Build an Image Optimizer Web Page

A beginner-friendly tutorial for recreating the uploaded PHP, HTML, CSS, and JavaScript project.

What you will build	A single-page image optimizer that lets users select images, preview originals, compress and resize them in the browser, download optimized files, and save copies to the server.
Stack	PHP for the upload endpoint; HTML for structure; CSS for layout; JavaScript, FileReader, Canvas, Blob, Object URLs, and Fetch for optimization.
Audience	A new developer who knows basic HTML and has started learning JavaScript and PHP.
Source file	Uploaded file: Pasted code.php.

Table of Contents

> This tutorial follows the same order as the source file: server-side PHP first, then document structure, styles, and browser behavior. That order matters because PHP must run before HTML is sent to the browser.

1. Big Picture Architecture

The page is a single-file web app. The PHP code at the top prepares the upload folder and handles POST requests. The HTML creates the visible page. The CSS makes the interface feel like a polished card-based tool. The JavaScript reads image files, compresses them using Canvas, creates previews, triggers downloads, and sends optimized Blob data back to PHP.

Layer	Responsibility	Main concept
PHP	Validate uploaded files and save optimized versions into the uploads folder.	$_FILES, finfo MIME detection, move_uploaded_file, sessions
HTML	Create the upload field, quality controls, result area, and preview modal.	Forms, inputs, buttons, semantic containers
CSS	Use variables, responsive flex layouts, cards, buttons, previews, and modal styling.	:root variables, flexbox, media queries
JavaScript	Read local images, draw them to Canvas, convert to Blob, preview, download, and upload.	FileReader, Image, Canvas, Blob, URL.createObjectURL, fetch

> Pipeline mental model: Input - the user selects images. Processing - JavaScript resizes and compresses them in the browser. Output - the user previews, downloads, and optionally saves the optimized images on the PHP server.

Important beginner note: the browser is doing the actual optimization work. PHP is not compressing the image. PHP only receives the already-optimized Blob from JavaScript and saves it to disk.

2. Set Up the Project

Step 1: Create the project folder

Create a folder such as image-optimizer. Inside it, create one file named index.php. The source page is designed as one PHP file that includes the server logic, HTML, CSS, and JavaScript together.

Recommended folder structure

```
image-optimizer/
  index.php
  uploads/        # PHP will create this if it does not exist
```

Step 2: Run it with a local PHP server

Open a terminal inside the project folder and start PHP's built-in server. Then open the local URL in your browser.

Local development command

```
php -S localhost:8000
# Then visit:
# http://localhost:8000/index.php
```

Step 3: Know what files are allowed

The current implementation allows JPEG and PNG after checking the real file MIME type. It does not rely only on the extension typed in the filename.

That detail matters because users can rename a dangerous or unsupported file with a fake .jpg extension. MIME detection helps the server check what the bytes actually represent.

3. Build the PHP Upload Guardrails

3.1 Define upload settings

At the top of index.php, define where uploads will be stored, the maximum file size, and a message string that can be shown to the user after upload attempts.

Core PHP variables

```
<?php
$uploadDir = __DIR__ . '/uploads';
$maxFileSizeBytes = 10 * 1024 * 1024;
$uploadMessage = '';
```

3.2 Add session-based cleanup

The source page keeps the uploads folder temporary. It uses a session timestamp and deletes previous uploaded files on refresh or after inactivity. For a learning project, this is a useful way to prevent the uploads folder from filling up forever.

