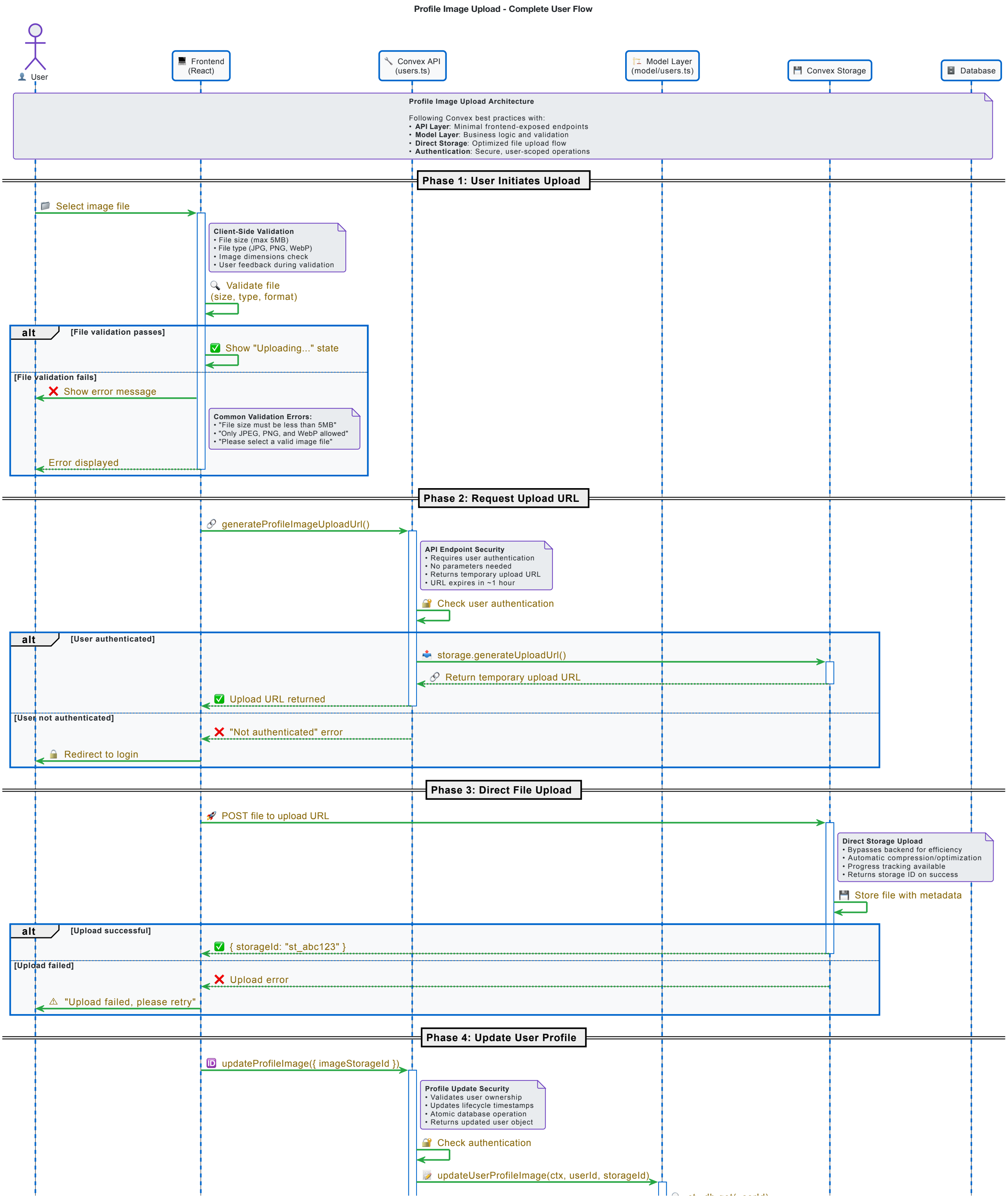
