Reliability & Safety Implementation Guide

This guide provides step-by-step instructions for implementing the reliability and safety features in CareLinkAI.

📚 Table of Contents

Overview

This implementation provides comprehensive reliability and safety features:

- ✅ **Standardized Error Handling** - Consistent API error responses
- ✅ **Input Validation** - Zod schemas for all forms
- ✅ **XSS Protection** - DOMPurify sanitization
- ✅ **RBAC** - Role-based access control
- ✅ **Structured Logging** - JSON logs with context
- ✅ **Retry Logic** - S3 uploads and DB transactions
- ✅ **File Validation** - Type and size checks
- ✅ **Toast Notifications** - User-friendly feedback

Quick Start

1. Dependencies

All required dependencies are already installed:
- `zod` - Validation
- `react-hot-toast` - Notifications
- `isomorphic-dompurify` - XSS protection

2. File Structure

```
src/
├── lib/
│   ├── errors/
│   │   ├── api-errors.ts       # Backend error handling
│   │   └── client-errors.ts    # Frontend error handling
│   ├── auth/
│   │   ├── rbac.ts             # Backend RBAC utilities
│   │   └── client-rbac.ts      # Frontend RBAC hooks
│   ├── validation/
│   │   ├── schemas.ts          # Zod schemas
│   │   └── sanitize.ts         # XSS protection
│   ├── s3/
│   │   └── upload.ts           # S3 utilities with retry
│   ├── db/
│   │   └── transactions.ts     # DB transaction wrappers
│   └── logger.ts               # Structured logging
├── components/
│   └── ui/
│       └── toast-provider.tsx  # Toast notifications
└── app/
    └── api/
        └── example-hardened/   # Reference implementation
            └── route.ts
```

3. Global Setup

The `ToastProvider` is already integrated in the root layout.

Core Utilities

Error Handling

Backend

```
import {
  handleApiError,
  createErrorResponse,
  ErrorCode,
  ErrorResponses
} from '@/lib/errors/api-errors';

// Use try-catch in all route handlers
export async function POST(req: NextRequest) {
  try {
    // Your code here

    // Manual error responses
    if (!data.email) {
      return createErrorResponse(
        ErrorCode.VALIDATION_ERROR,
        'Email is required',
        400,
        { field: 'email' }
      );
    }

    // Or use helper methods
    return ErrorResponses.validation('Invalid email', { field: 'email' });

  } catch (error) {
    // Automatic error handling
    return handleApiError(error, {
      path: req.nextUrl.pathname,
      method: 'POST',
      userId: session?.user.id,
    });
  }
}
```

Frontend

```
import {
  showErrorToast,
  showSuccessToast,
  handleApiResponse
} from '@/lib/errors/client-errors';

// Display toast notifications
async function handleSubmit() {
  try {
    const response = await fetch('/api/resource', {
      method: 'POST',
      body: JSON.stringify(data),
    });

    const result = await handleApiResponse(response);
    showSuccessToast('Resource created successfully!');

  } catch (error) {
    showErrorToast(error);
  }
}
```

Input Validation

```
import { registerSchema, updateProfileSchema } from '@/lib/validation/schemas';

// Server-side validation
const result = registerSchema.safeParse(body);

if (!result.success) {
  return createErrorResponse(
    ErrorCode.VALIDATION_ERROR,
    'Invalid input',
    400,
    { errors: result.error.format() }
  );
}

// Use validated data
const validData = result.data;
```

Input Sanitization

```
import {
  sanitizeObject,
  sanitizeRichText,
  sanitizeUrl
} from '@/lib/validation/sanitize';

// Sanitize entire object
const sanitized = sanitizeObject(data, {
  richTextFields: ['bio', 'description'],
  urlFields: ['website', 'photoUrl'],
});

// Individual sanitization
const cleanBio = sanitizeRichText(user.bio);
const cleanUrl = sanitizeUrl(user.website);
```

RBAC

Backend

```
import {
  requireAuth,
  requireRole,
  requireOwnership
} from '@/lib/auth/rbac';
import { UserRole } from '@prisma/client';

export async function GET(req: NextRequest) {
  try {
    // Require authentication
    const session = await requireAuth();

    // Or require specific role
    const session = await requireRole(UserRole.FAMILY);

    // Or require any of multiple roles
    const session = await requireAnyRole([UserRole.ADMIN, UserRole.OPERATOR]);

    // Check resource ownership
    const resource = await prisma.lead.findUnique({ where: { id } });
    requireOwnership(session, resource.familyId);

    // Your code here

  } catch (error) {
    return handleApiError(error);
  }
}
```