Source pattern: session cleanup and purge loop

```
$cacheTtlSeconds = 10 * 60; // 10 minutes
$sessionCacheKey = 'imgopt_last_active';

// Session-based cleanup:
// - deletes uploads after 10 minutes of inactivity
// - deletes uploads again on every page refresh (new request)
if (session_status() !== PHP_SESSION_ACTIVE) {
    session_start();
}

$now = time();
$last = isset($_SESSION[$sessionCacheKey]) ? (int)$_SESSION[$sessionCacheKey] : 0;

$shouldPurge = false;
if ($last === 0) {
    $_SESSION[$sessionCacheKey] = $now;
} else {
    if (($now - $last) > $cacheTtlSeconds) {
        $shouldPurge = true;
    }
    // Also purge on every refresh/new request from the same session.
    $shouldPurge = true;
    $_SESSION[$sessionCacheKey] = $now;
}

if ($shouldPurge && is_dir($uploadDir)) {
    foreach (glob($uploadDir . '/*') as $path) {
        if (is_file($path)) {
            @unlink($path);
```

In plain English: start a session, compare the current time with the last active time, decide whether to purge, and if purging is needed, loop through the uploads folder and delete files.

3.3 Create the uploads directory

Before saving files, the page checks whether the uploads directory exists. If it does not exist, PHP tries to create it. Then it verifies that the directory is writable.

Create and verify uploads directory

```
if (!is_dir($uploadDir)) {
```

```
    if (!mkdir($uploadDir, 0755, true) && !is_dir($uploadDir)) {
        $uploadMessage = 'The uploads directory could not be created on the server.';
    }
}

if (is_dir($uploadDir) && !is_writable($uploadDir)) {
    $uploadMessage = 'The uploads directory is not writable on this server.';
```

3.4 Handle only POST upload requests

The source checks for a POST request with both an uploaded image and a fileName field. This prevents upload-handling logic from running on normal page loads.

POST request detection

```
$requestMethod = $_SERVER['REQUEST_METHOD'] ?? 'GET';

if ($requestMethod === 'POST' && isset($_FILES['image']) && isset($_POST['fileName'])) {
    $savedFiles = [];


    // Only handle single file upload (current UI sends one image per request)
    $tmpPath = $_FILES['image']['tmp_name'] ?? '';
    $origError = $_FILES['image']['error'] ?? UPLOAD_ERR_NO_FILE;
```

3.5 Validate the upload before saving

The validation flow checks upload errors, file size, and whether PHP recognizes the temporary path as a real uploaded file. These checks should happen before you trust or move anything.

Upload validation and MIME type check

```
    if ($origError !== UPLOAD_ERR_OK) {
        $uploadMessage = 'Upload error: ' . $origError;
    } elseif ($fileSize <= 0 || $fileSize > $maxFileSizeBytes) {
        $uploadMessage = 'One of the images is larger than the allowed size.';
    } elseif (!is_uploaded_file($tmpPath)) {
        $uploadMessage = 'Invalid upload received.';
    } else {
        $fileName = $_POST['fileName'] ?? 'optimized-image';
        $fileName = preg_replace('/[^A-Za-z0-9._-]/', '_', $fileName) ?: 'optimized-image';
        $baseName = pathinfo($fileName, PATHINFO_FILENAME);

        $finfo = new finfo(FILEINFO_MIME_TYPE);
        $mime = $finfo->file($tmpPath) ?: '';

        // Only allow PNG/JPEG based on actual bytes.
        $ext = match ($mime) {
            'image/png' => '.png',
            'image/jpeg' => '.jpg',
            default => ''
        };

        if ($ext === '') {
            $uploadMessage = 'Unsupported image type.';
```

Strong beginner habit: validate before moving the file. Check the PHP upload error code, reject empty or oversized files, reject invalid temporary paths, sanitize the file name, and use finfo to allow only PNG and JPEG based on real MIME type.

3.6 Save using a safe name

The source sanitizes the incoming filename, extracts a base name, picks the correct extension from MIME type, avoids overwriting existing files, and saves through a temporary path before renaming it into place.

