Building a Multilingual Template System in Flutter: From Notion Clone to Self-Hosted Workspace

Have you ever wondered how to create a robust, self-hosted alternative to Notion that supports multiple languages? Let me take you through the journey of building **Bloquinho** - a complete Flutter-based workspace application that rivals modern productivity tools while giving users full control over their data.

🎯 What is Bloquinho?

Bloquinho is a self-hosted workspace application built with Flutter that provides:

- 📝 **Rich Markdown Editor** with LaTeX, Mermaid diagrams, and code highlighting
- 🗄 **Database Management** with custom tables and relationships
- 📊 **Project Management** with agenda and task tracking
- 🔐 **Password Manager** with secure local storage
- 📁 **Document Management** with hierarchical organization
- 🌐 **Multi-language Support** (Portuguese, English, French)
- ☁ **Cloud Integration** with OAuth2 providers (Google Drive, OneDrive, Dropbox)

🚀 The Challenge: Building a Multilingual Template System

One of the most interesting challenges we faced was creating a sophisticated template system that could automatically generate rich, formatted content in multiple languages when users create new pages. This isn't just about translating text - it's about providing contextually appropriate templates that adapt to the user's language preference.

The Requirements

- **Rich Templates**: Full-featured Markdown with LaTeX formulas, tables, diagrams
- **Language Detection**: Automatic template selection based on user's language
- **Smart Detection**: Different templates for root pages vs. regular pages
- **Maintainability**: Easy to add new languages and modify templates

🏗 Architecture Overview

Bloquinho follows a clean architecture pattern with clear separation of concerns:

```
lib/
├── core/                    # Core services and models
│   ├── services/          # Storage, auth, cloud services
│   ├── models/           # Data models
│   └── l10n/             # Internationalization
├── features/             # Feature modules
│   ├── bloquinho/        # Main editor and pages
│   ├── database/         # Database management
│   ├── workspace/        # Workspace management
│   └── auth/               # Authentication
└── shared/                 # Shared components and providers
```

The application uses **Riverpod** for state management, **Go Router** for navigation, and **Hive** for local storage, making it truly self-hosted while maintaining excellent performance.

💡 Implementing the Multilingual Template System

Step 1: Creating Language-Specific Templates

First, we created separate template files for each language:

~~~
// page_templates_pt.dart
class PageTemplatesPt {
  static const String rootPageTemplate = '''# 📊 Documento Markdown Completo

## 🎨 Demonstração de Cores e Formatação

Este documento exemplifica **todas as principais funcionalidades** do Markdown...

### Exemplos de Código

```python
def calcular_fibonacci(n):
    """Calcula a sequência de Fibonacci até n termos"""
    # Implementation here
~~~

🔢 Fórmulas Matemáticas (LaTeX)

A famosa equação de Einstein: $E = mc^2$

Matrizes

$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

📊 Tabelas Avançadas

Mês	Vendas	Meta	Status
Janeiro	€15.000	€12.000	✅ Superou

''';

static const String newPageTemplate = '''# Nova Página

Introdução

Bem-vindo à sua nova página!

Lista de Tarefas

- [ ] Primeira tarefa
- [ ] Segunda tarefa '"; }

~~~
### Step 2: Smart Template Selection Service

The core of our system is the `PageTemplateService` that intelligently selects
templates:

```dart
class PageTemplateService {
  /// Get template based on language and page type
  static String getTemplate(AppLanguage language, PageTemplateType type) {
    switch (language) {
      case AppLanguage.portuguese:
        return _getPortugueseTemplate(type);
      case AppLanguage.english:
        return _getEnglishTemplate(type);
      case AppLanguage.french:
        return _getFrenchTemplate(type);
    }
  }

  /// Detect if page should use root template based on title
  static bool isRootPageCandidate(String title) {
    final normalizedTitle = title.toLowerCase().trim();

    const rootPageTitles = [
      'main', 'principal', 'início', 'home', 'accueil',
      'root', 'raiz', 'index', 'dashboard', 'workspace'
    ];

    return rootPageTitles.contains(normalizedTitle);
  }

  /// Get appropriate template based on title and language
  static String getAppropriateTemplate(String title, AppLanguage language) {
    if (isRootPageCandidate(title)) {
      return getRootPageTemplate(language);
    } else {
      return getNewPageTemplate(language);
    }
  }
}
~~~

