

Contents

SECTION 41: COMPLETE INTERNATIONALIZATION SYSTEM (v4.7.0)	1
	1
41.1 ARCHITECTURE OVERVIEW	2
The Problem	2
The Solution	2
Supported Languages (18)	3
System Architecture	3
41.2 DATABASE SCHEMA	5
Migration: 041_localization_system.sql	5
41.3 TYPESCRIPT TYPES & CONSTANTS	13
File: packages/shared/src/i18n/types.ts	13
File: packages/shared/src/i18n/constants.ts	16
41.4 AI TRANSLATION LAMBDA	17
File: lambda/localization/translate.ts	17
File: lambda/localization/process-queue.ts	21
41.5 LOCALIZATION SERVICE (API)	22
File: lambda/localization/api.ts	22
41.6 REACT i18n IMPLEMENTATION	31
File: packages/shared/src/i18n/react/I18nProvider.tsx	31
File: packages/shared/src/i18n/react/LanguageSelector.tsx	34
File: packages/shared/src/i18n/react/Trans.tsx	35
41.7 ESLINT PLUGIN (HARDCODE PREVENTION)	37
File: packages/eslint-plugin-i18n/src/index.ts	37
File: .eslintrc.js (Project Root Addition)	43
41.8 SWIFT LOCALIZATION SERVICE (THINK TANK APP)	43
File: ThinkTank/Services/LocalizationService.swift	43
File: ThinkTank/Views/Components/LocalizedText.swift	48
41.9 ADMIN DASHBOARD - LOCALIZATION MANAGEMENT	50
File: admin-dashboard/app/localization/page.tsx	50
41.10 CDK INFRASTRUCTURE	60
File: cdk/lib/localization-stack.ts	60
41.11 INITIAL TRANSLATION SEED DATA	63
Migration: 041b_seed_localization.sql	63
	66

SECTION 41: COMPLETE INTERNATIONALIZATION SYSTEM (v4.7.0)

CRITICAL: This section implements complete i18n/localization for RADIAN. ZERO hardcoded strings allowed - ALL user-facing text must come from the localization registry.

41.1 ARCHITECTURE OVERVIEW

The Problem

Before v4.7.0, RADIANT had scattered hardcoded strings throughout the codebase:

BEFORE (v4.6.0 and earlier):

Hardcoded Strings Everywhere

React Component:

```
<Button>Submit</Button> ← Hardcoded!  
<Alert>Error occurred</Alert> ← Hardcoded!
```

Lambda Response:

```
{ message: "User not found" } ← Hardcoded!  
throw new Error("Invalid input") ← Hardcoded!
```

Swift App:

```
Text("Welcome") ← Hardcoded!  
Alert("Connection failed") ← Hardcoded!
```

Problems:

- Cannot support multiple languages
- No central place to update text
- Inconsistent terminology across apps
- No way to A/B test copy changes

The Solution

v4.7.0 implements **complete database-driven localization**:

AFTER (v4.7.0):

Centralized Localization Registry

localization_registry (Single Source of Truth)

```
key: "button.submit"  
default_text: "Submit"  
context: "Primary action button"  
category: "ui.buttons"
```

```

localization_translations
  en: "Submit"           (status: approved)
  es: "Enviar"            (status: approved)
  fr: "Soumettre"         (status: approved)
  de: "Absenden"          (status: ai_translated)
  ja: " "                 (status: approved)
... 18 languages total

```

```

React: {t('button.submit')}
Lambda: t('error.user_not_found', { userId })
Swift: L10n.button.submit

```

Supported Languages (18)

Code	Language	Native Name	RTL	Status
en	English	English	No	Primary
es	Spanish	Español	No	Supported
fr	French	Français	No	Supported
de	German	Deutsch	No	Supported
pt	Portuguese	Português	No	Supported
it	Italian	Italiano	No	Supported
nl	Dutch	Nederlands	No	Supported
pl	Polish	Polski	No	Supported
ru	Russian		No	Supported
tr	Turkish	Türkçe	No	Supported
ja	Japanese		No	Supported
ko	Korean		No	Supported
zh-CN	Chinese (Simplified)		No	Supported
zh-TW	Chinese (Traditional)		No	Supported
ar	Arabic		Yes	Supported
hi	Hindi		No	Supported
th	Thai		No	Supported
vi	Vietnamese	Tiếng Việt	No	Supported

System Architecture

RADIANT v4.7.0 LOCALIZATION ARCHITECTURE

BUILD TIME (Prevention)

ESLint Plugin: @radiant/eslint-plugin-i18n

```
BLOCKS: <Button>Submit</Button>
BLOCKS: throw new Error("Invalid input")
BLOCKS: { message: "User not found" }
ALLOWS: <Button>{t('button.submit')}</Button>
ALLOWS: throw new LocalizedError('error.invalid_input')
```

RUNTIME (Database-Driven)

PostgreSQL: localization_registry + localization_translations
All strings registered with unique keys
Translations for 18 languages
Status tracking (approved, ai_translated, needs_review)
Version history for all changes

AUTO-TRANSLATION (AI-Powered)

Lambda: localization-translate
Triggered when new registry entry added
Uses AWS Bedrock (Claude) for translation
Sets status = 'ai_translated' (NOT approved)
Sends notification to admin for review

ADMIN REVIEW (Human Approval)

Admin Dashboard: /admin/localization
View all strings needing review
Edit translations inline
Approve AI translations
Translation coverage dashboard
Bulk operations (approve all, export, import)

41.2 DATABASE SCHEMA

Migration: 041_localization_system.sql

```
-- =====
-- RADIANT v4.7.0 - Complete Internationalization System
-- Migration: 041_localization_system.sql
-- =====

-- =====
-- 41.2.1 Supported Languages Table
-- =====

CREATE TABLE localization_languages (
    code VARCHAR(10) PRIMARY KEY,      -- ISO 639-1 + region (e.g., 'en', 'zh-CN')
    name VARCHAR(100) NOT NULL,        -- English name
    native_name VARCHAR(100) NOT NULL,  -- Name in native script
    is_rtl BOOLEAN DEFAULT false,     -- Right-to-left language
    is_active BOOLEAN DEFAULT true,    -- Available for selection
    display_order INTEGER DEFAULT 100,
    -- Metadata
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Insert supported languages
INSERT INTO localization_languages (code, name, native_name, is_rtl, display_order) VALUES
('en', 'English', 'English', false, 1),
('es', 'Spanish', 'Español', false, 2),
('fr', 'French', 'Français', false, 3),
('de', 'German', 'Deutsch', false, 4),
('pt', 'Portuguese', 'Português', false, 5),
('it', 'Italian', 'Italiano', false, 6),
('nl', 'Dutch', 'Nederlands', false, 7),
('pl', 'Polish', 'Polski', false, 8),
('ru', 'Russian', 'Русский', false, 9),
('tr', 'Turkish', 'Türkçe', false, 10),
('ja', 'Japanese', '日本語', false, 11),
('ko', 'Korean', '한국어', false, 12),
('zh-CN', 'Chinese (Simplified)', '中文', false, 13),
('zh-TW', 'Chinese (Traditional)', '中文', false, 14),
('ar', 'Arabic', 'العربية', true, 15),
('hi', 'Hindi', 'हिन्दी', false, 16),
('th', 'Thai', 'ไทย', false, 17),
('vi', 'Vietnamese', 'Tiếng Việt', false, 18);
-- =====
```

```

-- 41.2.2 Localization Registry (Master String List)
-- =====

CREATE TABLE localization_registry (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- String identification
    key VARCHAR(255) NOT NULL UNIQUE,      -- Unique key like 'button.submit', 'error.user_not_found'
    default_text TEXT NOT NULL,           -- English default text

    -- Categorization
    category VARCHAR(100) NOT NULL,        -- 'ui.buttons', 'errors.validation', 'messages.success'
    subcategory VARCHAR(100),               -- Optional further grouping

    -- Context for translators
    context TEXT,                         -- Description/usage context for translators
    max_length INTEGER,                   -- Character limit if applicable
    placeholders JSONB DEFAULT '[]',       -- List of {name, description} for interpolation

    -- Source tracking
    source_app VARCHAR(50) NOT NULL,        -- 'admin', 'thinktank', 'api', 'shared'
    source_file VARCHAR(255),               -- Original file path where first used

    -- Status
    is_active BOOLEAN DEFAULT true,
    deprecated_at TIMESTAMPTZ,
    deprecated_replacement_key VARCHAR(255),

    -- Timestamps
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    created_by UUID REFERENCES administrators(id),

    -- Indexes for fast lookup
    CONSTRAINT valid_key_format CHECK (key ~ '^[a-z][a-z0-9_.]+[a-z0-9]$')
);

CREATE INDEX idx_localization_registry_key ON localization_registry(key);
CREATE INDEX idx_localization_registry_category ON localization_registry(category);
CREATE INDEX idx_localization_registry_source_app ON localization_registry(source_app);
CREATE INDEX idx_localization_registry_active ON localization_registry(is_active) WHERE is_acti

-- =====
-- 41.2.3 Localization Translations (Per-Language Values)
-- =====

CREATE TYPE translation_status AS ENUM (
    'pending',          -- Not yet translated

```

```

'ai_translated',    -- Auto-translated by AI, needs review
'in_review',        -- Being reviewed by human
'approved',         -- Human-approved for production
'rejected'          -- Rejected, needs re-translation
);

CREATE TABLE localization_translations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Foreign keys
    registry_id UUID NOT NULL REFERENCES localization_registry(id) ON DELETE CASCADE,
    language_code VARCHAR(10) NOT NULL REFERENCES localization_languages(code),

    -- Translation content
    translated_text TEXT NOT NULL,

    -- Status tracking
    status translation_status NOT NULL DEFAULT 'pending',

    -- AI translation metadata
    ai_model VARCHAR(100),           -- e.g., 'anthropic.claude-3-sonnet-20240229-v1:0'
    ai_confidence DECIMAL(3,2),       -- 0.00 to 1.00
    ai_translated_at TIMESTAMPTZ,

    -- Human review
    reviewed_by UUID REFERENCES administrators(id),
    reviewed_at TIMESTAMPTZ,
    review_notes TEXT,

    -- Version tracking
    version INTEGER NOT NULL DEFAULT 1,
    previous_text TEXT,             -- Previous translation for comparison

    -- Timestamps
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    -- Unique constraint
    UNIQUE(registry_id, language_code)
);

CREATE INDEX idx_localization_translations_registry ON localization_translations(registry_id);
CREATE INDEX idx_localization_translations_language ON localization_translations(language_code);
CREATE INDEX idx_localization_translations_status ON localization_translations(status);
CREATE INDEX idx_localization_translations_needs_review
    ON localization_translations(status)
    WHERE status IN ('ai_translated', 'in_review');

```

```

-- =====
-- 4.1.2.4 Audit Log for Translation Changes
-- =====

CREATE TABLE localization_audit_log (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- What changed
    registry_id UUID REFERENCES localization_registry(id) ON DELETE SET NULL,
    translation_id UUID REFERENCES localization_translations(id) ON DELETE SET NULL,
    language_code VARCHAR(10),

    -- Change details
    action VARCHAR(50) NOT NULL, -- 'created', 'updated', 'approved', 'rejected', 'ai_translation'
    old_value JSONB,
    new_value JSONB,

    -- Who made the change
    changed_by UUID REFERENCES administrators(id),
    changed_by_system BOOLEAN DEFAULT false, -- True if AI/system change

    -- Timestamps
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_localization_audit_registry ON localization_audit_log(registry_id);
CREATE INDEX idx_localization_audit_created ON localization_audit_log(created_at DESC);

-- =====
-- 4.1.2.5 Translation Queue (For AI Processing)
-- =====

CREATE TABLE localization_translation_queue (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    registry_id UUID NOT NULL REFERENCES localization_registry(id) ON DELETE CASCADE,
    language_code VARCHAR(10) NOT NULL REFERENCES localization_languages(code),

    -- Queue status
    status VARCHAR(20) NOT NULL DEFAULT 'pending'
        CHECK (status IN ('pending', 'processing', 'completed', 'failed')),

    -- Processing metadata
    attempts INTEGER DEFAULT 0,
    last_attempt_at TIMESTAMPTZ,
    error_message TEXT,

    -- Timestamps