Safe unique filename and atomic-ish save flow

```
            $counter = 1;
            $safeName = $baseName . $ext;

            while (file_exists($uploadDir . '/' . $safeName)) {
                $safeName = $baseName . '-' . $counter++ . $ext;
            }

            $savedPath = $uploadDir . '/' . $safeName;
            $tmpSave = $savedPath . '.tmp.' . bin2hex(random_bytes(6));

            if (move_uploaded_file($tmpPath, $tmpSave)) {
                if (@rename($tmpSave, $savedPath)) {
                    $savedFiles[] = $safeName;
                } else {
                    @unlink($tmpSave);
                    $uploadMessage = 'One of the optimized images could not be saved on the
server.';
                }
            } else {
                $uploadMessage = 'One of the optimized images could not be saved on the server.';
```

4. Create the Page HTML

After the PHP section closes, the file becomes a normal HTML document. The body contains one main card, a hero heading, the file upload input, control inputs, buttons, a results container, and a modal for zoomed previews.

4.1 Start the HTML shell

HTML document start

```
<!DOCTYPE html>
<html lang="en">

<head>

    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Image Optimizer</title>
    <style>
```

4.2 Build the visible card

The main card is the container that holds the entire tool. Inside it, the hero explains what the tool does and the chip labels it as deployment-ready.

Main card and hero section

```
<body>
    <div class="card">
        <div class="hero">
            <div>
                <h2>Image Optimizer for Web</h2>
                <p>Upload one or more photos, reduce their size, and keep them sharp for faster
websites and smoother
                    mobile loading.</p>
            </div>
            <div class="chip">Deployment-ready</div>
        </div>
```

4.3 Add the upload input

The input accepts images and allows multiple file selection. It is wrapped in a label so the whole upload box is clickable.

Multiple image upload input

```
        <label class="upload-box" for="imageUpload">
            <strong>Select images to optimize</strong>
            <input class="file-input" type="file" id="imageUpload" accept="image/*" multiple>
        </label>
```

4.4 Add optimization controls

The quality slider controls JPEG compression quality in the browser. The max width input controls how wide an image can be after resizing. The Run Optimize button processes selected images. The Download All button downloads all optimized versions that already exist in memory.

Quality, max width, and action buttons

```
        <div class="controls">
            <div class="row">
                <label>
                    Quality
                    <input type="range" id="quality" min="0.5" max="1" step="0.05" value="0.8">
                </label>
                <label>
                    Max Width
                    <input type="number" id="maxWidth" value="1200" min="200" max="2500">
                </label>
            </div>

            <div class="row">
                <button id="compressBtn">Run Optimize</button>
                <button id="downloadAllBtn">Download All</button>
            </div>
        </div>
```

4.5 Add results and modal elements

The results container starts empty. JavaScript fills it with preview cards after files are selected. The modal is hidden until a user clicks a preview image.

Dynamic results area and preview modal

```
        <p class="tip">Tip: lower the max width and quality slightly for the best balance between
speed and clarity.</p>

        <?php if ($uploadMessage !== ''): ?>
            <p class="message"><?php echo htmlspecialchars($uploadMessage); ?></p>
        <?php endif; ?>

        <div id="resultsContainer"></div>
        <p id="result" class="result-text"></p>
    </div>

    <div id="previewModal" class="modal">
        <div class="modal-content">
            <span id="closeModal" class="close-btn">&times;</span>
            <img id="modalImage" alt="Zoomed preview">
        </div>
```

5. Style the Interface with CSS

The CSS uses a small design system: variables for colors and shadows, a card layout, dashed upload box, flexible rows, preview cards, and a responsive rule for small screens.

5.1 Use CSS variables

Variables make the design easier to adjust. Instead of hunting for every blue value, you can change --primary once and the page updates everywhere that uses it.