Step 3: Integration with State Management

We enhanced our `PagesNotifier` to automatically apply templates:

```
class PagesNotifier extends StateNotifier<List<PageModel>> {
  /// Create new page with template based on language
  Future<void> createPageWithTemplate({
    required String title,
    String? icon,
    String? parentId,
    required AppLanguage language,
  }) async {
    await createPage(
      title: title,
      icon: icon,
      parentId: parentId,
      content: '', // Will be filled by template
      language: language,
    );
  }

  Future<void> createPage({
    required String title,
    String? icon,
    String? parentId,
    String content = '',
    AppLanguage? language,
  }) async {
    // ... validation code ...

    // Use appropriate template if content is empty
    String finalContent = content;
    if (content.isEmpty && language != null) {
      finalContent = PageTemplateService.getAppropriateTemplate(title, language);
    }

    final page = PageModel.create(
      title: title,
      icon: icon,
      parentId: parentId,
      content: finalContent,
    );

    // ... save logic ...
  }
}
```

Step 4: UI Integration

Finally, we updated the UI to use the new template system:

```
// In BloquinhoDashboardScreen
onParentSelected: (parentId) async {
  final pagesNotifier = ref.read(pagesNotifierProvider((
    profileName: currentProfile.name,
    workspaceName: currentWorkspace.name
  )));

  final currentLanguage = ref.read(appStringsProvider).currentLanguage;

  await pagesNotifier.createPageWithTemplate(
    title: strings.newSubpage,
    parentId: parentId,
    language: currentLanguage,
  );

  // Navigate to the new page...
}
```

🔥 Key Features That Make Bloquinho Special

1. **Rich Markdown Support**

- LaTeX mathematical formulas
- Mermaid diagrams for flowcharts and sequences
- Syntax highlighting for 20+ programming languages
- Custom HTML styling with inline CSS

2. **True Self-Hosting**

- No external dependencies for core functionality
- Local SQLite database with Hive for metadata
- File-based storage with hierarchical organization
- Optional cloud backup integration

3. **Advanced Database Features**

- Visual table editor with drag-and-drop columns
- Custom field types (text, number, date, boolean, file)
- Relationship management between tables
- Export to CSV/JSON formats

4. **Workspace Management**

- Multiple workspaces per user profile
- Workspace-specific settings and themes
- Import/export functionality
- Backup and restore capabilities

📊 Performance Considerations

Building a self-hosted application requires careful attention to performance:

- **Lazy Loading**: Pages and content are loaded on-demand
- **Caching Strategy**: Intelligent caching of frequently accessed content

- **File Organization**: Hierarchical file structure for efficient storage
- **State Management**: Optimized Riverpod providers with family modifiers

🌍 Internationalization Architecture

Our i18n system goes beyond simple text translation:

```
class AppStrings {
  final AppLanguage _language;

  String get createNewPage {
    switch (_language) {
      case AppLanguage.portuguese:
        return 'Criar nova página';
      case AppLanguage.english:
        return 'Create new page';
      case AppLanguage.french:
        return 'Créer une nouvelle page';
    }
  }
}
```

This approach ensures:

- **Type Safety**: Compile-time checking for missing translations
- **Context Awareness**: Different translations for different contexts
- **Easy Maintenance**: Centralized translation management

🚀 What's Next?

Bloquinho is continuously evolving with planned features:

- **Real-time Collaboration**: Multi-user editing with conflict resolution
- **Plugin System**: Extensible architecture for custom functionality
- **Mobile Optimization**: Enhanced mobile UI/UX
- **Advanced Search**: Full-text search across all content
- **API Layer**: RESTful API for third-party integrations

🎯 Key Takeaways

Building Bloquinho taught us several important lessons:

1. **Architecture Matters**: Clean separation of concerns makes features like multilingual templates much easier to implement
2. **User Experience**: Intelligent defaults (like automatic template selection) greatly improve UX
3. **Internationalization**: Plan for i18n from day one - retrofitting is much harder
4. **Self-Hosting**: Users value data ownership and control over their tools
5. **Performance**: Local-first architecture can be both fast and reliable

🔗 Try It Yourself

Bloquinho represents the future of self-hosted productivity tools. By combining the power of Flutter with thoughtful architecture, we've created something that's both powerful and user-friendly.

The multilingual template system is just one example of how attention to detail can significantly improve user experience. When users create a new page, they immediately get rich, contextually appropriate content in their preferred language - no configuration required.

What features would you like to see in a self-hosted workspace application? Share your thoughts in the comments below!

Tags: #flutter #dart #notion #productivity #opensource #selfhosted #internationalization #markdown #latex