```

```

created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
completed_at TIMESTAMPTZ,
UNIQUE(registry_id, language_code)
);

CREATE INDEX idx_translation_queue_pending ON localization_translation_queue(status)
WHERE status = 'pending';

-- =====
-- 4.1.2.6 Helper Functions
-- =====

-- Get translation with fallback chain: requested language -> English -> key
CREATE OR REPLACE FUNCTION get_translation(
    p_key VARCHAR(255),
    p_language VARCHAR(10) DEFAULT 'en'
) RETURNS TEXT AS $$

DECLARE
    v_result TEXT;
BEGIN
    -- Try requested language first
    SELECT lt.translated_text INTO v_result
    FROM localization_registry lr
    JOIN localization_translations lt ON lt.registry_id = lr.id
    WHERE lr.key = p_key
        AND lt.language_code = p_language
        AND lt.status = 'approved'
        AND lr.is_active = true;

    IF v_result IS NOT NULL THEN
        RETURN v_result;
    END IF;

    -- Fall back to English
    IF p_language != 'en' THEN
        SELECT lt.translated_text INTO v_result
        FROM localization_registry lr
        JOIN localization_translations lt ON lt.registry_id = lr.id
        WHERE lr.key = p_key
            AND lt.language_code = 'en'
            AND lt.status = 'approved'
            AND lr.is_active = true;

        IF v_result IS NOT NULL THEN
            RETURN v_result;
        END IF;
    END IF;

```

```

-- Fall back to default_text from registry
SELECT lr.default_text INTO v_result
FROM localization_registry lr
WHERE lr.key = p_key AND lr.is_active = true;

-- Return key if nothing found
RETURN COALESCE(v_result, p_key);
END;
$$ LANGUAGE plpgsql STABLE;

-- Get all translations for a language (for bulk loading)
CREATE OR REPLACE FUNCTION get_all_translations(
    p_language VARCHAR(10) DEFAULT 'en'
) RETURNS TABLE (
    key VARCHAR(255),
    text TEXT,
    is_fallback BOOLEAN
) AS $$

BEGIN
    RETURN QUERY
    SELECT
        lr.key,
        COALESCE(
            lt_lang.translated_text,
            lt_en.translated_text,
            lr.default_text
        ) as text,
        (lt_lang.translated_text IS NULL) as is_fallback
    FROM localization_registry lr
    LEFT JOIN localization_translations lt_lang
        ON lt_lang.registry_id = lr.id
        AND lt_lang.language_code = p_language
        AND lt_lang.status = 'approved'
    LEFT JOIN localization_translations lt_en
        ON lt_en.registry_id = lr.id
        AND lt_en.language_code = 'en'
        AND lt_en.status = 'approved'
    WHERE lr.is_active = true;
END;
$$ LANGUAGE plpgsql STABLE;

-- Get translation coverage statistics
CREATE OR REPLACE FUNCTION get_translation_coverage()
RETURNS TABLE (
    language_code VARCHAR(10),
    language_name VARCHAR(100),
    total_strings BIGINT,

```

```

translated_count BIGINT,
approved_count BIGINT,
ai_translated_count BIGINT,
pending_count BIGINT,
coverage_percent DECIMAL(5,2)
) AS $$

BEGIN
RETURN QUERY
WITH total AS (
    SELECT COUNT(*) as cnt FROM localization_registry WHERE is_active = true
)
SELECT
    ll.code as language_code,
    ll.name as language_name,
    t.cnt as total_strings,
    COUNT(lt.id) as translated_count,
    COUNT(lt.id) FILTER (WHERE lt.status = 'approved') as approved_count,
    COUNT(lt.id) FILTER (WHERE lt.status = 'ai_translated') as ai_translated_count,
    t.cnt - COUNT(lt.id) as pending_count,
    ROUND((COUNT(lt.id) FILTER (WHERE lt.status = 'approved'))::DECIMAL / NULLIF(t.cnt, 0))
FROM localization_languages ll
CROSS JOIN total t
LEFT JOIN localization_registry lr ON lr.is_active = true
LEFT JOIN localization_translations lt
    ON lt.registry_id = lr.id
    AND lt.language_code = ll.code
WHERE ll.is_active = true
GROUP BY ll.code, ll.name, ll.display_order, t.cnt
ORDER BY ll.display_order;
END;

$$ LANGUAGE plpgsql STABLE;

-- Trigger to auto-queue translations when new registry entry added
CREATE OR REPLACE FUNCTION queue_translations_for_new_entry()
RETURNS TRIGGER AS $$

BEGIN
    -- Queue translations for all active languages except English
    INSERT INTO localization_translation_queue (registry_id, language_code)
    SELECT NEW.id, ll.code
    FROM localization_languages ll
    WHERE ll.is_active = true AND ll.code != 'en';

    -- Auto-create English translation from default_text
    INSERT INTO localization_translations (registry_id, language_code, translated_text, status)
    VALUES (NEW.id, 'en', NEW.default_text, 'approved');

    RETURN NEW;
END;

```

```

$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_queue_translations
    AFTER INSERT ON localization_registry
    FOR EACH ROW
    EXECUTE FUNCTION queue_translations_for_new_entry();

-- Trigger to log translation changes
CREATE OR REPLACE FUNCTION log_translation_changes()
RETURNS TRIGGER AS $$

BEGIN
    INSERT INTO localization_audit_log (
        registry_id,
        translation_id,
        language_code,
        action,
        old_value,
        new_value,
        changed_by,
        changed_by_system
    ) VALUES (
        NEW.registry_id,
        NEW.id,
        NEW.language_code,
        CASE
            WHEN TG_OP = 'INSERT' THEN 'created'
            WHEN OLD.status != NEW.status AND NEW.status = 'approved' THEN 'approved'
            WHEN OLD.status != NEW.status AND NEW.status = 'rejected' THEN 'rejected'
            ELSE 'updated'
        END,
        CASE WHEN TG_OP = 'UPDATE' THEN jsonb_build_object(
            'text', OLD.translated_text,
            'status', OLD.status
        ) END,
        jsonb_build_object(
            'text', NEW.translated_text,
            'status', NEW.status
        ),
        NEW.reviewed_by,
        NEW.ai_model IS NOT NULL AND NEW.reviewed_by IS NULL
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_log_translation_changes
    AFTER INSERT OR UPDATE ON localization_translations
    FOR EACH ROW

```

```
EXECUTE FUNCTION log_translation_changes();
```

41.3 TYPESCRIPT TYPES & CONSTANTS

File: packages/shared/src/i18n/types.ts

```
// =====
// RADIANT v4.7.0 - Internationalization Types
// =====

/** 
 * Supported language codes
 */
export const SUPPORTED_LANGUAGES = [
  'en', 'es', 'fr', 'de', 'pt', 'it', 'nl', 'pl', 'ru', 'tr',
  'ja', 'ko', 'zh-CN', 'zh-TW', 'ar', 'hi', 'th', 'vi'
] as const;

export type LanguageCode = typeof SUPPORTED_LANGUAGES[number];

export const DEFAULT_LANGUAGE: LanguageCode = 'en';

export const RTL_LANGUAGES: LanguageCode[] = ['ar'];

/** 
 * Language metadata
 */
export interface Language {
  code: LanguageCode;
  name: string;
  nativeName: string;
  isRtl: boolean;
  isActive: boolean;
  displayOrder: number;
}

/** 
 * Translation status enum
 */
export type TranslationStatus =
  | 'pending'
  | 'ai_translated'
  | 'in_review'
  | 'approved'
  | 'rejected';

/**
```

```

* Localization registry entry
*/
export interface LocalizationEntry {
  id: string;
  key: string;
  defaultText: string;
  category: string;
  subcategory?: string;
  context?: string;
  maxLength?: number;
  placeholders?: Array<{
    name: string;
    description: string;
  }>;
  sourceApp: 'admin' | 'thinktank' | 'api' | 'shared';
  sourceFile?: string;
  isActive: boolean;
  deprecatedAt?: string;
  deprecatedReplacementKey?: string;
  createdAt: string;
  updatedAt: string;
}

/**
* Translation for a specific language
*/
export interface Translation {
  id: string;
  registryId: string;
  languageCode: LanguageCode;
  translatedText: string;
  status: TranslationStatus;
  aiModel?: string;
  aiConfidence?: number;
  aiTranslatedAt?: string;
  reviewedBy?: string;
  reviewedAt?: string;
  reviewNotes?: string;
  version: number;
  previousText?: string;
  createdAt: string;
  updatedAt: string;
}

/**
* Translation with key (for client-side use)
*/
export interface TranslationBundle {

```

```

[key: string]: string;
}

/**
 * Translation coverage statistics
 */
export interface TranslationCoverage {
  languageCode: LanguageCode;
  languageName: string;
  totalStrings: number;
  translatedCount: number;
  approvedCount: number;
  aiTranslatedCount: number;
  pendingCount: number;
  coveragePercent: number;
}

/**
 * Interpolation values for placeholders
 */
export type InterpolationValues = Record<string, string | number>;
```

// Categories for organizing translations

```

export const TRANSLATION_CATEGORIES = {
  // UI Elements
  'ui.buttons': 'Buttons and Actions',
  'ui.labels': 'Form Labels',
  'ui.placeholders': 'Input Placeholders',
  'ui.tooltips': 'Tooltips and Hints',
  'ui.navigation': 'Navigation Items',
  'ui.headings': 'Page and Section Headings',

  // Messages
  'messages.success': 'Success Messages',
  'messages.info': 'Informational Messages',
  'messages.warning': 'Warning Messages',
  'messages.loading': 'Loading States',

  // Errors
  'errors.validation': 'Validation Errors',
  'errors.auth': 'Authentication Errors',
  'errors.api': 'API Errors',
  'errors.network': 'Network Errors',
  'errors.system': 'System Errors',

  // Features
}
```

```

'features.chat': 'Chat Feature',
'features.thinktank': 'Think Tank Feature',
'features.admin': 'Admin Dashboard',
'features.billing': 'Billing & Payments',
'features.settings': 'User Settings',

// Content
'content.legal': 'Legal Content',
'content.marketing': 'Marketing Copy',
'content.help': 'Help & Documentation',
} as const;

export type TranslationCategory = keyof typeof TRANSLATION_CATEGORIES;

```

File: packages/shared/src/i18n/constants.ts

```

// =====
// RADIANT v4.7.0 - i18n Constants
// =====

import { LanguageCode } from './types';

/**
 * Language metadata mapping
 */
export const LANGUAGE_METADATA: Record<LanguageCode, {
  name: string;
  nativeName: string;
  isRtl: boolean;
}> = {
  'en': { name: 'English', nativeName: 'English', isRtl: false },
  'es': { name: 'Spanish', nativeName: 'Español', isRtl: false },
  'fr': { name: 'French', nativeName: 'Français', isRtl: false },
  'de': { name: 'German', nativeName: 'Deutsch', isRtl: false },
  'pt': { name: 'Portuguese', nativeName: 'Português', isRtl: false },
  'it': { name: 'Italian', nativeName: 'Italiano', isRtl: false },
  'nl': { name: 'Dutch', nativeName: 'Nederlands', isRtl: false },
  'pl': { name: 'Polish', nativeName: 'Polski', isRtl: false },
  'ru': { name: 'Russian', nativeName: 'Русский', isRtl: false },
  'tr': { name: 'Turkish', nativeName: 'Türkçe', isRtl: false },
  'ja': { name: 'Japanese', nativeName: '日本語', isRtl: false },
  'ko': { name: 'Korean', nativeName: '한국어', isRtl: false },
  'zh-CN': { name: 'Chinese (Simplified)', nativeName: '中文', isRtl: false },
  'zh-TW': { name: 'Chinese (Traditional)', nativeName: '中文', isRtl: false },
  'ar': { name: 'Arabic', nativeName: 'العربية', isRtl: true },
  'hi': { name: 'Hindi', nativeName: 'हिन्दी', isRtl: false },
  'th': { name: 'Thai', nativeName: 'ไทย', isRtl: false },
  'vi': { name: 'Vietnamese', nativeName: 'Tiếng Việt', isRtl: false },
}

