

# KingdomQuest Internationalization (i18n) Playbook

**Version:** 1.0.0

**Last Updated:** 2025-08-26

**Author:** MiniMax Agent

A comprehensive guide to the internationalization system implemented in KingdomQuest, covering technical architecture, translation workflow, cultural considerations, and maintenance procedures.

## Table of Contents

1. [Implementation Overview](#)
  2. [Technical Architecture](#)
  3. [Translation Workflow](#)
  4. [Developer Guidelines](#)
  5. [Cultural Review Process](#)
  6. [Scripture Localization](#)
  7. [RTL Support](#)
  8. [Quality Assurance](#)
  9. [Maintenance Procedures](#)
  10. [Contributor Guidelines](#)
  11. [Troubleshooting](#)
-








# Implementation Overview

## System Architecture

KingdomQuest uses **next-intl** as the primary internationalization framework, providing:

- **Framework:** Next.js 15 with App Router + next-intl
- **Supported Languages:** English (en), Afrikaans (af), Spanish (es)
- **URL Structure:** `/locale/path` (e.g., `/en/stories`, `/af/verhale`, `/es/historias`)
- **Translation Management:** JSON-based translation catalogs
- **Type Safety:** Full TypeScript integration with translation key validation
- **RTL Scaffolding:** Ready for Arabic, Hebrew, Persian, and other RTL languages

## Key Features

-  **Locale Detection:** Automatic browser preference detection with URL persistence
  -  **SEO Optimization:** Localized metadata, hreflang tags, and OpenGraph support
  -  **Cultural Adaptation:** Localized biblical content with appropriate translations
  -  **Dynamic Routing:** Translated URLs for better UX (e.g., `/af/verhale` for stories)
  -  **Scripture Localization:** Bible verses in appropriate translations per locale
  -  **Accessibility:** WCAG 2.1 AA compliance with locale-aware screen reader support
  -  **Performance:** Lazy-loaded translations with build-time optimization
-

# Technical Architecture

## Core Configuration Files

```
kingdom-quest/
├─ i18n.ts                # next-intl configuration
├─ middleware.ts          # Locale detection and routing
├─ i18n/                  # Translation catalogs
│   ├─ en.json            # English translations
│   ├─ af.json            # Afrikaans translations
│   ├─ es.json            # Spanish translations
│   └─ index.ts           # Translation utilities
├─ lib/i18n/              # i18n utilities
│   ├─ scripture-resolver.ts # Bible translation system
│   ├─ rtl-utils.ts        # RTL support utilities
│   └─ DirectionProvider.tsx # Direction context provider
└─ app/[locale]/          # Localized app structure
    ├─ layout.tsx          # Locale-aware root layout
    └─ page.tsx            # Localized pages
```

## Translation Key Structure

Translation keys use hierarchical namespaces for organization:

```
{
  "nav": {
    "home": "Home",
    "stories": "Stories"
  },
  "auth": {
    "signIn": "Sign In",
    "errors": {
      "invalidEmail": "Please enter a valid email address"
    }
  }
}
```

## Locale Configuration

### Supported Locales:

- `en` - English (default)
- `af` - Afrikaans
- `es` - Spanish (Español)

### RTL Support Ready For:

- `ar` - Arabic
  - `he` - Hebrew
  - `fa` - Persian/Farsi
  - `ur` - Urdu
- 

## Translation Workflow

### 1. Adding New Translation Keys

**Step 1:** Add the key to the base English file (`i18n/en.json`)

```
{
  "stories": {
    "newFeature": "New Feature Title"
  }
}
```

### **Step 2:** Use the automated update script

```
npm run i18n:update
```

This will:

- Add missing keys to all locale files
- Mark them for translation
- Update metadata

### **Step 3:** Provide translations for each locale

```
// af.json
{
  "stories": {
    "newFeature": "Nuwe Funksie Titel"
  }
}

// es.json
{
  "stories": {
    "newFeature": "Título de Nueva Función"
  }
}
```