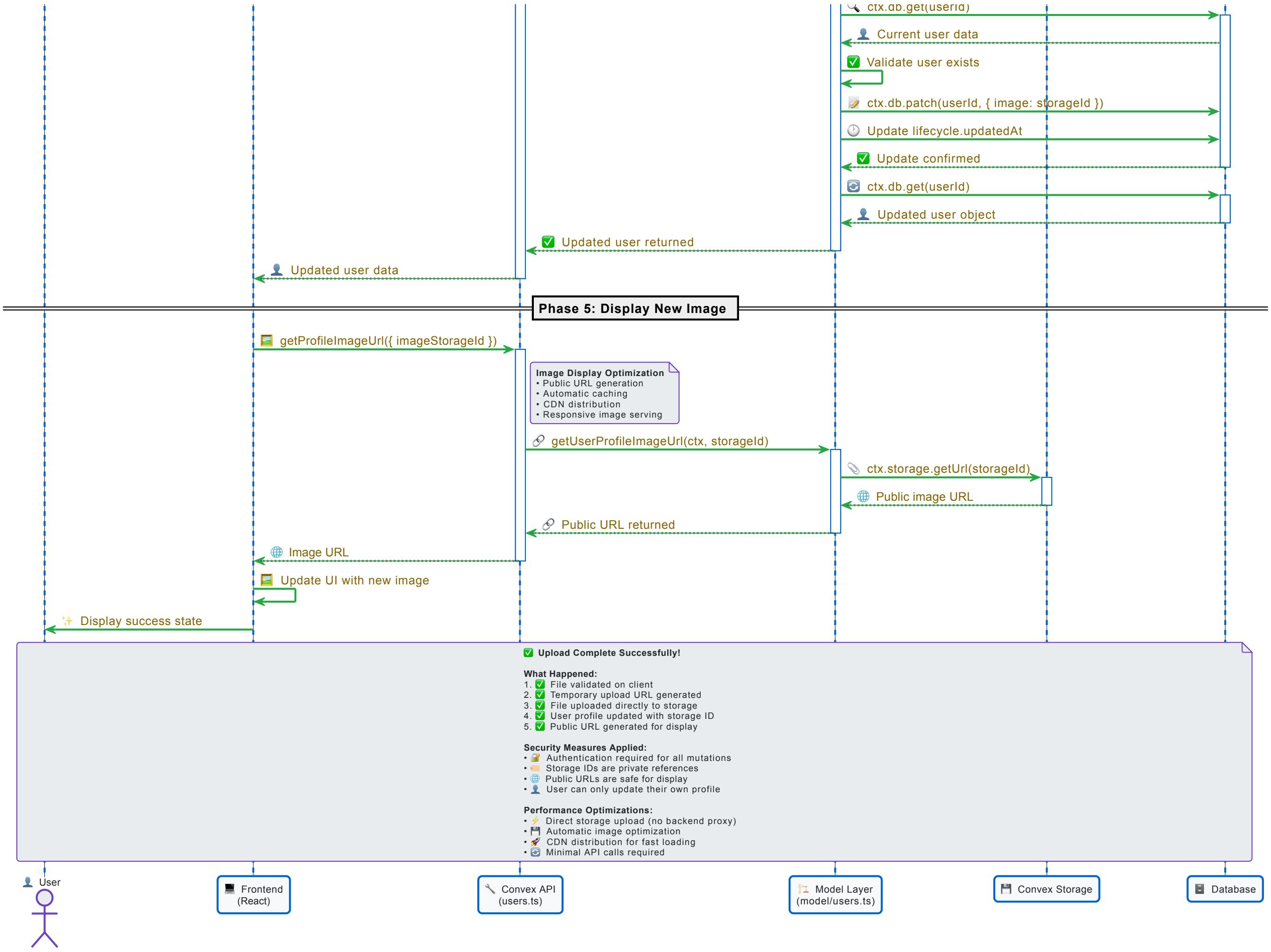
Profile Image Upload - Visual Diagrams