Frontend

```
import {
  useIsAdmin,
  useHasRole,
  RoleGate,
  AdminOnly
} from '@/lib/auth/client-rbac';
import { UserRole } from '@prisma/client';

function MyComponent() {
  const isAdmin = useIsAdmin();
  const isFamily = useHasRole(UserRole.FAMILY);

  return (
    <>
      {isAdmin && <AdminButton />}

      <RoleGate roles={[UserRole.ADMIN, UserRole.OPERATOR]}>
        <StaffOnlyContent />
      </RoleGate>

      <AdminOnly>
        <AdminPanel />
      </AdminOnly>
    </>
  );
}
```

Logging

```
import { logger } from '@/lib/logger';

// Info logging
logger.info('User registered', {
  userId: user.id,
  role: user.role,
});

// Error logging
logger.error('API error', {
  userId,
  path: req.url,
  error: error.message,
});

// Audit logging
logger.audit('Lead created', {
  userId: session.user.id,
  leadId: lead.id,
});
```

S3 Uploads

```
import {
  generatePresignedUrl,
  uploadToS3,
  validateFile
} from '@/lib/s3/upload';

// Backend: Generate presigned URL
const validation = validateFile(filename, contentType, size);
if (!validation.valid) {
  return ErrorResponses.validation(validation.error);
}

const key = generateS3Key(userId, 'credentials', filename);
const result = await generatePresignedUrl(key, contentType);

if (!result.success) {
  return ErrorResponses.external(result.error);
}

// Frontend: Upload file
const uploadResult = await uploadToS3(
  presignedUrl,
  file,
  contentType,
  3, // retries
  (progress) => setProgress(progress)
);

if (!uploadResult.success) {
  showErrorToast(uploadResult.error);
}
```

Database Transactions

```
import { executeTransaction, executeWithRetry } from '@/lib/db/transactions';

// Use transactions for related operations
const result = await executeTransaction(async (tx) => {
  const user = await tx.user.create({ data: userData });
  const profile = await tx.family.create({
    data: { userId: user.id, ...profileData }
  });
  return { user, profile };
});

// Use retry for single operations
const user = await executeWithRetry(() =>
  prisma.user.findUnique({ where: { id } })
);
```

Migration Guide

Migrating an Existing API Route

1. **Add imports**

```
import { handleApiError, createErrorResponse, ErrorCode } from '@/lib/errors/api-er-
rors';
import { requireAuth, requireRole } from '@/lib/auth/rbac';
import { logger } from '@/lib/logger';
import { sanitizeObject } from '@/lib/validation/sanitize';
import { yourSchema } from '@/lib/validation/schemas';
```

1. **Wrap in try-catch**

```
export async function POST(req: NextRequest) {
  try {
    // Your existing code
  } catch (error) {
    return handleApiError(error, { path: req.nextUrl.pathname });
  }
}
```

1. **Add authentication**

```
const session = await requireAuth();
// or
const session = await requireRole(UserRole.FAMILY);
```

1. **Add validation**

```
const result = yourSchema.safeParse(body);
if (!result.success) {
  return createErrorResponse(
    ErrorCode.VALIDATION_ERROR,
    'Invalid input',
    400,
    { errors: result.error.format() }
  );
}
```

1. **Sanitize input**

```
const sanitized = sanitizeObject(result.data, {
  richTextFields: ['bio'],
});
```

1. **Add ownership checks**

```
requireOwnership(session, resource.userId);
```

1. **Add logging**

```
logger.audit('Action performed', {
  userId: session.user.id,
  resourceId,
});
```

Best Practices

1. Always Validate Input

- Use Zod schemas for ALL user input
- Validate on server-side (client-side is optional)
- Sanitize before storing in database

2. Use Standardized Errors

- Use `handleApiError` for all errors
- Use specific error codes
- Log errors with context
- Don’t expose sensitive info to client

3. Enforce RBAC

- Check authentication on all protected routes
- Validate role permissions
- Check resource ownership
- Use permission helpers

4. Log Important Events

- Authentication attempts
- Resource creation/updates
- Permission violations
- Errors with context

5. Handle External Services

- Use retry logic for S3 uploads
- Validate files before upload
- Use transactions for related DB operations
- Handle network failures gracefully