## **2. Translation Quality Assurance**

### **Automated Validation:**

```
# Check translation completeness and consistency
npm run i18n:check

# Extract new translation keys from codebase
npm run i18n:extract
```

#### Manual Review Checklist:

- [ ] Cultural appropriateness
- [ ] Religious sensitivity
- [ ] Gender-neutral language where appropriate
- [ ] Consistent terminology
- [ ] Proper capitalization and punctuation
- [ ] UI text length considerations

### 3. Adding New Languages

**Step 1:** Update locale configuration in `middleware.ts`

```
export const locales = ['en', 'af', 'es', 'pt'] as const;

export const localeNames = {
  en: 'English',
  af: 'Afrikaans',
  es: 'Español',
  pt: 'Português'
} as const;
```

**Step 2:** Create translation file

```
# Copy English template
cp i18n/en.json i18n/pt.json
```

**Step 3:** Update metadata and translate content

#### Step 4: Add to Language Switcher

```
const languageOptions: LanguageOption[] = [  
  // ... existing languages  
  {  
    code: 'pt',  
    name: 'Portuguese',  
    nativeName: 'Português',  
    direction: 'ltr',  
    flag: '🇵🇹'  
  }  
];
```

#### Step 5: Configure Bible translation in Scripture Resolver

---

## Developer Guidelines

### Using Translations in Components

#### Client Components:

```

'use client';

import { useTranslations, useLocale } from 'next-intl';

export function MyComponent() {
  const t = useTranslations('stories');
  const locale = useLocale();

  return (
    <div>
      <h1>{t('title')}</h1>
      <p>{t('description', { name: 'John' })}</p>
    </div>
  );
}

```

## Server Components:

```

import { getTranslations } from 'next-intl/server';

export default async function MyPage() {
  const t = await getTranslations('stories');

  return (
    <div>
      <h1>{t('title')}</h1>
      <p>{t('description')}</p>
    </div>
  );
}

```

## Parameterized Translations:



```
{
  "welcome": "Welcome back, {name}!",
  "itemCount": "You have {count, plural, =0 {no items} =1 {one item} other {# items}}"
}
```

```
const t = useTranslations('home');

// Simple parameter
<p>{t('welcome', { name: user.name })}</p>

// Pluralization
<p>{t('itemCount', { count: items.length })}</p>
```

## Locale-Aware Navigation

```
import Link from 'next/link';
import { useLocale } from 'next-intl';

function Navigation() {
  const locale = useLocale();

  return (
    <nav>
      <Link href={`/${locale}/stories`} >Stories</Link>
      <Link href={`/${locale}/quiz`} >Quiz</Link>
    </nav>
  );
}
```

## Best Practices

1. **Always use translation keys** - Never hardcode user-facing strings
  2. **Use descriptive namespaces** - Group related translations logically
  3. **Keep keys descriptive** - `auth.signIn` not `auth.btn1`
  4. **Handle missing translations gracefully** - Provide fallbacks
  5. **Consider text expansion** - Languages can be 30-50% longer than English
  6. **Test with long translations** - Ensure UI handles text overflow
- 

## Cultural Review Process

### Cultural Sensitivity Guidelines

#### Religious Content:

- Respect different Christian traditions and denominational preferences
- Use inclusive language that doesn't favor specific theological positions
- Ensure biblical content is culturally appropriate for target regions
- Consult local Christian leaders for cultural validation

#### Visual Considerations:

- **Colors:** Be aware of cultural color symbolism
  - Red: Good luck in China, danger in Western cultures
  - White: Purity in West, mourning in some Asian cultures
  - Green: Islamic associations, nature in Western cultures
- **Imagery:** Ensure biblical illustrations are culturally sensitive
- **Icons:** Use universally understood symbols when possible

#### Language Considerations:

- **Formality Levels:** Some languages have formal/informal distinctions
- **Gender:** Consider grammatical gender rules and neutral options
- **Regional Variants:** Spanish has many regional differences