This document contains visual diagrams that illustrate the complete profile image upload system architecture, flow, and error handling strategies.

1. Upload Sequence Diagram

Complete User Flow from File Selection to Image Display


Profile Image Upload - Complete User Flow
User
Frontend (React)
Convex API (users.ts)
Model Layer (model/users.ts)
Convex Storage
Database
Profile Image Upload Architecture
Following Convex best practices with:
• API Layer: Minimal frontend-exposed endpoints
• Model Layer: Business logic and validation
• Direct Storage: Optimized file upload flow
• Authentication: Secure, user-scoped operations
Phase 1: User Initiates Upload
Select image file
Client-Side Validation
• File size (max 5MB)
• File type (JPG, PNG, WebP)
• Image dimensions check
• User feedback during validation
Validate file (size, type, format)
alt [File validation passes]
✅ Show "Uploading..." state
[File validation fails]
❌ Show error message
Common Validation Errors:
• "File size must be less than 5MB"
• "Only JPEG, PNG, and WebP allowed"
• "Please select a valid image file"
Error displayed
Phase 2: Request Upload URL
generateProfileImageUploadUrl()
API Endpoint Security
• Requires user authentication
• No parameters needed
• Returns temporary upload URL
• URL expires in ~1 hour
Check user authentication
alt [User authenticated]
storage.generateUploadUrl()
Return temporary upload URL
✅ Upload URL returned
[User not authenticated]
❌ "Not authenticated" error
Redirect to login
Phase 3: Direct File Upload
POST file to upload URL
Direct Storage Upload
• Bypasses backend for efficiency
• Automatic compression/optimization
• Progress tracking available
• Returns storage ID on success
Store file with metadata
alt [Upload successful]
✅ { storageId: "st_abc123" }
[Upload failed]
❌ Upload error
⚠ "Upload failed, please retry"
Phase 4: Update User Profile
updateProfileImage({ imageStorageId })
Profile Update Security
• Validates user ownership
• Updates lifecycle timestamps
• Atomic database operation
• Returns updated user object
Check authentication
updateUserProfileImage(ctx, userId, storageId)


ctx.db.get(userId)
Current user data
Validate user exists
ctx.db.patch(userId, { image: storageId })
Update lifecycle.updatedAt
Update confirmed
ctx.db.get(userId)
Updated user object
Updated user returned
Updated user data
Phase 5: Display New Image
getProfileImageUrl({ imageStorageId })
Image Display Optimization
• Public URL generation
• Automatic caching
• CDN distribution
• Responsive image serving
getUserProfileImageUrl(ctx, storageId)
ctx.storage.getUrl(storageId)
Public image URL
Public URL returned
Image URL
Update UI with new image
Display success state
✅ Upload Complete Successfully!
What Happened:
1. ✅ File validated on client
2. ✅ Temporary upload URL generated
3. ✅ File uploaded directly to storage
4. ✅ User profile updated with storage ID
5. ✅ Public URL generated for display
Security Measures Applied:
• 🔐 Authentication required for all mutations
• 🏷️ Storage IDs are private references
• 🌐 Public URLs are safe for display
• 👤 User can only update their own profile
Performance Optimizations:
• ⚡ Direct storage upload (no backend proxy)
• 💾 Automatic image optimization
• 🚀 CDN distribution for fast loading
• 🔄 Minimal API calls required
User
Frontend (React)
Convex API (users.ts)
Model Layer (model/users.ts)
Convex Storage
Database

This sequence diagram shows the complete user journey for uploading a profile image:

Key Flow Steps:

1. **File Selection & Validation** - User selects image, frontend validates file type and size
2. **Upload URL Generation** - Frontend requests temporary upload URL from Convex API
3. **Direct Storage Upload** - File uploads directly to Convex Storage (bypasses backend)
4. **Profile Update** - User profile updated with storage ID through API and model layer
5. **Image Display** - Public URL generated for displaying the uploaded image

Architecture Highlights:

