Upload System Analysis

Executive Summary

Issue: Timesheet file upload system was completely broken. Files were selected in UI but never uploaded to S3, no database records created, and TimesheetReviewModal showed no documents.

Root Cause: Inconsistent upload architecture between timesheet and contract systems. Frontend handled S3 uploads for timesheets while backend handled them for contracts, leading to a fragmented, error-prone flow.

Solution: Refactored timesheet upload system to match the working contract upload pattern - backend now handles all S3 uploads and database record creation.

Phase 1: Working Contract Document Upload System

Architecture Overview

Flow: Frontend → Backend → S3 Upload → Database Record → Success

Components

1. Frontend Hook (`hooks/contracts/useContractDocuments.ts`)

```
export function useContractDocuments(contractId: string) {
  const utils = api.useUtils();

  const uploadMutation = api.simpleContract.uploadDocument.useMutation({
    onSuccess: () => {
      void utils.simpleContract.listDocuments.invalidate();
    }
  });

  return {
    uploadDocument: uploadMutation.mutate,
    isUploading: uploadMutation.isPending
  };
}
```

Key Points:
- Simple mutation call
- Query invalidation after success
- No S3 logic in frontend

2. Backend Router (`server/api/routers/simpleContract.ts`)

```
uploadDocument: tenantProcedure
  .use(hasPermission(P.CONTRACT.UPDATE_GLOBAL))
  .input(z.object({
    contractId: z.string(),
    pdfBuffer: z.string(), // base64
    fileName: z.string(),
    mimeType: z.string(),
    fileSize: z.number(),
  }))
  .mutation(async ({ ctx, input }) => {
    // 1. Upload to S3
    const buffer = Buffer.from(input.pdfBuffer, "base64");
    const s3FileName = `tenant_${ctx.tenantId}/contract/${input.contractId}/v1/${in-
put.fileName}`;
    const s3Key = await uploadFile(buffer, s3FileName);

    // 2. Create database record
    const document = await ctx.prisma.document.create({
      data: {
        entityType: "contract",
        entityId: input.contractId,
        s3Key,
        fileName: input.fileName,
        mimeType: input.mimeType,
        fileSize: input.fileSize,
      },
    });

    return document;
  })
```

Key Points:
- Accepts base64 file buffer
- Handles S3 upload internally
- Creates database record with S3 key
- Single source of truth for upload logic

3. S3 Helper (`lib/s3.ts`)

```
export async function uploadFile(
  buffer: Buffer,
  fileName: string,
  contentType?: string
): Promise<string> {
  const key = buildKey(fileName);

  const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: buffer,
    ContentType: contentType || "application/octet-stream",
  });

  await s3Client.send(command);
  return key;
}
```

Key Points:
- Simple, reusable function
- Returns S3 key
- Used by all upload mutations

4. Document Model (Prisma Schema)

```
model Document {
  id         String   @id @default(cuid())
  tenantId   String

  entityType String?
  entityId   String?

  s3Key      String  // Primary storage identifier
  fileName   String
  mimeType   String
  fileSize   Int

  version          Int      @default(1)
  isLatestVersion  Boolean  @default(true)

  createdAt DateTime @default(now())
}
```

Key Points:
- Uses `s3Key` as primary storage identifier
- Supports versioning
- Generic entity relation (entityType/entityId)

Phase 2: Broken Timesheet Upload System

Original Architecture (BROKEN)

Flow: Frontend → S3 Upload → Frontend → Backend → Database Record → ❌ Failure

Root Causes Identified

1. Split Responsibility

Frontend (`TimesheetSubmissionForm.tsx`):

```
// ❌ BROKEN: Frontend handles S3 upload
async function uploadFileToS3(file: File | null, prefix: string): Promise<string | nul
l> {
  if (!file) return null;

  const arrayBuffer = await file.arrayBuffer();
  const buffer = Buffer.from(arrayBuffer);
  const key = `${prefix}/${Date.now()}-${file.name}`;

  const uploadedKey = await uploadToS3(buffer, key, file.type);
  return uploadedKey;
}

// Upload, then pass URL to backend
const timesheetFileUrl = await uploadFileToS3(timesheetFile, "timesheet-documents");
await uploadTimesheetDocument.mutateAsync({
  timesheetId,
  fileName: timesheetFile.name,
  fileUrl: timesheetFileUrl, // Pre-uploaded S3 key
});
```

Problems:
- Frontend has S3 credentials (security risk)
- Error handling split between frontend/backend
- No atomic transaction (S3 upload succeeds, DB record fails)
- Inconsistent with contract pattern

2. Incomplete Backend Mutation

Backend (`server/api/routers/timesheet.ts`):

```
// ❌ BROKEN: Only creates DB record
uploadExpenseDocument: tenantProcedure
  .input(z.object({
    timesheetId: z.string(),
    fileName: z.string(),
    fileUrl: z.string(), // Expects pre-uploaded S3 key
    fileSize: z.number(),
  }))
  .mutation(async ({ ctx, input }) => {
    // Just create database record
    const document = await ctx.prisma.timesheetDocument.create({
      data: {
        timesheetId: input.timesheetId,
        fileName: input.fileName,
        fileUrl: input.fileUrl, // Stores pre-uploaded key
        fileSize: input.fileSize,
      },
    });
    return document;
  })
```

Problems:
- No S3 upload logic
- Assumes frontend uploaded successfully
- No validation of S3 key
- Creates orphaned DB records if frontend fails

3. Missing Document Display

Frontend (`TimesheetReviewModal.tsx`):

```
// ❌ BROKEN: Using legacy fields
<TimesheetFileViewer
  fileUrl={data.timesheetFileUrl} // Legacy field
  fileName="timesheet.pdf"
/>

{data.expenseFileUrl && (
  <TimesheetFileViewer
    fileUrl={data.expenseFileUrl} // Legacy field
/>
)}
```

Problems:
- Uses deprecated `timesheetFileUrl` / `expenseFileUrl` fields
- Does not query `TimesheetDocument` records
- Shows “No files” even when documents exist

4. Schema Inconsistency

TimesheetDocument Model:

```
model TimesheetDocument {
  id          String   @id @default(cuid())
  timesheetId String

  fileName    String
  fileUrl     String   // ⚠️ Uses fileUrl instead of s3Key
  fileSize    Int

  category    String   @default("expense")
  uploadedAt  DateTime @default(now())
}
```

Problems:
- Uses `fileUrl` instead of `s3Key` (inconsistent naming)
- No versioning support
- No entity-type abstraction

Comparison: Contract vs Timesheet (Before Fix)

Aspect	Contract (Working)	Timesheet (Broken)
S3 Upload	Backend handles	Frontend handles
File Input	base64 buffer	Pre-uploaded S3 key
Error Handling	Centralized in backend	Split across frontend/backend
Atomicity	✅ Transactional	❌ Split operations
Security	✅ S3 creds in backend	❌ S3 creds exposed to frontend
Consistency	✅ Single pattern	❌ Fragmented
Document Display	✅ Queries Document records	❌ Uses legacy fields

---

Conclusion

The timesheet upload system failed due to:

1. **Architectural Inconsistency**: Frontend handled S3 uploads instead of backend
2. **Incomplete Mutation**: Backend only created DB records, didn't upload files
3. **Missing Display Logic**: Review modal didn't query TimesheetDocument records
4. **Error-Prone Flow**: Split operations prevented atomic transactions

The solution: **Refactor timesheet uploads to match the proven contract pattern** - backend handles all uploads and database operations.