```

```

};

/**
 * Bedrock model for translations
 */
export const TRANSLATION_AI_MODEL = 'anthropic.claude-3-sonnet-20240229-v1:0';

/**
 * Translation system prompt
 */
export const TRANSLATION_SYSTEM_PROMPT = `You are a professional translator for a software app.
Your task is to translate UI text, error messages, and user-facing content.

Guidelines:
1. Maintain the same tone and formality level as the source
2. Preserve all placeholders like {name}, {count}, etc. exactly as they appear
3. Keep technical terms consistent with industry standards for the target language
4. For UI elements (buttons, labels), keep translations concise
5. Respect character limits if specified
6. Consider the context provided to ensure accurate translation
7. For RTL languages (Arabic), ensure text flows correctly

Output ONLY the translated text, nothing else.`;

```

```

/**
 * Rate limits for translation API
 */
export const TRANSLATION_RATE_LIMITS = {
  maxConcurrent: 10,
  maxPerMinute: 100,
  maxPerHour: 1000,
  retryAttempts: 3,
  retryDelayMs: 1000,
};

```

41.4 AI TRANSLATION LAMBDA

File: lambda/localization/translate.ts

```

// =====
// RADIANT v4.7.0 - AI Translation Lambda
// Triggered by SQS queue when new translation needed
// =====

import { SQSHandler, SQSEvent } from 'aws-lambda';
import {
  BedrockRuntimeClient,

```

```

    InvokeModelCommand
} from '@aws-sdk/client-bedrock-runtime';
import { Pool } from 'pg';
import { SNSClient, PublishCommand } from '@aws-sdk/client-sns';
import {
  TRANSLATION_AI_MODEL,
  TRANSLATION_SYSTEM_PROMPT,
  LANGUAGE_METADATA
} from '@radiant/shared/i18n';

const bedrock = new BedrockRuntimeClient({ region: process.env.AWS_REGION });
const sns = new SNSClient({ region: process.env.AWS_REGION });
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false },
});
}

interface TranslationQueueMessage {
  registryId: string;
  languageCode: string;
  key: string;
  defaultText: string;
  context?: string;
  maxLength?: number;
  placeholders?: Array<{ name: string; description: string }>;
}

export const handler: SQSHandler = async (event: SQSEvent) => {
  for (const record of event.Records) {
    const message: TranslationQueueMessage = JSON.parse(record.body);

    try {
      await translateAndStore(message);
    } catch (error) {
      console.error('Translation failed:', error);
      await updateQueueStatus(message.registryId, message.languageCode, 'failed', error.message);
      throw error; // Re-throw to trigger SQS retry
    }
  }
};

async function translateAndStore(message: TranslationQueueMessage): Promise<void> {
  const { registryId, languageCode, key, defaultText, context, maxLength, placeholders } = message;

  // Update queue status to processing
  await updateQueueStatus(registryId, languageCode, 'processing');

  // Build translation prompt
}

```

```

const targetLanguage = LANGUAGE_METADATA[languageCode as keyof typeof LANGUAGE_METADATA];

let userPrompt = `Translate the following English text to ${targetLanguage.name} (${targetLanguage.name})`;

Text to translate: "${defaultText}"`;

if (context) {
  userPrompt += `\n\nContext: ${context}`;
}

if (maxLength) {
  userPrompt += `\n\nMaximum length: ${maxLength} characters`;
}

if (placeholders && placeholders.length > 0) {
  userPrompt += `\n\nPlaceholders to preserve exactly:`;
  placeholders.forEach(p => {
    userPrompt += `\n- ${p.name}: ${p.description}`;
  });
}

// Call Bedrock
const response = await bedrock.send(new InvokeModelCommand({
  modelId: TRANSLATION_AI_MODEL,
  contentType: 'application/json',
  accept: 'application/json',
  body: JSON.stringify({
    anthropic_version: 'bedrock-2023-05-31',
    max_tokens: 1024,
    system: TRANSLATION_SYSTEM_PROMPT,
    messages: [
      { role: 'user', content: userPrompt }
    ],
  }),
}));

const responseBody = JSON.parse(new TextDecoder().decode(response.body));
const translatedText = responseBody.content[0].text.trim();

// Validate placeholders are preserved
if (placeholders) {
  for (const p of placeholders) {
    if (!translatedText.includes(`-${p.name}`)) {
      throw new Error(`Translation missing placeholder: ${p.name}`);
    }
  }
}

```

```

// Store translation
await pool.query(`

    INSERT INTO localization_translations (
        registry_id, language_code, translated_text, status,
        ai_model, ai_confidence, ai_translated_at
    ) VALUES ($1, $2, $3, 'ai_translated', $4, $5, NOW())
    ON CONFLICT (registry_id, language_code)
    DO UPDATE SET
        translated_text = $3,
        status = 'ai_translated',
        ai_model = $4,
        ai_confidence = $5,
        ai_translated_at = NOW(),
        previous_text = localization_translations.translated_text,
        version = localization_translations.version + 1,
        updated_at = NOW()
    `, [registryId, languageCode, translatedText, TRANSLATION_AI_MODEL, 0.85]);

// Update queue status
await updateQueueStatus(registryId, languageCode, 'completed');

// Notify admin of new AI translation needing review
await notifyAdminOfNewTranslation(key, languageCode, translatedText);
}

async function updateQueueStatus(
    registryId: string,
    languageCode: string,
    status: string,
    errorMessage?: string
): Promise<void> {
    await pool.query(`

        UPDATE localization_translation_queue
        SET status = $3,
            last_attempt_at = NOW(),
            attempts = attempts + 1,
            error_message = $4,
            completed_at = CASE WHEN $3 = 'completed' THEN NOW() ELSE NULL END
        WHERE registry_id = $1 AND language_code = $2
    `, [registryId, languageCode, status, errorMessage]);
}

async function notifyAdminOfNewTranslation(
    key: string,
    languageCode: string,
    translatedText: string
): Promise<void> {
    const targetLanguage = LANGUAGE_METADATA[languageCode as keyof typeof LANGUAGE_METADATA];
}

```

```

    await sns.send(new PublishCommand({
      TopicArn: process.env.ADMIN_NOTIFICATION_TOPIC_ARN,
      Subject: `[RADIANT] New AI Translation Needs Review`,
      Message: JSON.stringify({
        type: 'translation_review_needed',
        key,
        languageCode,
        languageName: targetLanguage.name,
        translatedText: translatedText.substring(0, 200) + (translatedText.length > 200 ? '...' : ''),
        dashboardUrl: `${process.env.ADMIN_URL}/localization?filter=ai_translated`,
      }),
      MessageAttributes: {
        notificationType: {
          DataType: 'String',
          StringValue: 'translation_review',
        },
      },
    }));
  }
}

```

File: lambda/localization/process-queue.ts

```

// =====
// RADIANT v4.7.0 - Translation Queue Processor
// Scheduled Lambda to process pending translations
// =====

import { ScheduledHandler } from 'aws-lambda';
import { SQSClient, SendMessageCommand } from '@aws-sdk/client-sqs';
import { Pool } from 'pg';

const sqs = new SQSClient({ region: process.env.AWS_REGION });
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false },
});

export const handler: ScheduledHandler = async () => {
  // Get pending translations (limit batch size)
  const result = await pool.query(`

    SELECT
      q.registry_id,
      q.language_code,
      r.key,
      r.default_text,
      r.context,
      r.max_length,
  
```

```

    r.placeholders
    FROM localization_translation_queue q
    JOIN localization_registry r ON r.id = q.registry_id
    WHERE q.status = 'pending'
        AND q.attempts < 3
        AND r.is_active = true
    ORDER BY q.created_at ASC
    LIMIT 50
`);

console.log(`Processing ${result.rows.length} pending translations`);

for (const row of result.rows) {
    await sqs.send(new SendMessageCommand({
        QueueUrl: process.env.TRANSLATION_QUEUE_URL,
        MessageBody: JSON.stringify({
            registryId: row.registry_id,
            languageCode: row.language_code,
            key: row.key,
            defaultText: row.default_text,
            context: row.context,
            maxLength: row.max_length,
            placeholders: row.placeholders,
        }),
        MessageGroupId: row.language_code, // FIFO queue grouping
        MessageDuplicationId: `${row.registry_id}-${row.language_code}-${Date.now()}`,
    }));
}

return {
    processed: result.rows.length,
};
}

```

41.5 LOCALIZATION SERVICE (API)

File: `lambda/localization/api.ts`

```

// =====
// RADIANT v4.7.0 - Localization API Lambda
// =====

import { APIGatewayProxyHandler, APIGatewayProxyResult } from 'aws-lambda';
import { Pool } from 'pg';
import { extractAuthContext, requireRoles } from '../shared/auth';
import {
    LocalizationEntry,

```

```

Translation,
TranslationCoverage,
LanguageCode,
SUPPORTED_LANGUAGES
} from '@radiant/shared/i18n';

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false },
});

export const handler: APIGatewayProxyHandler = async (event): Promise<APIGatewayProxyResult> =>
{
  const { httpMethod, path, pathParameters, queryStringParameters, body } = event;

  try {
    // Public endpoint: get translations bundle
    if (httpMethod === 'GET' && path.match(/\/localization\/bundle\/[a-z-]+$/)) {
      return await getTranslationBundle(pathParameters?.language as LanguageCode);
    }

    // All other endpoints require authentication
    const auth = extractAuthContext(event);

    // GET /localization/languages - Get supported languages
    if (httpMethod === 'GET' && path === '/localization/languages') {
      return await getLanguages();
    }

    // GET /localization/coverage - Get translation coverage stats
    if (httpMethod === 'GET' && path === '/localization/coverage') {
      requireRoles(auth, ['admin', 'super_admin']);
      return await getCoverage();
    }

    // GET /localization/registry - List all registry entries
    if (httpMethod === 'GET' && path === '/localization/registry') {
      requireRoles(auth, ['admin', 'super_admin']);
      return await listRegistry(queryStringParameters);
    }

    // POST /localization/registry - Create new registry entry
    if (httpMethod === 'POST' && path === '/localization/registry') {
      requireRoles(auth, ['admin', 'super_admin']);
      return await createRegistryEntry(JSON.parse(body || '{}'), auth.userId);
    }

    // PUT /localization/registry/:id - Update registry entry
    if (httpMethod === 'PUT' && path.match(/\/localization\/registry\/[a-f0-9-]+$/)) {

```

```

    requireRoles(auth, ['admin', 'super_admin']);
    return await updateRegistryEntry(pathParameters?.id!, JSON.parse(body || '{}'));
}

// GET /localization/translations/:registryId - Get translations for entry
if (httpMethod === 'GET' && path.match(/\/localization\/translations\/[a-f0-9-]+$/)) {
    requireRoles(auth, ['admin', 'super_admin']);
    return await getTranslations(pathParameters?.registryId!);
}

// PUT /localization/translations/:id - Update translation
if (httpMethod === 'PUT' && path.match(/\/localization\/translations\/[a-f0-9-]+$/)) {
    requireRoles(auth, ['admin', 'super_admin']);
    return await updateTranslation(pathParameters?.id!, JSON.parse(body || '{}'), auth.userId);
}

// POST /localization/translations/:id/approve - Approve translation
if (httpMethod === 'POST' && path.match(/\/localization\/translations\/[a-f0-9-]+\\/approve$/)) {
    requireRoles(auth, ['admin', 'super_admin']);
    return await approveTranslation(pathParameters?.id!, auth.userId);
}

// POST /localization/translations/:id/reject - Reject translation
if (httpMethod === 'POST' && path.match(/\/localization\/translations\/[a-f0-9-]+\\/reject$/)) {
    requireRoles(auth, ['admin', 'super_admin']);
    return await rejectTranslation(pathParameters?.id!, JSON.parse(body || '{}'), auth.userId);
}

// GET /localization/pending - Get translations needing review
if (httpMethod === 'GET' && path === '/localization/pending') {
    requireRoles(auth, ['admin', 'super_admin']);
    return await getPendingReviews(queryStringParameters);
}

// POST /localization/bulk-approve - Bulk approve translations
if (httpMethod === 'POST' && path === '/localization/bulk-approve') {
    requireRoles(auth, ['super_admin']);
    return await bulkApprove(JSON.parse(body || '{}'), auth.userId);
}

    return { statusCode: 404, body: JSON.stringify({ error: 'Not found' }) };
} catch (error) {
    console.error('Localization API error:', error);
    return {
        statusCode: error.statusCode || 500,
        body: JSON.stringify({ error: error.message }),
    };
}
}