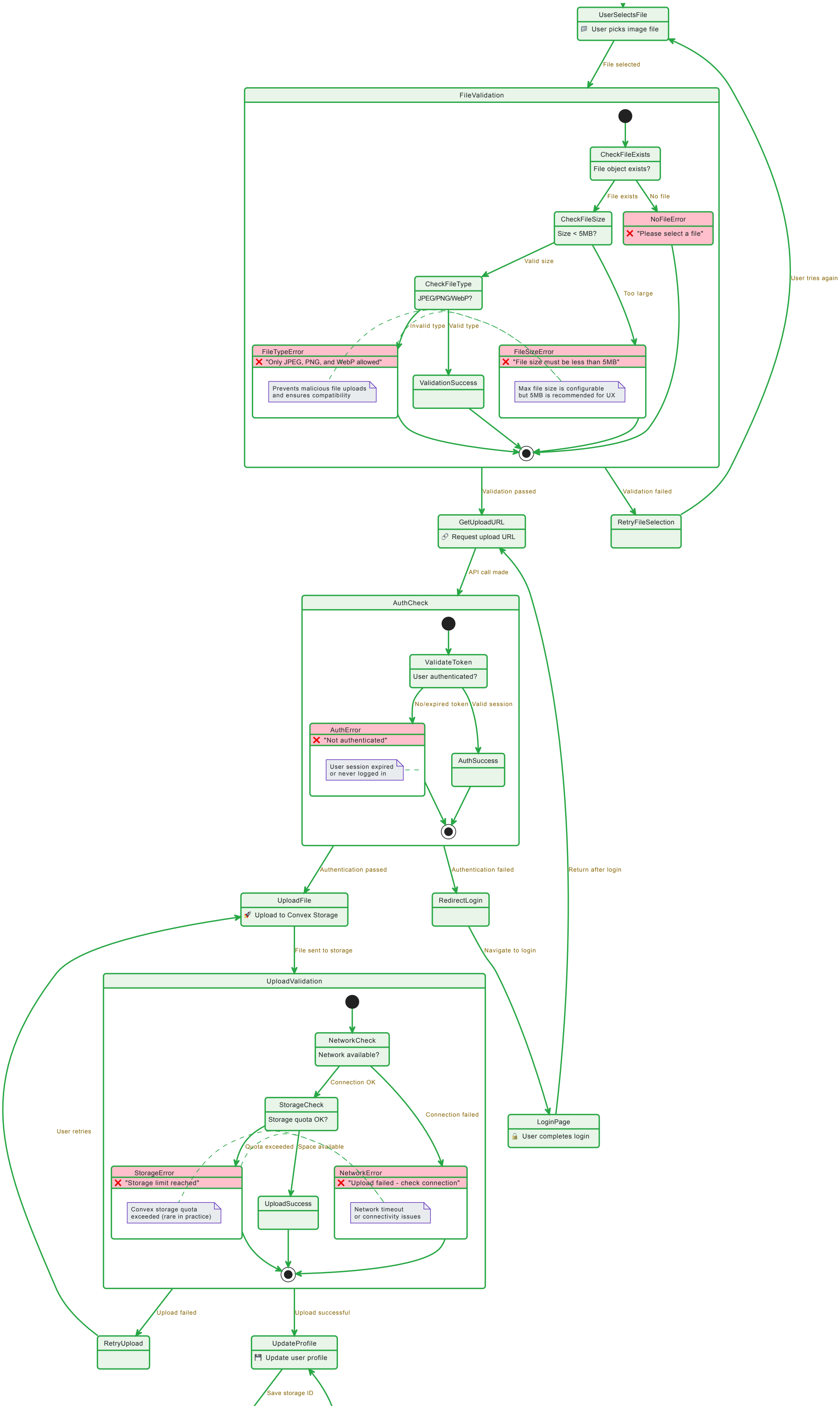
- 🔐 **Authentication**: Required for all mutations, secure user ownership validation
- ⚡ **Performance**: Direct storage upload optimizes speed and reduces server load
- 🏗️ **Clean Architecture**: Clear separation between API layer and business logic model
- 🛡️ **Security**: Storage IDs remain private, public URLs safe for display