CSS variables

```
:root {
    color-scheme: light;
    --bg: #f3f7ff;
    --panel: #ffffff;
    --text: #172033;
    --muted: #64748b;
    --primary: #2563eb;
    --primary-dark: #1d4ed8;
    --border: #dce5f5;
    --shadow: 0 16px 40px rgba(15, 23, 42, 0.1);
```

5.2 Style the body and card

Background and card layout

```
body {
    font-family: "Segoe UI", Arial, sans-serif;
    margin: 0;
    min-height: 100vh;
    background: linear-gradient(135deg, var(--bg), #e9f2ff 60%, #f8fbff);
    color: var(--text);
    padding: 24px;
}

.card {
    background: var(--panel);
    padding: 28px;
    border-radius: 20px;
    box-shadow: var(--shadow);
    max-width: 980px;
    margin: auto;
    border: 1px solid rgba(255, 255, 255, 0.7);
```

5.3 Style the upload box and controls

Upload box and controls panel

```
.upload-box {
    display: block;
    border: 2px dashed var(--border);
    border-radius: 14px;
    padding: 18px;
    text-align: center;
    color: var(--primary-dark);
    background: #f8fbff;
    cursor: pointer;
    margin-bottom: 16px;
}

.upload-box:hover {
    border-color: var(--primary);
    background: #f0f7ff;
}

.file-input {
    display: block;
    margin: 8px auto 0;
    font-size: 0.95rem;
}

.controls {
    background: #f8fbff;
    border: 1px solid var(--border);
    border-radius: 14px;
    padding: 16px;
    margin-bottom: 14px;
```

5.4 Style preview cards

Each selected image becomes an image-card. The card contains the file label, the original and optimized preview boxes, and a download button.

Image card and preview styling

```
        .image-card {
            display: flex;
            gap: 16px;
            align-items: flex-start;
            justify-content: space-between;
            border: 1px solid var(--border);
            border-radius: 14px;
            padding: 14px;
            margin-top: 14px;
            background: #fcfdff;
            box-shadow: 0 8px 18px rgba(15, 23, 42, 0.04);
        }

        .preview-stack {
            display: flex;
            gap: 12px;
            flex-wrap: wrap;
            flex: 1;
        }

        .preview-box {
            min-width: 180px;
            flex: 1;
        }

        .preview-box h4 {
            margin: 0 0 6px;
            color: var(--muted);
            font-size: 0.95rem;
        }

        .preview {
            width: 100%;
            max-width: 220px;
            height: 160px;
            object-fit: cover;
            border: 1px solid var(--border);
            border-radius: 10px;
            margin-top: 8px;
            cursor: zoom-in;
            background: #f4f7fb;
        }

        .download-btn {
            white-space: nowrap;
```

5.5 Style the zoom modal

Modal styling

```
        .modal {
            position: fixed;
            inset: 0;
            background: rgba(15, 23, 42, 0.72);
            display: none;
            align-items: center;
            justify-content: center;
            z-index: 1000;
            padding: 20px;
        }

        .modal.show {
            display: flex;
        }

        .modal-content {
            position: relative;
```

```
            max-width: 90vw;
            max-height: 90vh;
            background: #fff;
            border-radius: 16px;
            padding: 10px;
            box-shadow: var(--shadow);
        }

        .modal-content img {
            max-width: 100%;
            max-height: 80vh;
            display: block;
            border-radius: 10px;
        }

        .close-btn {
            position: absolute;
            top: 8px;
            right: 12px;
            cursor: pointer;
            font-size: 24px;
            color: #334155;
            background: rgba(255, 255, 255, 0.8);
            border-radius: 50%;
            width: 34px;
            height: 34px;
            display: grid;
```

5.6 Make the page responsive

When the screen is narrow, the image card switches from a horizontal row to a vertical stack. This prevents controls and images from becoming cramped on mobile.

Small-screen CSS rule

```
        @media (max-width: 700px) {
            .card {
                padding: 18px;
            }

            .image-card {
                flex-direction: column;
            }

            .download-btn {
                align-self: flex-start;
            }
```

6. Read and Preview Selected Images

The JavaScript begins by selecting important DOM elements and creating an empty selectedImages array. That array becomes the app's memory of what the user selected.