```

```

};

// Get translation bundle for client
async function getTranslationBundle(language: LanguageCode): Promise<APIGatewayProxyResult> {
  if (!SUPPORTED_LANGUAGES.includes(language)) {
    return { statusCode: 400, body: JSON.stringify({ error: 'Unsupported language' }) };
  }

  const result = await pool.query(`SELECT * FROM get_all_translations($1)`, [language]);

  const bundle: Record<string, string> = {};
  for (const row of result.rows) {
    bundle[row.key] = row.text;
  }

  return {
    statusCode: 200,
    headers: {
      'Content-Type': 'application/json',
      'Cache-Control': 'public, max-age=300', // Cache for 5 minutes
    },
    body: JSON.stringify(bundle),
  };
}

async function getLanguages(): Promise<APIGatewayProxyResult> {
  const result = await pool.query(`
    SELECT code, name, native_name, is_rtl, is_active, display_order
    FROM localization_languages
    WHERE is_active = true
    ORDER BY display_order
  `);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows.map(row => ({
      code: row.code,
      name: row.name,
      nativeName: row.native_name,
      isRtl: row.is_rtl,
      displayOrder: row.display_order,
    }))),
  };
}

async function getCoverage(): Promise<APIGatewayProxyResult> {
  const result = await pool.query(`SELECT * FROM get_translation_coverage()`);
}

```

```

    return {
      statusCode: 200,
      body: JSON.stringify(result.rows.map(row => ({
        languageCode: row.language_code,
        languageName: row.language_name,
        totalStrings: parseInt(row.total_strings),
        translatedCount: parseInt(row.translated_count),
        approvedCount: parseInt(row.approved_count),
        aiTranslatedCount: parseInt(row.ai_translated_count),
        pendingCount: parseInt(row.pending_count),
        coveragePercent: parseFloat(row.coverage_percent),
      })),),
    };
}

async function listRegistry(params: any): Promise<APIGatewayProxyResult> {
  const { category, sourceApp, search, page = '1', limit = '50' } = params || {};
  const offset = (parseInt(page) - 1) * parseInt(limit);

  let query = `
    SELECT lr.*,
      (SELECT COUNT(*) FROM localization_translations lt WHERE lt.registry_id = lr.id AND
      (SELECT COUNT(*) FROM localization_translations lt WHERE lt.registry_id = lr.id AND
      FROM localization_registry lr
      WHERE lr.is_active = true
    `;
  const queryParams: any[] = [];
  let paramIndex = 1;

  if (category) {
    query += ` AND lr.category = ${paramIndex++}`;
    queryParams.push(category);
  }

  if (sourceApp) {
    query += ` AND lr.source_app = ${paramIndex++}`;
    queryParams.push(sourceApp);
  }

  if (search) {
    query += ` AND (lr.key ILIKE ${paramIndex} OR lr.default_text ILIKE ${paramIndex})`;
    queryParams.push(`%${search}%`);
    paramIndex++;
  }

  query += ` ORDER BY lr.category, lr.key LIMIT ${paramIndex} OFFSET ${paramIndex}`;
  queryParams.push(parseInt(limit), offset);
}

```

```

const result = await pool.query(query, queryParams);

// Get total count
const countResult = await pool.query(`SELECT COUNT(*) FROM localization_registry WHERE is_active = true`);

return {
  statusCode: 200,
  body: JSON.stringify({
    data: result.rows,
    total: parseInt(countResult.rows[0].count),
    page: parseInt(page),
    limit: parseInt(limit),
  }),
};

}

async function createRegistryEntry(data: any, userId: string): Promise<APIGatewayProxyResult> {
  const { key, defaultText, category, subcategory, context, maxLength, placeholders, sourceApp }

  // Validate key format
  if (!/^[a-z][a-z0-9_.]+[a-z0-9]$/.test(key)) {
    return {
      statusCode: 400,
      body: JSON.stringify({ error: 'Invalid key format. Use lowercase with dots/underscores.' })
    };
  }

  const result = await pool.query(`INSERT INTO localization_registry (
    key, default_text, category, subcategory, context,
    max_length, placeholders, source_app, source_file, created_by
  ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10)
  RETURNING *
  `, [key, defaultText, category, subcategory, context, maxLength,
    JSON.stringify(placeholders || []), sourceApp, sourceFile, userId]);

  return {
    statusCode: 201,
    body: JSON.stringify(result.rows[0]),
  };
}

async function updateTranslation(id: string, data: any, userId: string): Promise<APIGatewayProxyResult> {
  const { translatedText, status } = data;

  const result = await pool.query(`UPDATE localization_registry SET
    translated_text = $1, status = $2
    WHERE id = $3
    `, [translatedText, status, id]);
}

```

```

    UPDATE localization_translations
    SET translated_text = COALESCE($2, translated_text),
        status = COALESCE($3, status),
        reviewed_by = $4,
        reviewed_at = NOW(),
        previous_text = translated_text,
        version = version + 1,
        updated_at = NOW()
    WHERE id = $1
    RETURNING *
    ` , [id, translatedText, status, userId]);
}

if (result.rows.length === 0) {
    return { statusCode: 404, body: JSON.stringify({ error: 'Translation not found' }) };
}

return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0]),
};
}

async function approveTranslation(id: string, userId: string): Promise<APIGatewayProxyResult> {
    const result = await pool.query(`

        UPDATE localization_translations
        SET status = 'approved',
            reviewed_by = $2,
            reviewed_at = NOW(),
            updated_at = NOW()
        WHERE id = $1
        RETURNING *
    ` , [id, userId]);

    if (result.rows.length === 0) {
        return { statusCode: 404, body: JSON.stringify({ error: 'Translation not found' }) };
    }

    return {
        statusCode: 200,
        body: JSON.stringify(result.rows[0]),
    };
}

async function rejectTranslation(id: string, data: any, userId: string): Promise<APIGatewayProxyResult> {
    const { reviewNotes } = data;

    const result = await pool.query(`

        UPDATE localization_translations

```

```

    SET status = 'rejected',
        reviewed_by = $2,
        reviewed_at = NOW(),
        review_notes = $3,
        updated_at = NOW()
    WHERE id = $1
    RETURNING *
    ` , [id, userId, reviewNotes]);
}

// Re-queue for translation
await pool.query(`
    INSERT INTO localization_translation_queue (registry_id, language_code)
    SELECT registry_id, language_code FROM localization_translations WHERE id = $1
    ON CONFLICT (registry_id, language_code) DO UPDATE SET
        status = 'pending',
        attempts = 0,
        error_message = NULL
` , [id]);

return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0]),
};
}

async function getPendingReviews(params: any): Promise<APIGatewayProxyResult> {
    const { language, page = '1', limit = '50' } = params || {};
    const offset = (parseInt(page) - 1) * parseInt(limit);

    let query = `
        SELECT lt.*, lr.key, lr.default_text, lr.context, lr.category,
            ll.name as language_name, ll.native_name
        FROM localization_translations lt
        JOIN localization_registry lr ON lr.id = lt.registry_id
        JOIN localization_languages ll ON ll.code = lt.language_code
        WHERE lt.status = 'ai_translated'
    `;
    const queryParams: any[] = [];
    let paramIndex = 1;

    if (language) {
        query += ` AND lt.language_code = ${paramIndex++}`;
        queryParams.push(language);
    }

    query += ` ORDER BY lt.ai_translated_at DESC LIMIT ${paramIndex++} OFFSET ${paramIndex}`;
    queryParams.push(parseInt(limit), offset);
}

```

```

const result = await pool.query(query, queryParams);

return {
  statusCode: 200,
  body: JSON.stringify(result.rows),
};

}

async function bulkApprove(data: any, userId: string): Promise<APIGatewayProxyResult> {
  const { translationIds } = data;

  if (!Array.isArray(translationIds) || translationIds.length === 0) {
    return { statusCode: 400, body: JSON.stringify({ error: 'translationIds required' }) };
  }

  const result = await pool.query(`

    UPDATE localization_translations
    SET status = 'approved',
        reviewed_by = $2,
        reviewed_at = NOW(),
        updated_at = NOW()
    WHERE id = ANY($1::uuid[])
    RETURNING id
  `, [translationIds, userId]);

  return {
    statusCode: 200,
    body: JSON.stringify({ approved: result.rows.length }),
  };
}

// Export individual functions for updateRegistryEntry and getTranslations
async function updateRegistryEntry(id: string, data: any): Promise<APIGatewayProxyResult> {
  const { defaultText, category, subcategory, context, maxLength, placeholders, isActive } = data;

  const result = await pool.query(`

    UPDATE localization_registry
    SET default_text = COALESCE($2, default_text),
        category = COALESCE($3, category),
        subcategory = COALESCE($4, subcategory),
        context = COALESCE($5, context),
        max_length = COALESCE($6, max_length),
        placeholders = COALESCE($7, placeholders),
        is_active = COALESCE($8, is_active),
        updated_at = NOW()
    WHERE id = $1
    RETURNING *
  `, [id, defaultText, category, subcategory, context, maxLength, placeholders, isActive, updated_at]);
}

```

```

    placeholders ? JSON.stringify(placeholders) : null, isActive]);
}

if (result.rows.length === 0) {
  return { statusCode: 404, body: JSON.stringify({ error: 'Registry entry not found' }) };
}

return {
  statusCode: 200,
  body: JSON.stringify(result.rows[0]),
};
}

async function getTranslations(registryId: string): Promise<APIGatewayProxyResult> {
  const result = await pool.query(`

    SELECT lt.*,
      ll.name as language_name,
      ll.native_name,
      ll.is_rtl
    FROM localization_translations lt
    JOIN localization_languages ll ON ll.code = lt.language_code
    WHERE lt.registry_id = $1
    ORDER BY ll.display_order
  `, [registryId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows),
  };
}

```

41.6 REACT i18n IMPLEMENTATION

File: packages/shared/src/i18n/react/I18nProvider.tsx

```

// =====
// RADIANT v4.7.0 - React i18n Provider
// =====

import React, {
  createContext,
  useContext,
  useState,
  useEffect,
  useCallback,
  ReactNode
} from 'react';
import {
  LanguageCode,
  TranslationBundle,
  InterpolationValues,

```

```

DEFAULT_LANGUAGE,
RTL_LANGUAGES,
LANGUAGE_METADATA
} from '../types';

interface I18nContextType {
  language: LanguageCode;
  setLanguage: (lang: LanguageCode) => void;
  t: (key: string, values?: InterpolationValues) => string;
  isRtl: boolean;
  isLoading: boolean;
  languages: typeof LANGUAGE_METADATA;
}

const I18nContext = createContext<I18nContextType | null>(null);

interface I18nProviderProps {
  children: ReactNode;
  defaultLanguage?: LanguageCode;
  apiBaseUrl: string;
}

export function I18nProvider({
  children,
  defaultLanguage = DEFAULT_LANGUAGE,
  apiBaseUrl
}: I18nProviderProps) {
  const [language, setLanguageState] = useState<LanguageCode>(() => {
    // Try to get from localStorage first
    if (typeof window !== 'undefined') {
      const stored = localStorage.getItem('radiant_language');
      if (stored && Object.keys(LANGUAGE_METADATA).includes(stored)) {
        return stored as LanguageCode;
      }
    }
    return defaultLanguage;
  });

  const [translations, setTranslations] = useState<TranslationBundle>({});

  const [isLoading, setIsLoading] = useState(true);

  // Load translations when language changes
  useEffect(() => {
    let cancelled = false;

    async function loadTranslations() {
      setIsLoading(true);
      try {