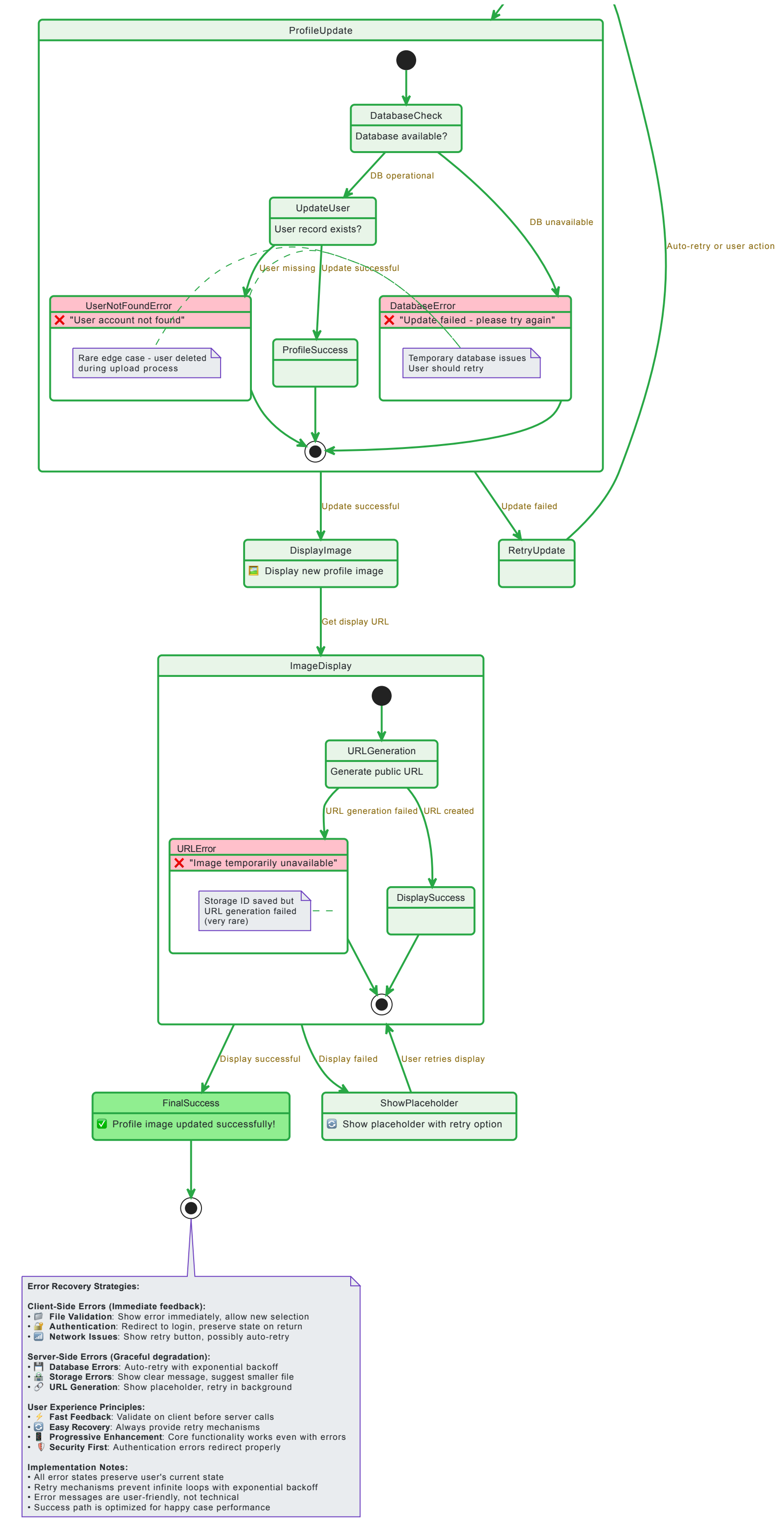
---

2. Error Handling Strategy

Comprehensive Error Scenarios and Recovery Paths

Profile Image Upload - Comprehensive Error Handling Strategy
User interaction


UserSelectsFile
🖼 User picks image file
File selected
FileValidation
CheckFileExists
File object exists?
File exists
No file
CheckFileSize
Size < 5MB?
NoFileError
❌ "Please select a file"
Valid size
Too large
CheckFileType
JPEG/PNG/WebP?
Invalid type
Valid type
FileTypeError
❌ "Only JPEG, PNG, and WebP allowed"
Prevents malicious file uploads and ensures compatibility
ValidationSuccess
FileSizeError
❌ "File size must be less than 5MB"
Max file size is configurable but 5MB is recommended for UX
User tries again
Validation passed
Validation failed
GetUploadURL
🔗 Request upload URL
RetryFileSelection
API call made
AuthCheck
ValidateToken
User authenticated?
No/expired token
Valid session
AuthError
❌ "Not authenticated"
User session expired or never logged in
AuthSuccess
Authentication passed
Authentication failed
Return after login
UploadFile
🚀 Upload to Convex Storage
RedirectLogin
File sent to storage
Navigate to login
UploadValidation
NetworkCheck
Network available?
Connection OK
Connection failed
StorageCheck
Storage quota OK?
User retries
Quota exceeded
Space available
StorageError
❌ "Storage limit reached"
Convex storage quota exceeded (rare in practice)
UploadSuccess
NetworkError
❌ "Upload failed - check connection"
Network timeout or connectivity issues
LoginPage
🔒 User completes login
Upload failed
Upload successful
RetryUpload
UpdateProfile
💾 Update user profile
Save storage ID