DOM references and state

```
        const uploadInput = document.getElementById('imageUpload');
        const qualityInput = document.getElementById('quality');
        const maxWidthInput = document.getElementById('maxWidth');
        const resultText = document.getElementById('result');
        const resultsContainer = document.getElementById('resultsContainer');
        const modal = document.getElementById('previewModal');
        const modalImage = document.getElementById('modalImage');
        const closeModal = document.getElementById('closeModal');

        let selectedImages = [];
```

6.1 Convert a file into a Data URL

FileReader turns a local File object into a Data URL. A Data URL can be used as an image src so the browser can show a preview before anything is uploaded.

FileReader helper

```
const readFileAsDataURL = (file) => new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => resolve(reader.result);
    reader.onerror = reject;
    reader.readAsDataURL(file);
});
```

6.2 Load the Data URL into an Image object

The loadImage helper creates an Image object and resolves only after the image has loaded. This is important because the compressor needs naturalWidth and naturalHeight, which are available after loading.

Image loading helper

```
const loadImage = (dataUrl) => new Promise((resolve, reject) => {
    const img = new Image();
    img.onload = () => resolve(img);
    img.onerror = reject;
    img.src = dataUrl;
});
```

6.3 Render selected image cards

The renderCards function clears the results container and creates a new card for every selected image. Each card has an original preview, an optimized preview placeholder, and a download button.

Dynamic preview card rendering

```
const renderCards = () => {
    resultsContainer.innerHTML = '';

    selectedImages.forEach((item) => {
        const card = document.createElement('div');
        card.className = 'image-card';

        const previewStack = document.createElement('div');
        previewStack.className = 'preview-stack';

        const originalBox = document.createElement('div');
        originalBox.className = 'preview-box';
        originalBox.innerHTML = `<h4>Original</h4>`;
        const originalImage = document.createElement('img');
        originalImage.className = 'preview';
        originalImage.src = item.dataUrl;
        originalImage.alt = 'Original preview';
        originalImage.addEventListener('click', () => openModal(originalImage.src));
        originalBox.appendChild(originalImage);

        const optimizedBox = document.createElement('div');
        optimizedBox.className = 'preview-box';
        optimizedBox.innerHTML = '<h4>Optimized</h4>';
        const optimizedImage = document.createElement('img');
        optimizedImage.className = 'preview';
        optimizedImage.alt = 'Optimized preview';
        optimizedImage.addEventListener('click', () => {
            if (optimizedImage.src) openModal(optimizedImage.src);
        });
        optimizedBox.appendChild(optimizedImage);

        previewStack.appendChild(originalBox);
        previewStack.appendChild(optimizedBox);

        const downloadButton = document.createElement('button');
        downloadButton.className = 'download-btn';
```

```
                downloadButton.textContent = 'Download';
                downloadButton.addEventListener('click', () => processImage(item, downloadButton,
true));

                const label = document.createElement('div');
                const strong = document.createElement('strong');
                strong.textContent = item.fileName;
                label.appendChild(strong);

                card.appendChild(label);

                card.appendChild(previewStack);
                card.appendChild(downloadButton);

                resultsContainer.appendChild(card);

                item.originalImage = originalImage;
                item.optimizedImage = optimizedImage;
                item.downloadButton = downloadButton;
```

6.4 Respond when files are selected

When the file input changes, the source loops through each selected file, reads it as a Data URL, loads it into an Image object, stores the result in selectedImages, and then calls renderCards.

File input change event

```
        uploadInput.addEventListener('change', async (event) => {
            const files = Array.from(event.target.files || []);
            if (!files.length) return;

            selectedImages = [];

            for (const file of files) {
                const dataUrl = await readFileAsDataURL(file);
                const img = await loadImage(dataUrl);
                selectedImages.push({
                    file,
                    dataUrl,
                    img,
                    fileName: file.name
                });
            }

            renderCards();
            resultText.textContent =
                `${selectedImages.length} image(s) loaded. Click compress to generate lighter
versions.`;
```

7. Compress Images with Canvas

The processImage function is the heart of the project. It reads the quality and max width controls, calculates a scale factor, draws the image onto a canvas, converts the canvas to a Blob, and then updates the optimized preview.