## Cultural Review Checklist

### Pre-Release Review:

- ☐ **Local Review:** Native speaker review by target culture member
- ☐ **Religious Review:** Theological appropriateness verification
- ☐ **Visual Review:** Cultural appropriateness of images and colors
- ☐ **Functional Review:** UI/UX testing in target locale
- ☐ **Technical Review:** Font rendering, text direction, input methods

### Ongoing Monitoring:

- ☐ User feedback collection in each locale
  - ☐ Regular content updates based on cultural input
  - ☐ Community engagement with local Christian communities
- 

## Scripture Localization

### Bible Translation System

The Scripture Locale Resolver automatically selects appropriate Bible translations based on user locale:

```
// Bible translations by locale
export const bibleTranslations = {
  en: {
    primary: 'WEB',    // World English Bible (Public Domain)
    secondary: 'KJV',  // King James Version
    name: 'World English Bible'
  },
  af: {
    primary: 'AFR1933', // Afrikaans 1933
    secondary: 'AFR1953', // Afrikaans 1953
    name: 'Afrikaans Bybel 1933'
  },
  es: {
    primary: 'RVA',      // Reina-Valera Antigua
    secondary: 'RVR1960', // Reina-Valera Revisada 1960
    name: 'Reina-Valera Antigua'
  }
};
```

## Usage in Components

```
import { ScriptureLocaleResolver } from '@lib/i18n/scripture-resolver';

const scriptureResolver = new ScriptureLocaleResolver();

// Get localized scripture
const scripture = await scriptureResolver.getScripture(
  'John 3:16',
  locale
);

console.log(scripture.text);           // Localized verse text
console.log(scripture.translation);    // Translation name (e.g.,
"AFR1933")
console.log(scripture.copyright);      // Copyright information
```

## Adding New Bible Translations

1. **Identify appropriate translation** for the target locale
  2. **Verify copyright status** - prefer public domain translations
  3. **Add to translation mapping** in `scripture-resolver.ts`
  4. **Test with various scripture references**
  5. **Update documentation**
-

# RTL Support

## Current RTL Infrastructure

The system includes comprehensive RTL scaffolding ready for Arabic, Hebrew, Persian, and other RTL languages:

**RTL Utilities** ( `lib/i18n/rtl-utils.ts` ):

```
// Direction detection
export function getTextDirection(locale: Locale): 'ltr' | 'rtl' {
  return isRTLLocale(locale) ? 'rtl' : 'ltr';
}

// RTL-aware CSS classes
export const rtlUtils = {
  ml: (value: string) => `ml-<span class="math-inline"
style="display: inline;"><math xmlns="http://www.w3.org/1998/Math/
MathML" display="inline"><mrow><mrow><mi>v</mi><mi>a</mi><mi>l</
mi><mi>u</mi><mi>e</mi></mrow><mi>r</mi><mi>t</mi><mi>l</mi><mi>:</
mi><mi>m</mi><mi>l</mi><mo>&#x02212;</mo><mn>0</mn><mi>r</
mi><mi>t</mi><mi>l</mi><mi>:</mi><mi>m</mi><mi>r</
mi><mo>&#x02212;</mo></mrow></math></span>{value}` ,
  mr: (value: string) => `mr-<span class="math-inline"
style="display: inline;"><math xmlns="http://www.w3.org/1998/Math/
MathML" display="inline"><mrow><mrow><mi>v</mi><mi>a</mi><mi>l</
mi><mi>u</mi><mi>e</mi></mrow><mi>r</mi><mi>t</mi><mi>l</mi><mi>:</
mi><mi>m</mi><mi>r</mi><mo>&#x02212;</mo><mn>0</mn><mi>r</
mi><mi>t</mi><mi>l</mi><mi>:</mi><mi>m</mi><mi>l</
mi><mo>&#x02212;</mo></mrow></math></span>{value}` ,
  'text-left': 'text-left rtl:text-right',
  'text-right': 'text-right rtl:text-left'
};
```

**Direction Provider:**

```
import { DirectionProvider, useDirection } from '@lib/i18n/
DirectionProvider';

// In layout
<DirectionProvider locale={locale}>
  {children}
</DirectionProvider>

// In components
const { direction, isRTL } = useDirection();
```

