-rich approach to daily tasks.

**Difficulty Bonus**  
3

**Basic Rubric (How to Start)**

1. **Front-End with Drag-and-Drop**
   * React + react-beautiful-dnd or Vue with a similar library.
   * Provide columns for each workflow stage.
2. **API & Database**
   * Node.js (Express) or Python (Flask) with endpoints for boards, lists, and cards.
   * Database: PostgreSQL, MongoDB, or MySQL.
3. **Containerization (Optional)**
   * Docker Compose for front-end, API, and DB if desired.
4. **Core Features**
   * Create boards, add columns, create new cards, drag-and-drop to reorder or move between columns.
   * Store changes in the database so everything persists.
5. **User Testing**
   * Confirm that tasks remain in the correct column after refresh.
   * Add editing card details (e.g., card title, description).

**Extra Credit Opportunity (Optional)**

* **Real-Time Collaboration**: Use Socket.IO or websockets so that multiple users see updates instantly.
* **User-Based Permissions**: Invite collaborators to specific boards, set roles (admin, read-only, etc.).

**7. IoT Sensor Simulator with Azure IoT Hub (or Local MQTT)**

**Purpose**  
This project **demonstrates how real or simulated devices** (e.g., sensors measuring temperature/humidity) can send data to a central platform for real-time storage, analytics, or dashboarding. It exposes you to event-driven architectures, message brokering (MQTT), and how to set up dashboards for streaming data.

**Use Case**

* A greenhouse or small farm environment logging conditions to optimize plant growth.
* A “smart home” demonstration collecting sensor data (door sensors, temperature, light levels).
* A robotics club streaming data from multiple bots or drones to a single dashboard.

**Difficulty Bonus**  
3

**Basic Rubric (How to Start)**

1. **Messaging Protocol**
   * **Cloud**: Azure IoT Hub free tier.
   * **Local**: Mosquitto (MQTT broker) installed in a container or on your machine.
2. **Simulated Device Script**
   * Python or Node.js program that sends random sensor readings at intervals.
   * Structure the data as JSON (e.g., { "temp": 22.5, "humidity": 60 }).
3. **Data Storage**
   * **Azure**: Use Cosmos DB or Table Storage.
   * **Local**: Try InfluxDB (time-series) or PostgreSQL.
4. **Visualization**
   * Grafana or a custom front-end that queries your DB for the latest readings.
5. **Testing**
   * Confirm your script sends data, see it appear in the DB, and visualize in near real-time on the dashboard.

**Extra Credit Opportunity (Optional)**

* **Alerts & Thresholds**: Trigger notifications if a sensor reading goes above/below a certain level.
* **Edge Computing**: Perform partial data processing on the “device” side before sending final metrics.

**8. Cloud Run “Guestbook” App**

**Purpose**  
A guestbook app is a **simple web application** where visitors can post short messages, greetings, or comments. Deploying it to a serverless container platform like **Google Cloud Run** showcases how containers can be run in a “serverless” fashion, scaling from zero to many instances automatically. This is an approachable way to learn containerization plus a user-facing CRUD app.

**Use Case**

* Special event pages (reunions, weddings) collecting well-wishes.
* A small brand or personal site with a “visitor log.”
* Tech demos illustrating container-based development for novices.

**Difficulty Bonus**  
3

**Basic Rubric (How to Start)**

1. **Containerize the App**
   * Node.js (Express) or Python (Flask/FastAPI) with routes for adding/viewing messages.
   * Build a Dockerfile to define dependencies.
2. **Database**
   * **GCP**: Use Firestore or Cloud SQL.
   * **Local**: Use Docker Compose with a DB container.
3. **Deploy**
   * **Cloud Run**: gcloud run deploy, linking your container image from Container Registry.
   * **Local**: docker-compose up or Minikube if you want a local K8s environment.
4. **Front-End Form**
   * Basic HTML or a small JS framework to let users submit messages and see existing ones.
5. **Testing**
   * Post messages, confirm they appear to all visitors.
   * Tweak memory/CPU settings if you want to see how scaling works.

**Extra Credit Opportunity (Optional)**

* **Admin Panel**: Let an admin remove inappropriate messages.
* **Authentication**: Require users to log in with Google or GitHub before posting.

**9. Real-Time Survey or Polling App**

**Purpose**  
A real-time polling application gathers **immediate feedback** from users (e.g., live audiences, students, or webinar attendees) and updates aggregated results in front of them. It teaches you about real-time data flows, front-end visualization, and possibly user authentication or ephemeral links for each poll.

**Use Case**

* A professor quickly gauging class comprehension mid-lecture.
* Webinar hosts keeping viewers engaged by taking a “pulse check.”
* Small meetups or events that want on-the-fly voting with live charts.

**Difficulty Bonus**  
3

**Basic Rubric (How to Start)**

1. **Front-End Survey UI**
   * Form or poll page built with HTML/JS, React, or Vue.
   * Provide question types (multiple choice, short answer, rating scale).
2. **Real-Time Mechanism**
   * **Socket.IO** for websockets, or use Firebase Realtime DB for instant updates.
3. **Data Storage**
   * Local DB (PostgreSQL or MongoDB) or a free-tier NoSQL (Firestore, DynamoDB).
4. **Visualization**
   * Libraries: Chart.js, D3.js, ECharts to render bar/pie charts.
   * Update the chart whenever new votes arrive.
5. **Load Testing**
   * Simulate multiple participants to ensure the system handles concurrency.

**Extra Credit Opportunity (Optional)**

* **Multiple Questions**: Let users navigate through a poll with multiple queries.
* **Authorization**: Restrict polling to certain participants **or** generate unique links/tokens.

**10. Multi-User Collaborative Whiteboard**

**Purpose**  
A collaborative whiteboard application **allows several users** to draw or annotate on a shared canvas in real time. This advanced project tackles concurrency, real-time data synchronization, possibly complex geometry or shape handling, and user interface design for intuitive drawing tools. It’s an excellent way to understand the challenges of multi-user state management and reliable event broadcasting.

**Use Case**

* Remote teams brainstorming product designs or workflows visually.
* Virtual classrooms where teachers and students collaborate on diagrams, equations, etc.
* Hobbyist communities that want an online “doodle room” for fun.

**Difficulty Bonus**  
4

**Basic Rubric (How to Start)**

1. **Front-End Canvas**
   * Plain HTML <canvas> or libraries like **Fabric.js** / **Konva.js**.
   * Tools for drawing lines, shapes, text, colors, eraser, etc.
2. **Real-Time Communication**
   * **Node.js + Socket.IO** to broadcast each drawing event (mouse down, move, up).
   * Consider a data structure (e.g., shape objects) for consistent updates.
3. **Persistence** (Optional)
   * Use a DB (Redis, MongoDB) to save board states if you want reloading or version history.
4. **User Identification**
   * Nicknames or a simple login system to differentiate drawings or track concurrency.
5. **Scaling & Deployment**
   * Locally with Docker Compose or in the cloud with a container platform.
   * Pay attention to performance: streaming many updates can be CPU-intensive.

**Extra Credit Opportunity (Optional)**

* **Multi-Room Support**: Let users join separate boards, each with its own session.
* **Undo/Redo**: Track user actions so they can revert changes.