```

```

        const response = await fetch(`[${apiBaseUrl}]/localization/bundle/${language}`);
        if (!response.ok) throw new Error('Failed to load translations');
        const bundle = await response.json();
        if (!cancelled) {
            setTranslations(bundle);
        }
    } catch (error) {
        console.error('Failed to load translations:', error);
        // Keep existing translations on error
    } finally {
        if (!cancelled) {
            setIsLoading(false);
        }
    }
}

loadTranslations();

return () => {
    cancelled = true;
};
}, [language, apiBaseUrl]);

// Update document direction for RTL languages
useEffect(() => {
    if (typeof document !== 'undefined') {
        document.documentElement.dir = RTL_LANGUAGES.includes(language) ? 'rtl' : 'ltr';
        document.documentElement.lang = language;
    }
}, [language]);

const setLanguage = useCallback((lang: LanguageCode) => {
    setLanguageState(lang);
    if (typeof window !== 'undefined') {
        localStorage.setItem('radiant_language', lang);
    }
}, []);

// Translation function with interpolation
const t = useCallback((key: string, values?: InterpolationValues): string => {
    let text = translations[key] || key;

    // Interpolate values
    if (values) {
        Object.entries(values).forEach(([k, v]) => {
            text = text.replace(new RegExp(`\\${k}\\`), String(v));
        });
    }
}

```

```

        return text;
    }, [translations]);
}

const value: I18nContextType = {
    language,
    setLanguage,
    t,
    isRtl: RTL_LANGUAGES.includes(language),
    isLoading,
    languages: LANGUAGE_METADATA,
};

return (
    <I18nContext.Provider value={value}>
        {children}
    </I18nContext.Provider>
);
}

/**
 * Hook to access i18n context
 */
export function useI18n(): I18nContextType {
    const context = useContext(I18nContext);
    if (!context) {
        throw new Error('useI18n must be used within an I18nProvider');
    }
    return context;
}

/**
 * Hook for translation function only
 */
export function useTranslation() {
    const { t, language, isRtl } = useI18n();
    return { t, language, isRtl };
}

```

File: packages/shared/src/i18n/react/LanguageSelector.tsx

```

// =====
// RADIANT v4.7.0 - Language Selector Component
// =====

import React from 'react';
import { useI18n } from './I18nProvider';
import { LanguageCode } from '../types';

```

```

interface LanguageSelectorProps {
  className?: string;
  showNativeName?: boolean;
  compact?: boolean;
}

export function LanguageSelector({
  className = '',
  showNativeName = true,
  compact = false
}: LanguageSelectorProps) {
  const { language, setLanguage, languages } = useI18n();

  const handleChange = (e: React.ChangeEvent<HTMLSelectElement>) => {
    setLanguage(e.target.value as LanguageCode);
  };

  return (
    <select
      value={language}
      onChange={handleChange}
      className={`language-selector ${className}`}
      aria-label="Select language"
    >
      {Object.entries(languages).map(([code, meta]) => (
        <option key={code} value={code}>
          {compact
            ? code.toUpperCase()
            : showNativeName
              ? `${meta.nativeName} (${meta.name})`
              : meta.name
          }
        </option>
      ))}
    </select>
  );
}

```

File: packages/shared/src/i18n/react/Trans.tsx

```

// =====
// RADIANT v4.7.0 - Trans Component for complex translations
// =====

import React, { ReactNode } from 'react';
import { useTranslation } from './I18nProvider';
import { InterpolationValues } from '../types';

```

```

interface TransProps {
  i18nKey: string;
  values?: InterpolationValues;
  components?: Record<string, ReactNode>;
  fallback?: string;
}

/**
 * Trans component for translations with embedded React components
 *
 * Usage:
 * <Trans
 *   i18nKey="message.welcome"
 *   values={{ name: 'John' }}
 *   components={{ bold: <strong />, link: <a href="/profile" /> }}
 * />
 *
 * Translation: "Hello <bold>{name}</bold>! Visit your <link>profile</link>."
 */
export function Trans({ i18nKey, values, components, fallback }: TransProps) {
  const { t } = useTranslation();

  let text = t(i18nKey);

  // If key not found, use fallback
  if (text === i18nKey && fallback) {
    text = fallback;
  }

  // Interpolate values first
  if (values) {
    Object.entries(values).forEach(([k, v]) => {
      text = text.replace(new RegExp(`\\${k}`), String(v));
    });
  }

  // If no components, return plain text
  if (!components) {
    return <>{text}</>;
  }

  // Parse and replace component placeholders
  const parts: ReactNode[] = [];
  let remaining = text;
  let key = 0;

  Object.entries(components).forEach(([name, component]) => {

```

```

const regex = new RegExp(`<${name}>(.*)</${name}>`, 'g');
const newParts: ReactNode[] = [];
let lastIndex = 0;
let match;

while ((match = regex.exec(remaining)) !== null) {
    // Add text before the match
    if (match.index > lastIndex) {
        newParts.push(remaining.slice(lastIndex, match.index));
    }

    // Clone component with content
    if (React.isValidElement(component)) {
        newParts.push(
            React.cloneElement(component, { key: key++ }, match[1])
        );
    }
}

lastIndex = regex.lastIndex;
}

// Add remaining text
if (lastIndex < remaining.length) {
    newParts.push(remaining.slice(lastIndex));
}

remaining = newParts.join('');
});

return <>{remaining}</>;
}

```

41.7 ESLINT PLUGIN (HARDCODE PREVENTION)

File: packages/eslint-plugin-i18n/src/index.ts

```

// =====
// RADIANT v4.7.0 - ESLint Plugin to Prevent Hardcoded Strings
// =====

import { ESLintUtils, TSESTree } from '@typescript-eslint/utils';

const createRule = ESLintUtils.RuleCreator(
    (name) => `https://radiant.dev/eslint/${name}`
);

// Patterns that are allowed without translation

```

```

const ALLOWED_PATTERNS = [
  /^[a-z][a-z0-9_.]+[a-z0-9]$/, // Translation keys like 'button.submit'
  /^https?:\/\//, // URLs
  /^[A-Z_]+$/, // Constants like 'GET', 'POST'
  /^\d+$/, // Pure numbers
  /^[a-z]+\.[a-z]+$/, // File extensions like 'image.png'
  /^#[0-9a-fA-F]+$/, // Hex colors
  /^rgb/, // RGB colors
  /^data:/, // Data URLs
  /^[a-z-]+/[a-z-]+$/, // MIME types
  /^s*/$, // Whitespace only
];

// JSX attributes that commonly have hardcoded strings
const ALLOWED JSX_ATTRIBUTES = [
  'className', 'class', 'id', 'name', 'type', 'href', 'src', 'alt',
  'placeholder', 'title', 'aria-label', 'data-testid', 'key', 'role',
  'target', 'rel', 'method', 'action', 'encType', 'accept', 'pattern',
];

// Function names that accept translation keys
const TRANSLATION_FUNCTIONS = ['t', 'i18n', 'translate', 'L10n'];

export const noHardcodedStrings = createRule({
  name: 'no-hardcoded-strings',
  meta: {
    type: 'suggestion',
    docs: {
      description: 'Disallow hardcoded user-facing strings. Use translation keys instead.',
    },
    messages: {
      hardcodedString:
        'Hardcoded string "{{text}}" detected. Use translation: t(\'{\{suggestedKey}\}}\'',
      hardcodedJsxText:
        'Hardcoded JSX text "{{text}}" detected. Use: {t(\'{\{suggestedKey}\}}\'},',
    },
    schema: [
      {
        type: 'object',
        properties: {
          ignorePaths: {
            type: 'array',
            items: { type: 'string' },
          },
          ignorePatterns: {
            type: 'array',
            items: { type: 'string' },
          },
        },
      },
    ],
  },
});

```

```

        },
    },
],
},
defaultOptions: [{ ignorePaths: [], ignorePatterns: [] }],
create(context, [options]) {
  const filename = context.filename || context.getFilename();

  // Skip test files and config files
  if (
    filename.includes('.test.') ||
    filename.includes('.spec.') ||
    filename.includes('.config.') ||
    filename.includes('__tests__') ||
    filename.includes('__mocks__')
  ) {
    return {};
  }

  // Skip files in ignore paths
  if (options.ignorePaths?.some(path => filename.includes(path))) {
    return {};
  }

  function isAllowedString(value: string): boolean {
    // Check built-in patterns
    if (ALLOWED_PATTERNS.some(pattern => pattern.test(value))) {
      return true;
    }

    // Check custom ignore patterns
    if (options.ignorePatterns?.some(pattern => new RegExp(pattern).test(value))) {
      return true;
    }

    // Allow very short strings (likely not user-facing)
    if (value.length <= 2) {
      return true;
    }

    return false;
  }

  function generateSuggestedKey(text: string): string {
    // Generate a key based on the text
    const words = text.toLowerCase()
      .replace(/[^a-z0-9\s]/g, '')
      .split(/\s+/)

```

```

    .slice(0, 4)
    .join('_');
  return `ui.text.${words || 'untitled'}`;
}

function isInsideTranslationCall(node: TSESTree.Node): boolean {
  let parent = node.parent;
  while (parent) {
    if (
      parent.type === 'CallExpression' &&
      parent.callee.type === 'Identifier' &&
      TRANSLATION_FUNCTIONS.includes(parent.callee.name)
    ) {
      return true;
    }
    parent = parent.parent;
  }
  return false;
}

return {
  // Check JSX text content
  JSXText(node) {
    const text = node.value.trim();
    if (text && !isAllowedString(text)) {
      context.report({
        node,
        messageId: 'hardcodedJsxText',
        data: {
          text: text.substring(0, 30) + (text.length > 30 ? '...' : ''),
          suggestedKey: generateSuggestedKey(text),
        },
      });
    }
  },
  // Check string literals in JSX expressions
  'JSXExpressionContainer > Literal'(node: TSESTree.Literal) {
    if (typeof node.value !== 'string') return;
    const text = node.value.trim();

    if (text && !isAllowedString(text) && !isInsideTranslationCall(node)) {
      context.report({
        node,
        messageId: 'hardcodedString',
        data: {
          text: text.substring(0, 30) + (text.length > 30 ? '...' : ''),
          suggestedKey: generateSuggestedKey(text),
        },
      });
    }
  },
}

```

```

        },
    });
}
},

// Check JSX attribute values (only specific attributes)
'JSXAttribute > Literal'(node: TSESTree.Literal) {
    if (typeof node.value !== 'string') return;

    const parent = node.parent as TSESTree.JSXAttribute;
    const attrName = parent.name.type === 'JSXIdentifier'
        ? parent.name.name
        : '';

// Skip allowed attributes
    if (ALLOWED_JSX_ATTRIBUTES.includes(attrName)) {
        return;
    }

// Check attributes that should be translated
    const translatableAttrs = ['label', 'title', 'placeholder', 'aria-label', 'errorMessage'];
    if (translatableAttrs.includes(attrName)) {
        const text = node.value.trim();
        if (text && !isAllowedString(text)) {
            context.report({
                node,
                messageId: 'hardcodedString',
                data: {
                    text: text.substring(0, 30) + (text.length > 30 ? '...' : ''),
                    suggestedKey: generateSuggestedKey(text),
                },
            });
        }
    }
},

// Check error messages in throw statements
'ThrowStatement CallExpression'(node: TSESTree.CallExpression) {
    if (node.arguments.length === 0) return;

    const firstArg = node.arguments[0];
    if (firstArg.type === 'Literal' && typeof firstArg.value === 'string') {
        const text = firstArg.value.trim();
        if (text && !isAllowedString(text) && !isInsideTranslationCall(firstArg)) {
            context.report({
                node: firstArg,
                messageId: 'hardcodedString',
                data: {

```


File: `.eslintrc.js` (Project Root Addition)

```
// Add to existing ESLint config
module.exports = {
  // ... existing config
  plugins: [
    // ... existing plugins
    '@radiant/i18n',
  ],
  rules: {
    // ... existing rules
    '@radiant/i18n/no-hardcoded-strings': ['error', {
      ignorePaths: [
        'node_modules',
        '__tests__',
        '*.test.ts',
        '*.spec.ts',
        'migrations',
      ],
      ignorePatterns: [
        '^[A-Z][A-Z_]+$', // Constants
        '/api/' // API paths
      ],
    }],
  },
};
```

