SpaBooker — End-to-End Flow Fix & Build Notes

This document consolidates all fixes, code patches, and checklists to enforce the intended flow: Owner registers → uploads license → Super Admin approves → owner logs in → completes Setup Wizard → then can add staff/services/etc. It also removes tenant leaks (client-sent spaId) and guarantees the server derives tenancy on every admin API.

# 1) Executive Summary

Root cause of “Spa does not exist”: the client screens were sending spaId (often defaulting to 1). Admin routes must never trust client-provided spaId. The server must derive the spa from the session via injectAdminSpa. Additionally, the Super Admin approval path must create a Spa and link the admin (set user.adminSpaId). Write operations are gated behind ensureSetupComplete until the Setup Wizard is finished.

# 2) Server Fixes (Required)

## 2.1 Approve Endpoint — Create Spa + Link Admin + Approve

On POST /api/super-admin/applications/:id/approve, implement idempotent creation/linkage and return { spaId }:

// server/routes/super-admin.ts  
router.post("/applications/:id/approve", isSuperAdmin, async (req, res) => {  
 const id = Number(req.params.id);  
 const app = await storage.getAdminApplicationById(id);  
 if (!app) return res.status(404).json({ message: "Application not found" });  
 if (app.status !== "pending") return res.status(409).json({ message: "Application already reviewed" });  
 if (!app.licenseUrl) return res.status(400).json({ message: "License document is required" });  
  
 const user = await storage.getUser(app.userId);  
 if (!user) return res.status(404).json({ message: "User not found" });  
  
 const existingSpa = (user.adminSpaId && (await storage.getSpaById(user.adminSpaId))) ||  
 (await storage.findSpaByOwnerAndName(app.userId, app.businessName));  
  
 const spa = existingSpa || await storage.createSpa({  
 name: app.businessName,  
 ownerUserId: app.userId,  
 setupComplete: false,  
 status: "active",  
 });  
  
 await storage.updateUser(app.userId, {  
 status: "approved",  
 role: user.role === "super\_admin" ? "super\_admin" : "admin",  
 adminSpaId: spa.id,  
 });  
  
 await storage.updateAdminApplication(id, { status: "approved", reviewedAt: new Date() });  
 return res.json({ ok: true, spaId: spa.id });  
});

## 2.2 Tenant Derivation — injectAdminSpa

Middleware to attach the admin’s spa to each request, instead of accepting spaId from the client:

export const injectAdminSpa: RequestHandler = async (req, res, next) => {  
 const user = req.user as any;  
 const userId = user.claims.sub;  
 const dbUser = await storage.getUser(userId);  
 if (!dbUser) return res.status(500).json({ message: "User not found in database" });  
  
 if (!dbUser.adminSpaId) {  
 return res.status(400).json({  
 message: "No spa assigned to this admin account. Please complete the setup wizard first.",  
 setupRequired: true,  
 });  
 }  
  
 const spa = await storage.getSpaById(dbUser.adminSpaId);  
 if (!spa) {  
 return res.status(404).json({  
 message: "Spa not found. Please contact support or complete the setup wizard again.",  
 setupRequired: true,  
 });  
 }  
  
 (req as any).adminSpa = spa;  
 (req as any).dbUser = dbUser;  
 next();  
};

## 2.3 Wizard Gate — ensureSetupComplete

Block writes until the Setup Wizard is complete for that spa:

export async function ensureSetupComplete(req, res, next) {  
 const spa = (req as any).adminSpa;  
 if (!spa) return res.status(500).json({ message: "Spa context missing" });  
 if (!spa.setupComplete) {  
 return res.status(412).json({ message: "Setup wizard incomplete", setupRequired: true });  
 }  
 next();  
}

## 2.4 Route Protection Order

For all admin write routes (staff, services, etc.), apply middleware in this order:

isAuthenticated → isAdmin → injectAdminSpa → ensureSetupComplete (for writes)

Example staff create:

router.post("/api/admin/staff",  
 isAuthenticated,  
 isAdmin,  
 injectAdminSpa,  
 ensureSetupComplete,  
 async (req, res) => {  
 const spa = (req as any).adminSpa;  
 const { name, email, ...rest } = req.body;  
 const newStaff = await storage.createStaff({ spaId: spa.id, name, email, ...rest });  
 res.json(newStaff);  
 }  
);

## 2.5 DomainError for Business Logic (Optional but Recommended)

Use a DomainError type to map expected errors to non-500 statuses in handleRouteError:

export class DomainError extends Error {  
 status: number; code?: string;  
 constructor(message: string, status = 400, code?: string) { super(message); this.status = status; this.code = code; }  
}

function handleRouteError(res: any, error: any, message: string) {  
 if (error instanceof DomainError) {  
 return res.status(error.status).json({ message: error.message, code: error.code });  
 }  
 if (error.name === "ZodError") return res.status(400).json({ message: "Validation error", errors: error.errors });  
 if (error.code === "23505") return res.status(409).json({ message: "Duplicate entry - this record already exists" });  
 if (error.code === "23503") return res.status(400).json({ message: "Foreign key constraint failed - referenced record does not exist" });  
 if (error.code === "23502") return res.status(400).json({ message: "Missing required field" });  
 console.error(message, error);  
 return res.status(500).json({ message: error.message || message });  
}