## RTL-Ready Component Examples

```
// RTL-aware navigation
<nav className={`flex ${isRTL ? 'flex-row-reverse' : 'flex-row'} `}>
  <Button className={rtlUtils.ml('4')}>Previous</Button>
  <Button className={rtlUtils.mr('4')}>Next</Button>
</nav>

// RTL-aware text alignment
<p className={rtlUtils['text-left']}>Aligned text</p>

// RTL-aware icons
<ArrowRight className={` ${isRTL ? 'rotate-180' : ''} `} />
```

## Adding RTL Language Support

**Step 1:** Add locale to RTL configuration

```
// In rtl-utils.ts
export const RTL_LOCALES: readonly Locale[] = ['ar', 'he', 'fa']
as const;
```

## Step 2: Update Tailwind CSS configuration

```
// tailwind.config.ts
module.exports = {
  // Enable RTL support
  plugins: [
    require('tailwindcss/plugin')(({ addUtilities }) => {
      addUtilities({
        '.rtl': { direction: 'rtl' },
        '.ltr': { direction: 'ltr' }
      })
    })
  ]
}
```

## Step 3: Test all UI components in RTL mode

## Step 4: Validate with native RTL speakers

---

# Quality Assurance

## Automated Testing

### Translation Validation:

```
# Check all translations for completeness
npm run i18n:check

# Validate translation keys exist in codebase
npm run i18n:extract

# Update missing translations
npm run i18n:update
```



## E2E Testing with Locales:

```
// tests/e2e/i18n.spec.ts
import { test, expect } from '@playwright/test';

test.describe('Internationalization', () => {
  ['en', 'af', 'es'].forEach(locale => {
    test(`Navigation works in ${locale}`, async ({ page }) => {
      await page.goto(`/${locale}`);
      await expect(page.locator('nav')).toBeVisible();

      // Test locale-specific navigation
      if (locale === 'af') {
        await expect(page.locator('text=Verhale')).toBeVisible();
      } else if (locale === 'es') {
        await expect(page.locator('text=Historias')).toBeVisible();
      }
    });
  });
});
```

## Manual Testing Checklist

### Per Locale Testing:

- [ ] **Navigation:** All menu items display correctly
- [ ] **Authentication:** Sign in/up flows work properly
- [ ] **Content:** Stories, quizzes load with correct language
- [ ] **Forms:** Validation messages appear in correct language
- [ ] **Error Handling:** Error messages are localized
- [ ] **URLs:** Localized routes work correctly
- [ ] **SEO:** Meta tags and OpenGraph data are localized

### Cross-Locale Testing:

- [ ] **Language Switching:** Smooth transitions between locales
- [ ] **URL Persistence:** Language choice persists across navigation

- [ ] **Browser Detection:** Automatic locale detection works
  - [ ] **Fallback Handling:** Missing translations fall back gracefully
- 

## Maintenance Procedures

### Regular Maintenance Tasks

#### Monthly:

- [ ] Run `npm run i18n:check` to validate translation completeness
- [ ] Review translation quality reports
- [ ] Update any outdated translations
- [ ] Check for new untranslated strings

#### Quarterly:

- [ ] Review cultural feedback from users
- [ ] Update Bible translations if new versions become available
- [ ] Assess need for new locale support based on user analytics
- [ ] Performance review of i18n system

#### Annually:

- [ ] Comprehensive cultural review with native speakers
- [ ] Theology review with local Christian leaders
- [ ] Translation memory cleanup and optimization
- [ ] Documentation updates

### Performance Optimization

#### Translation Bundle Optimization:

```
# Analyze bundle sizes
npm run analyze

# Check for unused translation keys
npm run i18n:extract -- --check-unused
```

## Lazy Loading Optimization:

Translations are automatically lazy-loaded per route to minimize bundle size:

```
// Only load translations for current namespace
const t = useTranslations('stories'); // Only loads 'stories'
namespace
```