41.8 SWIFT LOCALIZATION SERVICE (THINK TANK APP)

File: `ThinkTank/Services/LocalizationService.swift`

```
// =====
// RADIANT v4.7.0 - Swift Localization Service
// =====

import Foundation
import Combine

/// Supported languages
enum SupportedLanguage: String, CaseIterable, Codable {
  case en = "en"
  case es = "es"
  case fr = "fr"
  case de = "de"
  case pt = "pt"
  case it = "it"
  case nl = "nl"
```

```

case pl = "pl"
case ru = "ru"
case tr = "tr"
case ja = "ja"
case ko = "ko"
case zhCN = "zh-CN"
case zhTW = "zh-TW"
case ar = "ar"
case hi = "hi"
case th = "th"
case vi = "vi"

var displayName: String {
    switch self {
        case .en: return "English"
        case .es: return "Español"
        case .fr: return "Français"
        case .de: return "Deutsch"
        case .pt: return "Português"
        case .it: return "Italiano"
        case .nl: return "Nederlands"
        case .pl: return "Polski"
        case .ru: return " "
        case .tr: return "Türkçe"
        case .ja: return " "
        case .ko: return " "
        case .zhCN: return " "
        case .zhTW: return " "
        case .ar: return " "
        case .hi: return " "
        case .th: return " "
        case .vi: return "Tiếng Việt"
    }
}

var isRTL: Bool {
    self == .ar
}

/// Get system preferred language or default to English
static var preferred: SupportedLanguage {
    let preferredIdentifier = Locale.preferredLanguages.first ?? "en"

    // Try exact match first
    if let exact = SupportedLanguage(rawValue: preferredIdentifier) {
        return exact
    }
}

```

```

// Try language code only (e.g., "en-US" -> "en")
let languageCode = String(preferredIdentifier.prefix(2))
if let lang = SupportedLanguage(rawValue: languageCode) {
    return lang
}

// Handle Chinese variants
if preferredIdentifier.hasPrefix("zh") {
    if preferredIdentifier.contains("Hans") || preferredIdentifier.contains("CN") {
        return .zhCN
    } else {
        return .zhTW
    }
}

return .en
}

/// Localization service singleton
@MainActor
final class LocalizationService: ObservableObject {
    static let shared = LocalizationService()

    @Published private(set) var currentLanguage: SupportedLanguage
    @Published private(set) var isLoading = false
    @Published private(set) var translations: [String: String] = [:]

    private let apiBaseURL: URL
    private var cancellables = Set<AnyCancellable>()

    private init() {
        // Load saved language or use system preferred
        if let savedLang = UserDefaults.standard.string(forKey: "radiant_language"),
           let lang = SupportedLanguage(rawValue: savedLang) {
            self.currentLanguage = lang
        } else {
            self.currentLanguage = .preferred
        }

        self.apiBaseURL = URL(string: Configuration.shared.apiDataURL)!

        // Load translations for current language
        Task {
            await loadTranslations()
        }
    }
}

```

```

/// Change current language
func setLanguage(_ language: SupportedLanguage) async {
    guard language != currentLanguage else { return }

    currentLanguage = language
    UserDefaults.standard.set(language.rawValue, forKey: "radian_language")

    await loadTranslations()
}

/// Load translations from API
func loadTranslations() async {
    isLoading = true
    defer { isLoading = false }

    do {
        let url = apiBaseURL.appendingPathComponent("localization/bundle/\(currentLanguage")
        let (data, response) = try await URLSession.shared.data(from: url)

        guard let httpResponse = response as? HTTPURLResponse,
              httpResponse.statusCode == 200 else {
            throw LocalizationError.networkError
        }

        let bundle = try JSONDecoder().decode([String: String].self, from: data)
        translations = bundle

        // Cache translations locally
        cacheTranslations(bundle)
    }

    } catch {
        print("Failed to load translations: \(error)")
        // Load from cache on error
        loadCachedTranslations()
    }
}

/// Get translated string for key
func translate(_ key: String, values: [String: Any] = [:]) -> String {
    var text = translations[key] ?? key

    // Interpolate values
    for (name, value) in values {
        text = text.replacingOccurrences(of: "{\(name)}", with: String(describing: value))
    }

    return text
}

```

```

    /// Shorthand translation function
    func t(_ key: String, _ values: [String: Any] = [:]) -> String {
        translate(key, values: values)
    }

    // MARK: - Caching

    private func cacheTranslations(_ bundle: [String: String]) {
        guard let data = try? JSONEncoder().encode(bundle) else { return }

        let cacheURL = getCacheURL()
        try? data.write(to: cacheURL)
    }

    private func loadCachedTranslations() {
        let cacheURL = getCacheURL()
        guard let data = try? Data(contentsOf: cacheURL),
              let bundle = try? JSONDecoder().decode([String: String].self, from: data) else {
            return
        }
        translations = bundle
    }

    private func getCacheURL() -> URL {
        let cacheDir = FileManager.default.urls(for: .cachesDirectory, in: .userDomainMask).first!
        return cacheDir.appendingPathComponent("translations_\\"(currentLanguage.rawValue).json")
    }
}

enum LocalizationError: Error {
    case networkError
    case decodingError
}

// MARK: - Property Wrapper for SwiftUI

@propertyWrapper
struct Localized: DynamicProperty {
    @ObservedObject private var service = LocalizationService.shared
    private let key: String
    private let values: [String: Any]

    init(_ key: String, values: [String: Any] = [:]) {
        self.key = key
        self.values = values
    }
}

```

```

    var wrappedValue: String {
        service.translate(key, values: values)
    }
}

// MARK: - SwiftUI Extensions

extension View {
    /// Apply RTL layout if current language is RTL
    func localizedLayout() -> some View {
        environment(\.layoutDirection,
                    LocalizationService.shared.currentLanguage.isRTL ? .rightToLeft : .leftToRight)
    }
}

// MARK: - String Extension

extension String {
    /// Translate this key
    var localized: String {
        LocalizationService.shared.translate(self)
    }

    /// Translate with values
    func localized(with values: [String: Any]) -> String {
        LocalizationService.shared.translate(self, values: values)
    }
}

```

File: ThinkTank/Views/Components/LocalizedText.swift

```

// =====
// RADIANT v4.7.0 - LocalizedText SwiftUI Component
// =====

import SwiftUI

/// SwiftUI component for localized text
struct LocalizedText: View {
    @ObservedObject private var localization = LocalizationService.shared

    let key: String
    let values: [String: Any]

    init(_ key: String, values: [String: Any] = [:]) {
        self.key = key
        self.values = values
    }
}

```

```

var body: some View {
    Text(localization.translate(key, values))
}
}

/// Localized button with automatic translation
struct LocalizedButton: View {
    @ObservedObject private var localization = LocalizationService.shared

    let key: String
    let action: () -> Void

    init(_ key: String, action: @escaping () -> Void) {
        self.key = key
        self.action = action
    }

    var body: some View {
        Button(action: action) {
            Text(localization.translate(key))
        }
    }
}

/// Language selector view
struct LanguageSelector: View {
    @ObservedObject private var localization = LocalizationService.shared
    @State private var showingPicker = false

    var body: some View {
        Button {
            showingPicker = true
        } label: {
            HStack {
                Image(systemName: "globe")
                Text(localization.currentLanguage.displayName)
            }
        }
        .sheet(isPresented: $showingPicker) {
            NavigationStack {
                List(SupportedLanguage.allCases, id: \.self) { language in
                    Button {
                        Task {
                            await localization.setLanguage(language)
                        }
                        showingPicker = false
                    } label: {
                }
            }
        }
    }
}

```

```

        HStack {
            Text(language.displayName)
            Spacer()
            if language == localization.currentLanguage {
                Image(systemName: "checkmark")
                    .foregroundColor(.accentColor)
            }
        }
        .foregroundColor(.primary)
    }
    .navigationTitle("ui.settings.language".localized)
    .toolbar {
        ToolbarItem(placement: .cancellationAction) {
            Button("ui.buttons.cancel".localized) {
                showingPicker = false
            }
        }
    }
}
}

// MARK: - Preview

#Preview {
    VStack(spacing: 20) {
        LocalizedText("ui.buttons.submit")
        LocalizedText("messages.welcome", values: ["name": "John"])
        LocalizedButton("ui.buttons.save") {
            print("Saved!")
        }
        LanguageSelector()
    }
    .padding()
}

```

41.9 ADMIN DASHBOARD - LOCALIZATION MANAGEMENT

File: admin-dashboard/app/localization/page.tsx

```

// =====
// RADIANT v4.7.0 - Localization Management Dashboard
// =====

'use client';

```

```
import React, { useState, useEffect } from 'react';
import {
  Box,
  Typography,
  Tabs,
  Tab,
  Card,
  CardContent,
  Table,
  TableBody,
  TableCell,
  TableContainer,
  TableHead,
  TableRow,
  Paper,
  Chip,
  IconButton,
  Button,
  TextField,
  Select,
  MenuItem,
  Dialog,
 DialogTitle,
 DialogContent,
  DialogActions,
  LinearProgress,
  Alert,
  Tooltip,
  Badge,
} from '@mui/material';
import {
  Edit,
  Check,
  Close,
  Refresh,
  Search,
  Language,
  Warning,
  CheckCircle,
  AutoAwesome,
} from '@mui/icons-material';
import { useTranslation } from '@/hooks/useTranslation';
import { api } from '@/lib/api';

interface Translation {
  id: string;
  registryId: string;
}
```

```

languageCode: string;
translatedText: string;
status: 'pending' | 'ai_translated' | 'in_review' | 'approved' | 'rejected';
aiModel?: string;
aiConfidence?: number;
reviewedBy?: string;
reviewedAt?: string;
key: string;
defaultText: string;
context?: string;
category: string;
languageName: string;
nativeName: string;
}

interface TranslationCoverage {
  languageCode: string;
  languageName: string;
  totalStrings: number;
  translatedCount: number;
  approvedCount: number;
  aiTranslatedCount: number;
  pendingCount: number;
  coveragePercent: number;
}

export default function LocalizationPage() {
  const { t } = useTranslation();
  const [activeTab, setActiveTab] = useState(0);
  const [coverage, setCoverage] = useState<TranslationCoverage[]>([]);
  const [pendingReviews, setPendingReviews] = useState<Translation[]>([]);
  const [selectedLanguage, setSelectedLanguage] = useState<string>('all');
  const [searchQuery, setSearchQuery] = useState('');
  const [editDialog, setEditDialog] = useState<Translation | null>(null);
  const [editedText, setEditedText] = useState('');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    loadData();
  }, [selectedLanguage]);

  async function loadData() {
    setLoading(true);
    try {
      const [coverageRes, pendingRes] = await Promise.all([
        api.get('/localization/coverage'),
        api.get('/localization/pending', {
          params: { language: selectedLanguage !== 'all' ? selectedLanguage : undefined }
        })
      ]);
    
```

```

        },
    ]);
    setCoverage(coverageRes.data);
    setPendingReviews(pendingRes.data);
} catch (error) {
    console.error('Failed to load localization data:', error);
} finally {
    setLoading(false);
}
}

async function handleApprove(translation: Translation) {
    try {
        await api.post(`/localization/translations/${translation.id}/approve`);
        loadData();
    } catch (error) {
        console.error('Failed to approve translation:', error);
    }
}

async function handleReject(translation: Translation, notes: string) {
    try {
        await api.post(`/localization/translations/${translation.id}/reject`, {
            reviewNotes: notes,
        });
        loadData();
    } catch (error) {
        console.error('Failed to reject translation:', error);
    }
}

async function handleSaveEdit() {
    if (!editDialog) return;
    try {
        await api.put(`/localization/translations/${editDialog.id}`, {
            translatedText: editedText,
            status: 'approved',
        });
        setEditDialog(null);
        loadData();
    } catch (error) {
        console.error('Failed to save translation:', error);
    }
}

async function handleBulkApprove() {
    const aiTranslated = pendingReviews.filter(t => t.status === 'ai_translated');
    if (aiTranslated.length === 0) return;
}