Error Recovery Strategies:

Client-Side Errors (Immediate feedback):
- 📁 **File Validation**: Show error immediately, allow new selection
- 🔐 **Authentication**: Redirect to login, preserve state on return
- 📶 **Network Issues**: Show retry button, possibly auto-retry

Server-Side Errors (Graceful degradation):
- 💾 **Database Errors**: Auto-retry with exponential backoff
- 🏧 **Storage Errors**: Show clear message, suggest smaller file
- 🔗 **URL Generation**: Show placeholder, retry in background

User Experience Principles:
- ⚡ **Fast Feedback**: Validate on client before server calls
- 🔄 **Easy Recovery**: Always provide retry mechanisms
- 📱 **Progressive Enhancement**: Core functionality works even with errors
- 🛡️ **Security First**: Authentication errors redirect properly

Implementation Notes:
- All error states preserve user's current state
- Retry mechanisms prevent infinite loops with exponential backoff
- Error messages are user-friendly, not technical
- Success path is optimized for happy case performance

This diagram maps all possible error scenarios and their recovery strategies:

Error Categories:

- 📁 **File Validation Errors** - Invalid file type, size limits, corrupt files
- 🔐 **Authentication Errors** - Expired sessions, unauthorized access
- 🌐 **Network Errors** - Connection failures, timeouts, upload interruptions
- 💾 **Storage Errors** - Quota exceeded, storage system failures
- 🗄️ **Database Errors** - Profile update failures, user not found

Recovery Strategies:

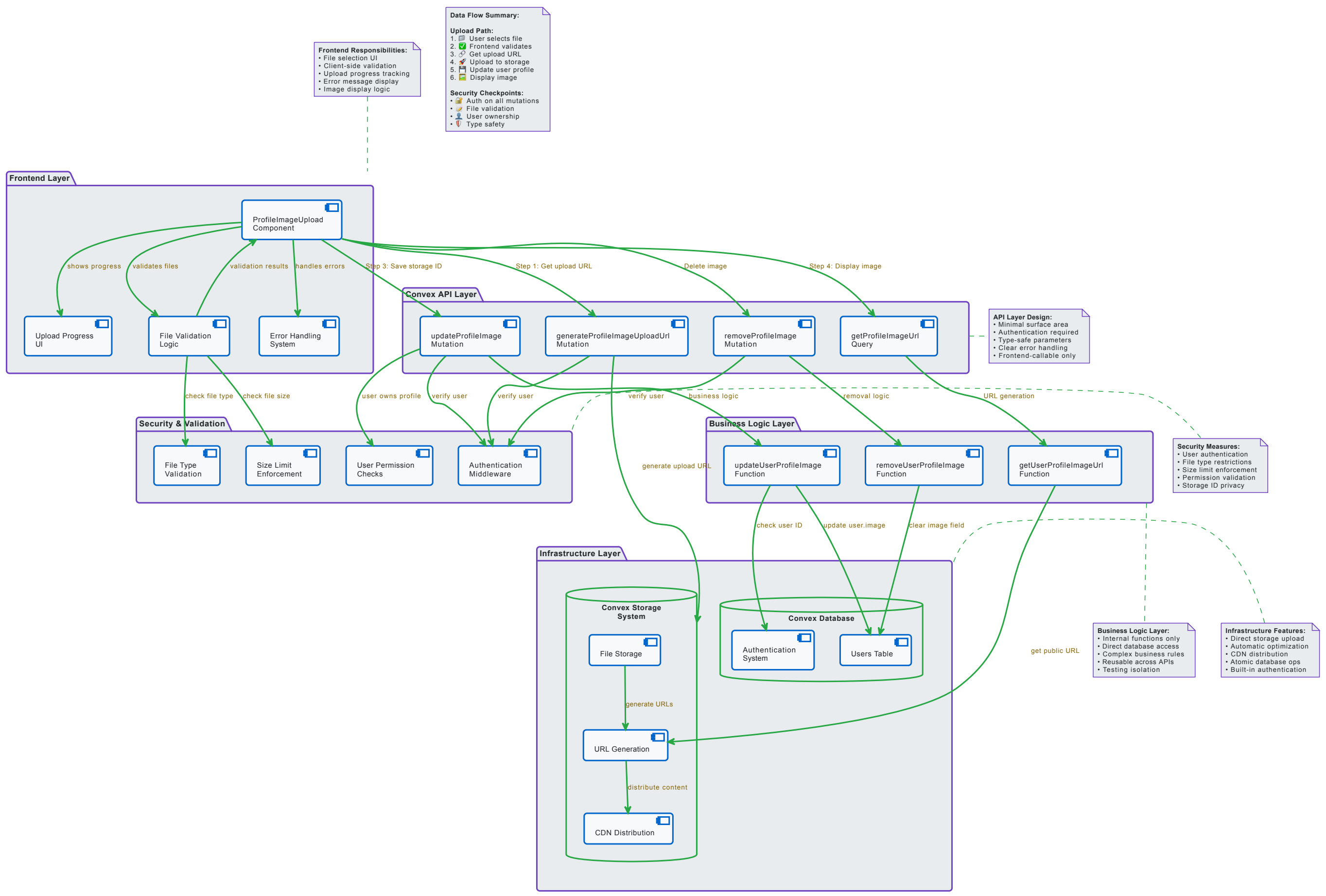
- 🔄 **Automatic Retry** - For transient network/server issues
- 👤 **User-Initiated Retry** - Clear error messages with retry buttons
- 🔒 **Authentication Recovery** - Redirect to login while preserving state
- 📝 **Graceful Degradation** - Core functionality works even with partial failures