7.1 Create a Blob from a canvas

Canvas.toBlob is asynchronous. The helper wraps it in a Promise so processImage can use await and keep the code readable.

canvasToBlob Promise helper

```
        const canvasToBlob = (canvas, outputType, quality) => new Promise((resolve, reject) => {
            canvas.toBlob((blob) => {
                if (!blob) {
                    reject(new Error('Compression failed.'));
                    return;
                }
```

```
            resolve(blob);
        }, outputType, quality);
```

7.2 Calculate the new size

The page keeps the original aspect ratio. It calculates a scale that never enlarges the image. If the original is already smaller than the max width, scale stays at 1. If the image is wider, it shrinks proportionally.

Canvas size and drawImage resizing

```
        const maxWidth = parseInt(maxWidthInput.value, 10) || 1200;
        const quality = parseFloat(qualityInput.value) || 0.8;
        const canvas = document.createElement('canvas');
        const ctx = canvas.getContext('2d');
        const originalWidth = item.img.naturalWidth;
        const originalHeight = item.img.naturalHeight;
        const scale = Math.min(1, maxWidth / originalWidth);
        const width = Math.max(1, Math.round(originalWidth * scale));
        const height = Math.max(1, Math.round(originalHeight * scale));

        canvas.width = width;
        canvas.height = height;
        ctx.drawImage(item.img, 0, 0, width, height);
```

7.3 Pick the output type and create the optimized URL

The source keeps PNG files as PNG and treats other supported files as JPEG. After the Blob is created, URL.createObjectURL turns it into a temporary browser URL for preview and download.

Create optimized Blob, preview URL, and download name

```
        const outputType = item.fileName.toLowerCase().endsWith('.png') ? 'image/png' :
'image/jpeg';
        const blob = await canvasToBlob(canvas, outputType, quality);
        // Revoke previous object URL to avoid memory leaks
        if (item.optimizedObjectUrl) {
            URL.revokeObjectURL(item.optimizedObjectUrl);
        }

        const optimizedUrl = URL.createObjectURL(blob);
        item.optimizedObjectUrl = optimizedUrl;

        item.optimizedImage.src = optimizedUrl;
        item.optimizedImage.alt = 'Optimized preview';



        const downloadName = 'optimized-' + sanitizeFileName(item.fileName) + (
            outputType === 'image/png' ? '.png' : '.jpg');
        item.optimizedImage.downloadName = downloadName;
```

Object URLs use browser memory. The source revokes the previous optimized URL before creating a new one. That is a small but professional memory-leak prevention habit.

7.4 Process all selected images

The Run Optimize button checks that at least one file was selected, then processes images one by one. This sequential loop is easier for a beginner to understand than launching all processing at once.

Run Optimize event handler

```
    document.getElementById('compressBtn').addEventListener('click', async () => {
        if (!selectedImages.length) {
            resultText.textContent = 'Please choose at least one image first.';
            return;
        }

        for (const item of selectedImages) {
```

```
            await processImage(item);
        }

        resultText.textContent =
            `Compressed ${selectedImages.length} image(s) and saved them on the server.`;
```

8. Save Optimized Files to the Server

After JavaScript creates the optimized Blob, it sends that Blob to index.php with fetch. PHP receives it through $_FILES and saves it using the upload guardrails from the first section.

8.1 Send Blob data with FormData

Client-side upload to PHP

```
        const formData = new FormData();
        formData.append('image', blob, downloadName);
        formData.append('fileName', downloadName);

        await fetch('index.php', {
            method: 'POST',
            body: formData
```

8.2 Understand the full trip

Step	What happens
1	User selects an image from their computer.
2	JavaScript reads it locally and shows the original preview.
3	JavaScript draws it onto Canvas at a smaller width.
4	Canvas produces an optimized Blob.
5	JavaScript previews the Blob and prepares a download name.
6	fetch sends the Blob to index.php as multipart/form-data.
7	PHP validates and saves the file into uploads/.