```

```

if (!confirm(t('admin.localization.confirm_bulk_approve', { count: aiTranslated.length })))
  return;
}

try {
  await api.post('/localization/bulk-approve', {
    translationIds: aiTranslated.map(t => t.id),
  });
  loadData();
} catch (error) {
  console.error('Failed to bulk approve:', error);
}
}

const getStatusChip = (status: string) => {
  const statusConfig = {
    pending: { color: 'default' as const, icon: <Warning fontSize="small" /> },
    ai_translated: { color: 'warning' as const, icon: <AutoAwesome fontSize="small" /> },
    in_review: { color: 'info' as const, icon: <Search fontSize="small" /> },
    approved: { color: 'success' as const, icon: <CheckCircle fontSize="small" /> },
    rejected: { color: 'error' as const, icon: <Close fontSize="small" /> },
  };
  const config = statusConfig[status as keyof typeof statusConfig] || statusConfig.pending;
  return (
    <Chip
      size="small"
      label={t(`admin.localization.status.${status}`)}
      color={config.color}
      icon={config.icon}
    />
  );
};

const aiTranslatedCount = pendingReviews.filter(t => t.status === 'ai_translated').length;

return (
  <Box sx={{ p: 3 }}>
    <Box sx={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center', mb: 3 }}>
      <Typography variant="h4">
        <Language sx={{ mr: 1, verticalAlign: 'middle' }} />
        {t('admin.localization.title')}
      </Typography>
      <Button
        variant="outlined"
        startIcon={<Refresh />}
        onClick={loadData}
      >

```

```

        {t('admin.buttons.refresh')}
    </Button>
</Box>

{loading && <LinearProgress sx={{ mb: 2 }} />}

<Tabs value={activeTab} onChange={({_, v) => setActiveTab(v)} sx={{ mb: 3 }}>
    <Tab label={t('admin.localization.tabs.coverage')} />
    <Tab
        label={
            <Badge badgeContent={aiTranslatedCount} color="warning">
                {t('admin.localization.tabs.review')}
            </Badge>
        }
    />
    <Tab label={t('admin.localization.tabs.registry')} />
</Tabs>

{/* Coverage Tab */}
{activeTab === 0 && (
    <TableContainer component={Paper}>
        <Table>
            <TableHead>
                <TableRow>
                    <TableCell>{t('admin.localization.language')}</TableCell>
                    <TableCell align="right">{t('admin.localization.total')}</TableCell>
                    <TableCell align="right">{t('admin.localization.approved')}</TableCell>
                    <TableCell align="right">{t('admin.localization.ai_translated')}</TableCell>
                    <TableCell align="right">{t('admin.localization.pending')}</TableCell>
                    <TableCell>{t('admin.localization.coverage')}</TableCell>
                </TableRow>
            </TableHead>
            <TableBody>
                {coverage.map((lang) => (
                    <TableRow key={lang.languageCode}>
                        <TableCell>
                            <Box sx={{ display: 'flex', alignItems: 'center', gap: 1 }}>
                                <Typography fontWeight="medium">{lang.languageName}</Typography>
                                <Typography variant="caption" color="text.secondary">
                                    ({lang.languageCode})
                                </Typography>
                            </Box>
                        </TableCell>
                        <TableCell align="right">{lang.totalStrings}</TableCell>
                        <TableCell align="right">
                            <Chip size="small" color="success" label={lang.approvedCount} />
                        </TableCell>
                        <TableCell align="right">

```

```

        {lang.aiTranslatedCount > 0 && (
            <Chip size="small" color="warning" label={lang.aiTranslatedCount} />
        )}
    </TableCell>
    <TableCell align="right">
        {lang.pendingCount > 0 && (
            <Chip size="small" color="default" label={lang.pendingCount} />
        )}
    </TableCell>
    <TableCell>
        <Box sx={{ display: 'flex', alignItems: 'center', gap: 1 }}>
            <LinearProgress
                variant="determinate"
                value={lang.coveragePercent}
                sx={{ width: 100, height: 8, borderRadius: 4 }}
                color={lang.coveragePercent === 100 ? 'success' : 'primary'}
            />
            <Typography variant="body2">
                {lang.coveragePercent.toFixed(1)}%
            </Typography>
        </Box>
    </TableCell>
</TableRow>
))}
</TableBody>
</Table>
</TableContainer>
)}

/* Review Tab */
{activeTab === 1 && (
    <Box>
        <Box sx={{ display: 'flex', gap: 2, mb: 2 }}>
            <Select
                size="small"
                value={selectedLanguage}
                onChange={(e) => setSelectedLanguage(e.target.value)}
                sx={{ minWidth: 200 }}
            >
                <MenuItem value="all">{t('admin.localization.all_languages')}

```

```

placeholder={t('admin.localization.search_placeholder')}
value={searchQuery}
onChange={(e) => setSearchQuery(e.target.value)}
InputProps={{ startAdornment: <Search sx={{ mr: 1, color: 'text.secondary' }} />
/>
{aiTranslatedCount > 0 && (
<Button
variant="contained"
color="warning"
startIcon=<CheckCircle />
onClick={handleBulkApprove}
>
{t('admin.localization.bulk_approve', { count: aiTranslatedCount })}
</Button>
)}
)
</Box>

{pendingReviews.length === 0 ? (
<Alert severity="success">
{t('admin.localization.no_pending_reviews')}
</Alert>
) : (
<TableContainer component={Paper}>
<Table>
<TableHead>
<TableRow>
<TableCell>{t('admin.localization.key')}

```

```

<Typography variant="caption" color="text.secondary">
  {translation.category}
</Typography>
</TableCell>
<TableCell>
  <Typography variant="body2">
    {translation.defaultText}
  </Typography>
  {translation.context && (
    <Typography variant="caption" color="text.secondary">
      {translation.context}
    </Typography>
  )}
</TableCell>
<TableCell>
  <Typography
    variant="body2"
    dir={translation.languageCode === 'ar' ? 'rtl' : 'ltr'}
  >
    {translation.translatedText}
  </Typography>
  {translation.aiConfidence && (
    <Typography variant="caption" color="text.secondary">
      AI confidence: {(translation.aiConfidence * 100).toFixed(0)}%
    </Typography>
  )}
</TableCell>
<TableCell>
  <Chip
    size="small"
    label={translation.nativeName || translation.languageName}
  />
</TableCell>
<TableCell>
  {getStatusChip(translation.status)}
</TableCell>
<TableCell align="right">
  <Tooltip title={t('admin.buttons.edit')}>
    <IconButton
      size="small"
      onClick={() => {
        setEditDialog(translation);
        setEditedText(translation.translatedText);
      }}
    >
      <Edit fontSize="small" />
    </IconButton>
  </Tooltip>

```

```

        <Tooltip title={t('admin.buttons.approve')}>
          <IconButton
            size="small"
            color="success"
            onClick={() => handleApprove(translation)}
          >
            <Check fontSize="small" />
          </IconButton>
        </Tooltip>
        <Tooltip title={t('admin.buttons.reject')}>
          <IconButton
            size="small"
            color="error"
            onClick={() => {
              const notes = prompt(t('admin.localization.reject_reason'));
              if (notes) handleReject(translation, notes);
            }}
          >
            <Close fontSize="small" />
          </IconButton>
        </Tooltip>
      </TableCell>
    </TableRow>
  )));
</TableBody>
</Table>
</TableContainer>
)
</Box>
)
}

/* Edit Dialog */
<Dialog open={!editDialog} onClose={() => setEditDialog(null)} maxWidth="md" fullWidth>
  <DialogTitle>{t('admin.localization.edit_translation')}

```

```

<TextField
    fullWidth
    multiline
    rows={4}
    value={editedText}
    onChange={(e) => setEditedText(e.target.value)}
    label={t('admin.localization.translation')}
    sx={{ mt: 2 }}
    dir={editDialog.languageCode === 'ar' ? 'rtl' : 'ltr'}
/>
</Box>
)
</DialogContent>
<DialogActions>
<Button onClick={() => setEditDialog(null)}>
    {t('admin.buttons.cancel')}
</Button>
<Button onClick={handleSaveEdit} variant="contained" color="primary">
    {t('admin.buttons.save_and_approve')}
</Button>
</DialogActions>
</Dialog>
</Box>
);
}

```

41.10 CDK INFRASTRUCTURE

File: cdk/lib/localization-stack.ts

```

// =====
// RADIANT v4.7.0 - Localization Infrastructure Stack
// =====

import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as sqs from 'aws-cdk-lib/aws-sqs';
import * as sns from 'aws-cdk-lib/aws-sns';
import * as events from 'aws-cdk-lib/aws-events';
import * as targets from 'aws-cdk-lib/aws-events-targets';
import * as iam from 'aws-cdk-lib/aws-iam';
import { SqsEventSource } from 'aws-cdk-lib/aws-lambda-event-sources';
import { Construct } from 'constructs';

interface LocalizationStackProps extends cdk.StackProps {
    environment: string;
    vpcId: string;
}

```

```

    dbSecurityGroupId: string;
    dbHost: string;
    dbName: string;
    dbSecretArn: string;
    adminNotificationTopicArn: string;
    adminUrl: string;
}

export class LocalizationStack extends cdk.Stack {
    public readonly translationQueue: sqs.Queue;
    public readonly translateLambda: lambda.Function;
    public readonly apiLambda: lambda.Function;

    constructor(scope: Construct, id: string, props: LocalizationStackProps) {
        super(scope, id, props);

        // Dead letter queue for failed translations
        const dlq = new sqs.Queue(this, 'TranslationDLQ', {
            queueName: `radiant-${props.environment}-translation-dlq`,
            retentionPeriod: cdk.Duration.days(14),
        });

        // Translation queue (FIFO for ordering by language)
        this.translationQueue = new sqs.Queue(this, 'TranslationQueue', {
            queueName: `radiant-${props.environment}-translation-queue fifo`,
            fifo: true,
            contentBasedDeduplication: true,
            visibilityTimeout: cdk.Duration.minutes(5),
            deadLetterQueue: {
                queue: dlq,
                maxReceiveCount: 3,
            },
        });
    }

    // Lambda execution role with Bedrock access
    const lambdaRole = new iam.Role(this, 'LocalizationLambdaRole', {
        assumedBy: new iam.ServicePrincipal('lambda.amazonaws.com'),
        managedPolicies: [
            iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/AWSLambdaVPCAccessExecutionRole'),
        ],
    });

    // Bedrock permissions
    lambdaRole.addToPolicy(new iam.PolicyStatement({
        actions: [
            'bedrock:InvokeModel',
            'bedrock:InvokeModelWithResponseStream',
        ],
    }));
}

```

```

    resources: ['*'],
}));
```

// SNS permissions for notifications

```

lambdaRole.addToPolicy(new iam.PolicyStatement({
  actions: ['sns:Publish'],
  resources: [props.adminNotificationTopicArn],
}));
```

// Secrets Manager for DB credentials

```

lambdaRole.addToPolicy(new iam.PolicyStatement({
  actions: ['secretsmanager:GetSecretValue'],
  resources: [props.dbSecretArn],
}));
```

// Common Lambda environment - use DATABASE_URL for consistency

```

const lambdaEnvironment = {
  NODE_ENV: props.environment,
  DATABASE_URL: props.databaseUrl, // Connection string format
  ADMIN_NOTIFICATION_TOPIC_ARN: props.adminNotificationTopicArn,
  ADMIN_URL: props.adminUrl,
  TRANSLATION_QUEUE_URL: this.translationQueue.queueUrl,
};
```

// Translation Lambda (processes queue)

```

this.translateLambda = new lambda.Function(this, 'TranslateLambda', {
  functionName: `radiant-${props.environment}-localization-translate`,
  runtime: lambda.Runtime.NODEJS_20_X,
  handler: 'translate.handler',
  code: lambda.Code.fromAsset('lambda/localization'),
  timeout: cdk.Duration.minutes(2),
  memorySize: 512,
  role: lambdaRole,
  environment: lambdaEnvironment,
});
```

// Add SQS trigger

```

this.translateLambda.addEventSource(new SqsEventSource(this.translationQueue, {
  batchSize: 1,
  maxConcurrency: 10,
}));
```