---

3. System Architecture

Component Relationships and Data Flow

Profile Image Upload - System Architecture & Component Relationships



Frontend Responsibilities:
- File selection UI
- Client-side validation
- Upload progress tracking
- Error message display
- Image display logic

Data Flow Summary:

Upload Path:
1. 📁 User selects file
2. ✅ Frontend validates
3. 🔗 Get upload URL
4. 🚀 Upload to storage
5. 💾 Update user profile
6. 🖼️ Display image

Security Checkpoints:
- 🔐 Auth on all mutations
- 📝 File validation
- 👤 User ownership
- 🛡️ Type safety

API Layer Design:
- Minimal surface area
- Authentication required
- Type-safe parameters
- Clear error handling
- Frontend-callable only

Security Measures:
- User authentication
- File type restrictions
- Size limit enforcement
- Permission validation
- Storage ID privacy

Business Logic Layer:
- Internal functions only
- Direct database access
- Complex business rules
- Reusable across APIs
- Testing isolation

Infrastructure Features:
- Direct storage upload
- Automatic optimization
- CDN distribution
- Atomic database ops
- Built-in authentication

This component diagram illustrates the system architecture and relationships:

Architecture Layers:

1. **Frontend Layer** - React components, file validation, upload UI, error handling
2. **API Layer** - Minimal frontend-exposed endpoints following Convex best practices
3. **Model Layer** - Business logic functions, database operations, validation rules
4. **Infrastructure Layer** - Convex Storage system, database, CDN distribution

Key Design Patterns:

- 🎯 **Separation of Concerns** - Each layer has distinct responsibilities
- 🔒 **Security First** - Authentication middleware and permission checks
- 📦 **Minimal API Surface** - Only essential endpoints exposed to frontend
- 🏗️ **Reusable Business Logic** - Model layer functions can be shared across APIs

Data Flow:

1. **Upload Request** → Frontend validates → API generates URL
2. **File Upload** → Direct to storage → Returns storage ID
3. **Profile Update** → API → Model layer → Database
4. **Image Display** → Query API → Storage URL → CDN delivery

---

Implementation Notes

For Frontend Developers:

- Use the **sequence diagram** to understand the complete API integration flow
- Reference the **error handling diagram** to implement proper error states and recovery
- Follow the **architecture diagram** to understand component boundaries and responsibilities

For Backend Developers:

- The **architecture diagram** shows the model layer separation and business logic placement
- The **sequence diagram** illustrates the authentication checkpoints and security measures
- The **error handling diagram** guides comprehensive error response design

For QA/Testing:

- The **error handling diagram** provides a comprehensive test case matrix
- The **sequence diagram** shows all integration points that need testing
- The **architecture diagram** identifies component boundaries for unit vs integration testing

These diagrams serve as living documentation that should be updated as the feature evolves.