> Because fetch does not reload the page, the user stays on the same screen while the server quietly receives each optimized file.

8.3 Display upload feedback

When PHP sets $uploadMessage, the HTML prints it as a green message box. The source uses htmlspecialchars so server messages are safely escaped before being placed into the page.

Escaped PHP message output

```
    <?php if ($uploadMessage !== ''): ?>
        <p class="message"><?php echo htmlspecialchars($uploadMessage); ?></p>
    <?php endif; ?>
```

9. Add Modal Preview and Downloads

9.1 Trigger a single download

The triggerDownload helper creates a temporary anchor, assigns a Blob URL, sets the download filename, clicks it programmatically, and then removes the anchor.

Download helper

```
        const triggerDownload = (url, fileName) => {
            const downloadLink = document.createElement('a');
            downloadLink.href = url;
            downloadLink.download = fileName;
            document.body.appendChild(downloadLink);
            downloadLink.click();
            downloadLink.remove();
```

9.2 Download after optimizing

Each image card's Download button calls processImage for that item. If downloadAfterOptimize is true, processImage triggers the download immediately after the Blob URL is ready.

Single-image download button wiring

```
                const downloadButton = document.createElement('button');
                downloadButton.className = 'download-btn';
                downloadButton.textContent = 'Download';
                downloadButton.addEventListener('click', () => processImage(item, downloadButton,
true));
```

9.3 Download all optimized files

The Download All button checks for optimized Blob URLs that already exist. It then loops over the selected images and downloads each optimized file.

Download All event handler

```
        document.getElementById('downloadAllBtn').addEventListener('click', async () => {
            if (!selectedImages.length) {
                resultText.textContent = 'Please choose at least one image first.';
                return;
            }

            for (const item of selectedImages) {
                if (item.optimizedImage && item.optimizedImage.src &&
item.optimizedImage.src.startsWith(
                        'blob:')) {
                    const fileName = item.optimizedImage.downloadName || 'optimized-' +
item.fileName.replace(
                        /\.[^.]+$/, '') + '.jpg';
                    triggerDownload(item.optimizedImage.src, fileName);
                }
            }

            resultText.textContent = 'Downloaded all optimized images.';
```

9.4 Open and close the image modal

The modal makes previews easier to inspect. Clicking a preview sets the modal image src and adds the show class. Clicking the close button or the dark overlay closes it.

Modal open and close helpers

```
        const openModal = (src) => {
            modalImage.src = src;
            modal.classList.add('show');
        };

        const closePreviewModal = () => {
            modal.classList.remove('show');
            modalImage.src = '';
```

Modal close event listeners

```
        closeModal.addEventListener('click', closePreviewModal);
```

```
        modal.addEventListener('click', (event) => {
            if (event.target === modal) {
                closePreviewModal();
            }
```

10. Test the Page

A new developer should test the page in small stages instead of waiting until the end. Use this checklist while building.

Test	Expected result
Open index.php with PHP server	The card interface appears with upload input, quality slider, max width input, and buttons.
Select one JPEG	The original preview appears and the result text says one image was loaded.
Click Run Optimize	The optimized preview appears, result text updates, and the uploads folder receives the optimized file.
Select multiple images	Each selected image gets its own preview card.
Click Download on one card	That image is processed and downloaded with an optimized filename.
Click Download All after optimizing	All optimized Blob URLs download one by one.
Upload unsupported file	PHP rejects it with an unsupported image type or upload error message.
Resize browser under 700px wide	Image cards stack vertically and remain readable.