// Queue processor Lambda (scheduled)

```

const queueProcessorLambda = new lambda.Function(this, 'QueueProcessorLambda', {
  functionName: `radiant-${props.environment}-localization-queue-processor`,
  runtime: lambda.Runtime.NODEJS_20_X,
  handler: 'process-queue.handler',
  code: lambda.Code.fromAsset('lambda/localization'),
```

```

        timeout: cdk.Duration.minutes(1),
        memorySize: 256,
        role: lambdaRole,
        environment: lambdaEnvironment,
    });

    // Grant queue send permissions
    this.translationQueue.grantSendMessages(queueProcessorLambda);

    // Schedule queue processor every 5 minutes
    new events.Rule(this, 'QueueProcessorSchedule', {
        ruleName: `radiant-${props.environment}-translation-queue-processor`,
        schedule: events.Schedule.rate(cdk.Duration.minutes(5)),
        targets: [new targets.LambdaFunction(queueProcessorLambda)],
    });

    // API Lambda
    this.apiLambda = new lambda.Function(this, 'LocalizationApiLambda', {
        functionName: `radiant-${props.environment}-localization-api`,
        runtime: lambda.Runtime.NODEJS_20_X,
        handler: 'api.handler',
        code: lambda.Code.fromAsset('lambda/localization'),
        timeout: cdk.Duration.seconds(30),
        memorySize: 512,
        role: lambdaRole,
        environment: lambdaEnvironment,
    });

    // Outputs
    new cdk.CfnOutput(this, 'TranslationQueueUrl', {
        value: this.translationQueue.queueUrl,
        exportName: `radiant-${props.environment}-translation-queue-url`,
    });

    new cdk.CfnOutput(this, 'LocalizationApiArn', {
        value: this.apiLambda.functionArn,
        exportName: `radiant-${props.environment}-localization-api-arn`,
    });
}
}

```

41.11 INITIAL TRANSLATION SEED DATA

Migration: 041b_seed_localization.sql

```
-- =====
-- RADIANT v4.7.0 - Seed Initial Translations
```

```

-- Migration: 041b_seed_localization.sql
-- =====

-- UI Buttons
INSERT INTO localization_registry (key, default_text, category, context, source_app) VALUES
('ui.buttons.submit', 'Submit', 'ui.buttons', 'Primary form submission button', 'shared'),
('ui.buttons.cancel', 'Cancel', 'ui.buttons', 'Cancel/close action', 'shared'),
('ui.buttons.save', 'Save', 'ui.buttons', 'Save changes button', 'shared'),
('ui.buttons.delete', 'Delete', 'ui.buttons', 'Delete/remove action', 'shared'),
('ui.buttons.edit', 'Edit', 'ui.buttons', 'Edit/modify action', 'shared'),
('ui.buttons.close', 'Close', 'ui.buttons', 'Close dialog/modal', 'shared'),
('ui.buttons.confirm', 'Confirm', 'ui.buttons', 'Confirmation action', 'shared'),
('ui.buttons.back', 'Back', 'ui.buttons', 'Navigate back', 'shared'),
('ui.buttons.next', 'Next', 'ui.buttons', 'Navigate forward/next step', 'shared'),
('ui.buttons.refresh', 'Refresh', 'ui.buttons', 'Reload/refresh data', 'shared'),
('ui.buttons.search', 'Search', 'ui.buttons', 'Search action', 'shared'),
('ui.buttons.send', 'Send', 'ui.buttons', 'Send message/request', 'shared'),
('ui.buttons.copy', 'Copy', 'ui.buttons', 'Copy to clipboard', 'shared'),
('ui.buttons.download', 'Download', 'ui.buttons', 'Download file', 'shared'),
('ui.buttons.upload', 'Upload', 'ui.buttons', 'Upload file', 'shared'),
('ui.buttons.retry', 'Retry', 'ui.buttons', 'Retry failed action', 'shared'),
('ui.buttons.approve', 'Approve', 'ui.buttons', 'Approve action (admin)', 'admin'),
('ui.buttons.reject', 'Reject', 'ui.buttons', 'Reject action (admin)', 'admin'),
('ui.buttons.save_and_approve', 'Save & Approve', 'ui.buttons', 'Save and approve translation')

-- Messages
INSERT INTO localization_registry (key, default_text, category, context, source_app, placeholder)
('messages.welcome', 'Welcome, {name}!', 'messages.success', 'Welcome message after login', 'shared'),
('messages.saved', 'Changes saved successfully', 'messages.success', 'After successful save', 'shared'),
('messages.deleted', 'Item deleted successfully', 'messages.success', 'After successful deletion', 'shared'),
('messages.copied', 'Copied to clipboard', 'messages.success', 'After copy action', 'shared'),
('messages.loading', 'Loading...', 'messages.loading', 'Generic loading state', 'shared', '[]'),
('messages.processing', 'Processing...', 'messages.loading', 'Processing action', 'shared', '[]'),
('messages.no_results', 'No results found', 'messages.info', 'Empty search results', 'shared', '[]'),
('messages.confirm_delete', 'Are you sure you want to delete this item?', 'messages.warning', '[]')

-- Errors
INSERT INTO localization_registry (key, default_text, category, context, source_app, placeholder)
('errors.generic', 'An error occurred. Please try again.', 'errors.system', 'Generic error fallback', 'shared'),
('errors.network', 'Network error. Please check your connection.', 'errors.network', 'Network connection error', 'shared'),
('errors.unauthorized', 'You are not authorized to perform this action.', 'errors.auth', 'Permission denied', 'shared'),
('errors.session_expired', 'Your session has expired. Please log in again.', 'errors.auth', 'Session timeout', 'shared'),
('errors.not_found', 'The requested item was not found.', 'errors.api', '404 error', 'shared', '[]'),
('errors.validation', 'Please check your input and try again.', 'errors.validation', 'Form validation error', 'shared'),
('errors.required_field', 'This field is required.', 'errors.validation', 'Required field validation', 'shared'),
('errors.invalid_email', 'Please enter a valid email address.', 'errors.validation', 'Email format validation', 'shared'),
('errors.rate_limit', 'Too many requests. Please wait a moment.', 'errors.api', 'Rate limiting exceeded', 'shared')

```

```

-- Think Tank specific
INSERT INTO localization_registry (key, default_text, category, context, source_app, placeholder)
('thinktank.chat.placeholder', 'Ask me anything...', 'features.thinktank', 'Chat input placeholder'),
('thinktank.chat.thinking', 'Thinking...', 'features.thinktank', 'AI is processing', 'thinktank'),
('thinktank.chat.error', 'Sorry, I encountered an error. Please try again.', 'features.thinktank'),
('thinktank.chat.regenerate', 'Regenerate response', 'features.thinktank', 'Button to regenerate'),
('thinktank.chat.stop', 'Stop generating', 'features.thinktank', 'Button to stop AI generation'),
('thinktank.chat.new', 'New conversation', 'features.thinktank', 'Start new chat', 'thinktank'),
('thinktank.models.select', 'Select model', 'features.thinktank', 'Model selector label', 'thinktank'),
('thinktank.models.auto', 'Auto (recommended)', 'features.thinktank', 'Automatic model selection');

-- Billing & Subscription Tiers (v4.13.0+)
INSERT INTO localization_registry (key, default_text, category, context, source_app) VALUES
('billing.tiers.free.name', 'Free Trial', 'billing.tiers', 'Free tier display name', 'shared'),
('billing.tiers.free.description', 'Try RADIANT with limited features', 'billing.tiers', 'Free tier description'),
('billing.tiers.individual.name', 'Individual', 'billing.tiers', 'Individual tier display name'),
('billing.tiers.individual.description', 'Full-featured personal plan', 'billing.tiers', 'Individual tier description'),
('billing.tiers.family.name', 'Family', 'billing.tiers', 'Family tier display name', 'shared'),
('billing.tiers.family.description', 'Share AI across your household', 'billing.tiers', 'Family tier description'),
('billing.tiers.team.name', 'Team', 'billing.tiers', 'Team tier display name', 'shared'),
('billing.tiers.team.description', 'Collaborate with your small team', 'billing.tiers', 'Team tier description'),
('billing.tiers.team.badge', 'Most Popular', 'billing.badges', 'Team tier badge text', 'shared'),
('billing.tiers.business.name', 'Business', 'billing.tiers', 'Business tier display name', 'shared'),
('billing.tiers.business.description', 'Enterprise features for growing companies', 'billing.tiers', 'Business tier description'),
('billing.tiers.enterprise.name', 'Enterprise', 'billing.tiers', 'Enterprise tier display name'),
('billing.tiers.enterprise.description', 'Full-scale enterprise deployment', 'billing.tiers', 'Enterprise tier description'),
('billing.tiers.enterprise_plus.name', 'Enterprise Plus', 'billing.tiers', 'Enterprise Plus tier display name'),
('billing.tiers.enterprise_plus.description', 'Maximum security and compliance', 'billing.tiers', 'Enterprise Plus tier description'),
('billing.tiers.enterprise_plus.badge', 'Full Compliance Included', 'billing.badges', 'Enterprise Plus tier badge text'),
('billing.credits.title', 'Credits', 'billing.credits', 'Credits section title', 'shared'),
('billing.credits.balance', 'Credit Balance', 'billing.credits', 'Current credit balance label'),
('billing.credits.purchase', 'Purchase Credits', 'billing.credits', 'Buy credits button', 'shared'),
('billing.credits.low_balance', 'Low credit balance', 'billing.credits', 'Low balance warning');

-- Admin specific
INSERT INTO localization_registry (key, default_text, category, context, source_app) VALUES
('admin.nav.dashboard', 'Dashboard', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.users', 'Users', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.tenants', 'Tenants', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.models', 'AI Models', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.billing', 'Billing', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.localization', 'Localization', 'features.admin', 'Navigation item', 'admin'),
('admin.nav.settings', 'Settings', 'features.admin', 'Navigation item', 'admin');

-- Localization admin specific
INSERT INTO localization_registry (key, default_text, category, context, source_app, placeholder)
('admin.localization.title', 'Localization Management', 'features.admin', 'Page title', 'admin'),
('admin.localization.tabs.coverage', 'Coverage', 'features.admin', 'Tab label', 'admin', '[]');

```

```

('admin.localization.tabs.review', 'Review Queue', 'features.admin', 'Tab label', 'admin', '[]'),
('admin.localization.tabs.registry', 'String Registry', 'features.admin', 'Tab label', 'admin'),
('admin.localization.language', 'Language', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.total', 'Total', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.approved', 'Approved', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.ai_translated', 'AI Translated', 'features.admin', 'Table header', 'admin'),
('admin.localization.pending', 'Pending', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.coverage', 'Coverage', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.key', 'Key', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.original', 'Original (English)', 'features.admin', 'Table header', 'admin'),
('admin.localization.translation', 'Translation', 'features.admin', 'Table header/label', 'admin'),
('admin.localization.status', 'Status', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.actions', 'Actions', 'features.admin', 'Table header', 'admin', '[]'),
('admin.localization.context', 'Context', 'features.admin', 'Context for translators', 'admin'),
('admin.localization.all_languages', 'All Languages', 'features.admin', 'Filter option', 'admin'),
('admin.localization.search_placeholder', 'Search keys or text...', 'features.admin', 'Search'),
('admin.localization.no_pending_reviews', 'No translations pending review. Great work!', 'feat'),
('admin.localization.edit_translation', 'Edit Translation', 'features.admin', 'Dialog title'),
('admin.localization.reject_reason', 'Please provide a reason for rejection:', 'features.admin'),
('admin.localization.bulk_approve', 'Approve All AI Translations ({count})', 'features.admin'),
('admin.localization.confirm_bulk_approve', 'Approve {count} AI-translated strings?', 'features'),
('admin.localization.status.pending', 'Pending', 'features.admin', 'Status label', 'admin', '[]'),
('admin.localization.status.ai_translated', 'AI Translated', 'features.admin', 'Status label'),
('admin.localization.status.in_review', 'In Review', 'features.admin', 'Status label', 'admin'),
('admin.localization.status.approved', 'Approved', 'features.admin', 'Status label', 'admin'),
('admin.localization.status.rejected', 'Rejected', 'features.admin', 'Status label', 'admin'),

-- Settings
INSERT INTO localization_registry (key, default_text, category, context, source_app) VALUES
('ui.settings.language', 'Language', 'features.settings', 'Language setting label', 'shared'),
('ui.settings.theme', 'Theme', 'features.settings', 'Theme setting label', 'shared'),
('ui.settings.notifications', 'Notifications', 'features.settings', 'Notifications setting labo',
('ui.settings.profile', 'Profile', 'features.settings', 'Profile section label', 'shared'),
('ui.settings.security', 'Security', 'features.settings', 'Security section label', 'shared'),
('ui.settings.account', 'Account', 'features.settings', 'Account section label', 'shared');

-- Note: English translations are auto-created by the trigger from default_text
-- Other languages will be AI-translated automatically via the queue

```