Common Patterns

Pattern 1: Create Resource

```
export async function POST(req: NextRequest) {
  try {
    // 1. Auth & role check
    const session = await requireRole(UserRole.FAMILY);

    // 2. Parse & validate
    const body = await req.json();
    const result = createLeadSchema.safeParse(body);

    if (!result.success) {
      return createErrorResponse(
        ErrorCode.VALIDATION_ERROR,
        'Invalid input',
        400,
        { errors: result.error.format() }
      );
    }

    // 3. Sanitize
    const sanitized = sanitizeObject(result.data);

    // 4. Create with transaction
    const resource = await executeTransaction(async (tx) => {
      return await tx.lead.create({
        data: {
          ...sanitized,
          familyId: session.user.id,
        },
      });
    });

    // 5. Log
    logger.audit('Lead created', {
      userId: session.user.id,
      leadId: resource.id,
    });

    // 6. Return
    return NextResponse.json(
      { success: true, data: resource },
      { status: 201 }
    );

  } catch (error) {
    return handleApiError(error, {
      path: req.nextUrl.pathname,
      userId: session?.user.id,
    });
  }
}
```

Pattern 2: Update Resource

```
export async function PATCH(req: NextRequest) {
  try {
    // 1. Auth
    const session = await requireAuth();

    // 2. Parse
    const body = await req.json();
    const { id, ...updates } = body;

    // 3. Validate
    const result = updateSchema.partial().safeParse(updates);
    if (!result.success) {
      return ErrorResponses.validation('Invalid input', {
        errors: result.error.format(),
      });
    }

    // 4. Check exists & ownership
    const existing = await prisma.resource.findUnique({ where: { id } });
    if (!existing) {
      return ErrorResponses.notFound('Resource');
    }

    requireOwnership(session, existing.userId);

    // 5. Sanitize & update
    const sanitized = sanitizeObject(result.data);
    const updated = await prisma.resource.update({
      where: { id },
      data: sanitized,
    });

    // 6. Log
    logger.audit('Resource updated', {
      userId: session.user.id,
      resourceId: id,
    });

    // 7. Return
    return NextResponse.json({ success: true, data: updated });

  } catch (error) {
    return handleApiError(error);
  }
}
```

Pattern 3: File Upload

```
export async function POST(req: NextRequest) {
  try {
    // 1. Auth
    const session = await requireAuth();

    // 2. Parse & validate
    const body = await req.json();
    const result = fileUploadSchema.safeParse(body);

    if (!result.success) {
      return ErrorResponses.validation('Invalid file', {
        errors: result.error.format(),
      });
    }

    // 3. Validate file
    const validation = validateFile(
      result.data.filename,
      result.data.contentType,
      result.data.size
    );

    if (!validation.valid) {
      return ErrorResponses.validation(validation.error);
    }

    // 4. Generate S3 key and presigned URL
    const key = generateS3Key(
      session.user.id,
      'credentials',
      result.data.filename
    );

    const urlResult = await generatePresignedUrl(
      key,
      result.data.contentType
    );

    if (!urlResult.success) {
      return createErrorResponse(
        ErrorCode.EXTERNAL_SERVICE_ERROR,
        urlResult.error || 'Upload failed',
        503
      );
    }

    // 5. Log
    logger.info('Presigned URL generated', {
      userId: session.user.id,
      key,
    });

    // 6. Return
    return NextResponse.json({
      success: true,
      data: {
        uploadUrl: urlResult.url,
        fileUrl: `https://${process.env.AWS_S3_BUCKET}.s3.amazonaws.com/${key}`,
      },
    });

  } catch (error) {
    return handleApiError(error);
  }
}
```

Testing

Test Error Handling

```
# Invalid input
curl -X POST http://localhost:5000/api/leads \
  -H "Content-Type: application/json" \
  -d '{"targetType": "INVALID"}'

# Expected: 400 with VALIDATION_ERROR

# Unauthorized
curl -X GET http://localhost:5000/api/family/profile

# Expected: 401 with UNAUTHORIZED

# Forbidden
curl -X GET http://localhost:5000/api/admin/users \
  -H "Cookie: next-auth.session-token=family_token"

# Expected: 403 with FORBIDDEN
```

Test Input Validation

```
// Submit XSS attempt
const maliciousData = {
  bio: '<script>alert("XSS")</script>',
  website: 'javascript:alert("XSS")',
};

// Expected: HTML stripped, javascript: URL rejected
```

Test RBAC

```
// Try accessing operator route as family
// Expected: 403 Forbidden

// Try accessing another user's profile
// Expected: 403 Forbidden

// Try admin route as operator
// Expected: 403 Forbidden
```

Test File Uploads

```
// Upload oversized file (15MB)
// Expected: 400 with "File is too large"

// Upload invalid type (.exe)
// Expected: 400 with "Invalid file type"

// Simulate S3 failure
// Expected: Retry 3 times, then error
```

Reference

- Reliability Checklist (./RELIABILITY_CHECKLIST.md)
- API Error Codes (./API_ERROR_CODES.md)
- RBAC Matrix (./RBAC_MATRIX.md)
- Example Route (../src/app/api/example-hardened/route.ts)