Common beginner problems

- The page downloads PHP instead of running it: you opened the file directly. Start a PHP server and visit localhost.
- Uploads folder stays empty: check that PHP has permission to create and write to uploads/.
- Optimized preview is blank: make sure the selected file actually loaded into an Image object before processing.
- Quality slider does not affect PNG much: PNG compression through canvas.toBlob does not use the quality setting the same way JPEG does.
- Fetch returns 404: make sure the file is named index.php or update fetch("index.php") to match your route.

11. Production Hardening Checklist

The project is a good learning app, but any public deployment should be hardened. Use this checklist before putting it on a real website.

Area	Upgrade
Upload security	Keep MIME checking, limit size, reject unknown types, and prevent executable uploads. Store uploads outside the public web root when possible.
CSRF protection	Add a session CSRF token to the page and require it in the POST request.
Authentication	If uploaded images are private, require login and separate each user's files.
Storage cleanup	Use a scheduled cleanup job instead of deleting all files on every refresh in shared production scenarios.
Image formats	Consider WebP or AVIF output, but test browser support and server requirements first.

Error handling	Show fetch errors in the UI and log server-side failures to a safe log file.
Large files	Show progress, limit image dimensions, and avoid processing too many large images at once in the browser.
Accessibility	Add clearer labels, focus states, keyboard support for modal closing, and aria attributes for dialogs.

Security mindset: never trust the browser. The JavaScript may be your UI, but the PHP server must still validate every uploaded file because users can bypass or modify browser-side code.

12. Full Build Path Recap

Here is the complete sequence a new developer can follow to recreate the page from scratch.

1. Create image-optimizer/index.php and run it through php -S localhost:8000.
2. At the top of index.php, define $uploadDir, $maxFileSizeBytes, and $uploadMessage.
3. Start a session and add cleanup logic for temporary uploads.
4. Create uploads/ if missing and confirm it is writable.
5. Handle POST requests only when $_FILES["image"] and $_POST["fileName"] exist.
6. Validate upload errors, file size, and is_uploaded_file before trusting the file.
7. Use finfo to allow only image/png and image/jpeg.
8. Sanitize the file name, prevent duplicates, and save with move_uploaded_file and rename.
9. Create the HTML card with hero text, file input, quality slider, max width input, buttons, results area, and modal.
10. Add CSS variables, card layout, upload box, controls, image cards, preview images, modal, and responsive rules.
11. In JavaScript, select DOM nodes and keep selected images in an array.
12. Use FileReader to create Data URLs and Image objects to access image dimensions.
13. Render one dynamic card per selected image.
14. Use Canvas to resize images without enlarging smaller images.
15. Convert Canvas output to Blob and create an object URL for preview/download.
16. Send the optimized Blob to PHP with FormData and fetch.
17. Wire up Run Optimize, Download, Download All, and modal close behavior.
18. Test single JPEG, single PNG, multiple images, unsupported files, mobile width, and server saves.

Mini mental model: PHP protects and saves. HTML structures. CSS presents. JavaScript transforms. Canvas compresses. Fetch connects the browser result back to the server.

Suggested next exercises

- Add a before-and-after file size comparison for each image.
- Add a progress bar while multiple images are processed.
- Add WebP output as an option.
- Add drag-and-drop support to the upload box.
- Split the single file into index.php, styles.css, and app.js once the beginner understands the full flow.

Appendix: Beginner Glossary

Term	Meaning in this project
Blob	A binary file-like object created by Canvas. It represents the optimized image data.
Canvas	A browser drawing surface. This project draws the original image onto Canvas at a new size, then exports it.
Data URL	A string representation of file data that can be placed directly into an image src for preview.
FileReader	A browser API that reads files selected by the user.
FormData	A browser API for building multipart form uploads for fetch.
MIME type	A content type such as image/png or image/jpeg. PHP checks this from the actual file bytes.
Object URL	A temporary browser URL that points to a Blob in memory. Useful for previews and downloads.
POST request	An HTTP request used to submit data to the server. The optimized Blob is sent to PHP this way.
Session	Server-side state connected to a browser visitor. The source uses it to track last active time.

End of tutorial.