# 3) Client Fixes (React)

## 3.1 SuperAdmin UI

• Treat date fields as strings in the type.  
• Use the approve endpoint response to show the created spaId for QA.

type AdminApplication = {  
 id: number;  
 userId: string;  
 businessName: string;  
 businessType: string;  
 status: string;  
 appliedAt: string; // ← string  
 reviewedAt: string | null; // ← string|null  
 rejectionReason: string | null;  
 user: { id: string; email: string; firstName: string; lastName: string } | null;  
};

onSuccess: (data: { spaId?: string }) => {  
 toast({  
 title: "Application Approved",  
 description: data?.spaId ? `Spa linked (ID: ${data.spaId})` : "Approved successfully.",  
 });  
 queryClient.invalidateQueries({ queryKey: ["/api/super-admin/applications"], exact: false });  
},

## 3.2 AdminStaff Screen — Remove spaId & Handle 412/401/403

• Never send spaId; server derives tenancy.  
• Redirect to /admin/setup on 412 with setupRequired.  
• Handle 401/403 gracefully.

// createStaffMutation  
apiRequest('POST', '/api/admin/staff', data); // no spaId  
  
// updateStaffMutation  
apiRequest('PUT', `/api/admin/staff/${id}`, data); // no spaId  
  
// onError handler snippet  
const status = error?.status ?? error?.response?.status;  
const data = error?.data ?? error?.response?.data;  
if (status === 412 && data?.setupRequired) { toast({ title: "Finish setup", description: "Complete the setup wizard first." }); window.location.href = "/admin/setup"; return; }  
if (status === 401) { toast({ title: "Session expired", description: "Please log in again." }); window.location.href = "/api/login"; return; }  
if (status === 403) { toast({ title: "Forbidden", description: "You don’t have access.", variant: "destructive" }); return; }

## 3.3 AdminServices Screen — Remove spaId & Handle 412/401/403

• Do not fetch /api/user for spaId.  
• Remove spaId from create/update service & create category payloads.  
• Ensure numeric fields are numbers.

// Service create/update payload  
const dataToSubmit = {  
 name: serviceForm.name,  
 categoryId: serviceForm.categoryId,  
 description: serviceForm.description || null,  
 price: Number(serviceForm.price),  
 duration: parseInt(serviceForm.duration, 10),  
};  
  
// Category create  
apiRequest('POST', '/api/admin/service-categories', { name: newCategoryName.trim() }); // no spaId  
  
// Standard onError  
// (same as in AdminStaff, route to /admin/setup on 412)

# 4) Storage Layer Validation (As Implemented)

You implemented multi-layer validation and Postgres-aware error handling. Key validations include:

• Service Category: name required; spaId must exist.  
• Service: name, spaId, duration (>0), price required; spaId and categoryId verified.  
• Staff: name, spaId required; email unique if provided.

Keep validations; optionally convert generic Error(...) to DomainError for accurate HTTP statuses.

# 5) SQL Diagnostics (Quick Checks)

Run these to verify data integrity after deployment:

-- 5.1 Approved apps with no Spa created  
SELECT a.id, a.user\_id, a.business\_name  
FROM admin\_applications a  
LEFT JOIN spas s ON s.owner\_user\_id = a.user\_id AND s.name = a.business\_name  
WHERE a.status = 'approved' AND s.id IS NULL;  
  
-- 5.2 Users approved but not linked to a spa  
SELECT u.id, u.email  
FROM users u  
WHERE u.role IN ('admin','super\_admin') AND u.status = 'approved'  
 AND (u.adminSpaId IS NULL OR NOT EXISTS (SELECT 1 FROM spas s WHERE s.id = u.adminSpaId));

# 6) Integration & E2E Tests

1) Approve creates Spa & links admin: POST /approve → 200 with { spaId }.  
2) Admin writes blocked pre-setup: POST /api/admin/staff → 412 { setupRequired: true }.  
3) Complete wizard: POST /api/admin/setup/complete → 200; subsequent staff/service writes succeed.  
4) Idempotency: repeated approve does not duplicate spa or link.  
5) Tenant derivation: admin routes succeed without client-sent spaId.

UI E2E: verify AdminStaff/AdminServices screens redirect to /admin/setup when 412 is returned.

# 7) Optional: whoami Endpoint for Frontend Guards

GET /api/admin/whoami → { role, adminSpaId, setupComplete }  
- If !adminSpaId or !setupComplete on the client, redirect to /admin/setup early.

# 8) Final Drop Checklist

• Approve endpoint returns { spaId } and sets user.adminSpaId  
• All admin routes: isAuthenticated → isAdmin → injectAdminSpa → ensureSetupComplete (writes)  
• Client never sends spaId; all mutations rely on server tenancy  
• 412 handler redirects to /admin/setup; 401 to /api/login; 403 shows forbidden toast  
• Storage validations intact; consider DomainError mapping for non-500 responses  
• SQL diagnostics return 0 rows