---

# Contributor Guidelines

## For Translators

### Getting Started:

1. **Fork the repository** and create a feature branch
2. **Choose your locale** - work on existing or propose new language
3. **Use translation tools** - Leverage automated checking scripts
4. **Follow style guide** - Maintain consistency with existing translations
5. **Test your changes** - Use local development environment

### Translation Standards:

- **Consistency:** Use consistent terminology throughout
- **Context:** Consider UI context when translating
- **Cultural Sensitivity:** Ensure appropriateness for target culture
- **Religious Accuracy:** Maintain theological accuracy
- **Technical Accuracy:** Preserve technical meaning

### Submission Process:

1. **Validate translations:** Run `npm run i18n:check`
2. **Test locally:** Verify UI appearance and functionality
3. **Create Pull Request:** Include description of changes

4. **Address feedback:** Respond to review comments
5. **Celebrate:** Your contribution helps global users!

## For Developers

### Adding Translation Support to New Features:

1. **Extract all user-facing strings** - Never hardcode text
2. **Use appropriate namespaces** - Group related translations
3. **Add to base locale first** - Start with `en.json`
4. **Run update script** - `npm run i18n:update` to propagate
5. **Test with multiple locales** - Ensure UI works correctly
6. **Consider text expansion** - Plan for longer translations

### Code Review Checklist:

- ☐ No hardcoded strings in user-facing components
  - ☐ Translation keys follow naming conventions
  - ☐ Proper namespace organization
  - ☐ Parameterized translations where needed
  - ☐ Locale-aware navigation implemented
  - ☐ RTL considerations addressed if applicable
- 

## Troubleshooting

### Common Issues

#### Problem: Translations not loading

Symptoms: Keys showing instead of translated text

Solutions:

1. Check if translation key exists in JSON file

2. Verify namespace is correct in `useTranslations( 'namespace' )`
3. Ensure locale is properly configured in middleware
4. Clear Next.js cache: `rm -rf .next`

### **Problem: Hydration mismatches with translations**

Symptoms: Server/client render differences

Solutions:

1. Ensure translations are loaded consistently on server/client
2. Use `<Suspense>` boundary around translated components
3. Check for dynamic content that varies by locale

### **Problem: RTL layout issues**

Symptoms: UI elements incorrectly positioned in RTL mode

Solutions:

1. Use logical CSS properties instead of directional ones
2. Apply RTL utility classes from `rtl-utils.ts`
3. Test icon directions and margins
4. Verify flex direction handling

### **Problem: Scripture not loading in target locale**

Symptoms: Always shows English Bible verses

Solutions:

1. Check `scripture-resolver.ts` has translation mapping for locale
2. Verify Bible translation data exists in database
3. Check fallback logic is working properly
4. Validate locale parameter is being passed correctly

## Debug Commands

```
# Validate all translations
npm run i18n:check

# Extract translation keys from codebase
npm run i18n:extract

# Update translation files
npm run i18n:update

# Check Next.js build
npm run build

# Analyze bundle sizes
npm run analyze
```

## Getting Help

### Internal Resources:

- Review this playbook documentation
- Check automated validation reports
- Examine existing implementations in codebase
- Use TypeScript types for guidance

### External Resources:

- [next-intl documentation](#)
  - [Unicode CLDR Locale Data](#)
  - [W3C Internationalization Guidelines](#)
  - [Mozilla i18n Guide](#)
-

# Conclusion

The KingdomQuest i18n system provides a robust foundation for global accessibility while maintaining cultural sensitivity and theological accuracy. The system is designed for scalability, allowing easy addition of new languages while maintaining high quality standards.

This playbook serves as both a reference guide and operational manual. Keep it updated as the system evolves, and always prioritize user experience and cultural appropriateness in all translation decisions.

## **Key Success Metrics:**

- Translation completeness: 100% for all supported locales
  - Cultural appropriateness: Validated by native speakers
  - Performance impact: <5% increase in bundle size per additional locale
  - User satisfaction: Positive feedback from global user community
- 

Last updated: 2025-08-26 by MiniMax Agent

For questions or contributions, please follow the contributor guidelines above.