AuthorForge Tempering

Complete Implementation Guide

Export & Publishing Engine

Version 1.0

November 13, 2025

Table of Contents

Overview & Architecture

- System Design
- Technology Stack
- Component Architecture

Stage 1: Complete SolidJS Components

- Core Type Definitions
- Main Tempering Page
- Quick Export View

Stage 2: Component Details & Subsystems

- Custom Hooks
- Source Panel
- Profile Editor Panel
- Validation Panel
- Asset Binding Panel
- Live Preview Panel
- Export History Panel

Stage 3: Python Export Engine

- Base Engine Architecture
- EPUB Engine
- PDF Engine (WeasyPrint)
- DOCX Engine
- Markdown Engine
- Content Processor

Stage 4: Database Schema & Migrations

PostgreSQL Schema

SQLAlchemy Models

Repository Layer

Migration Management

Overview & Architecture

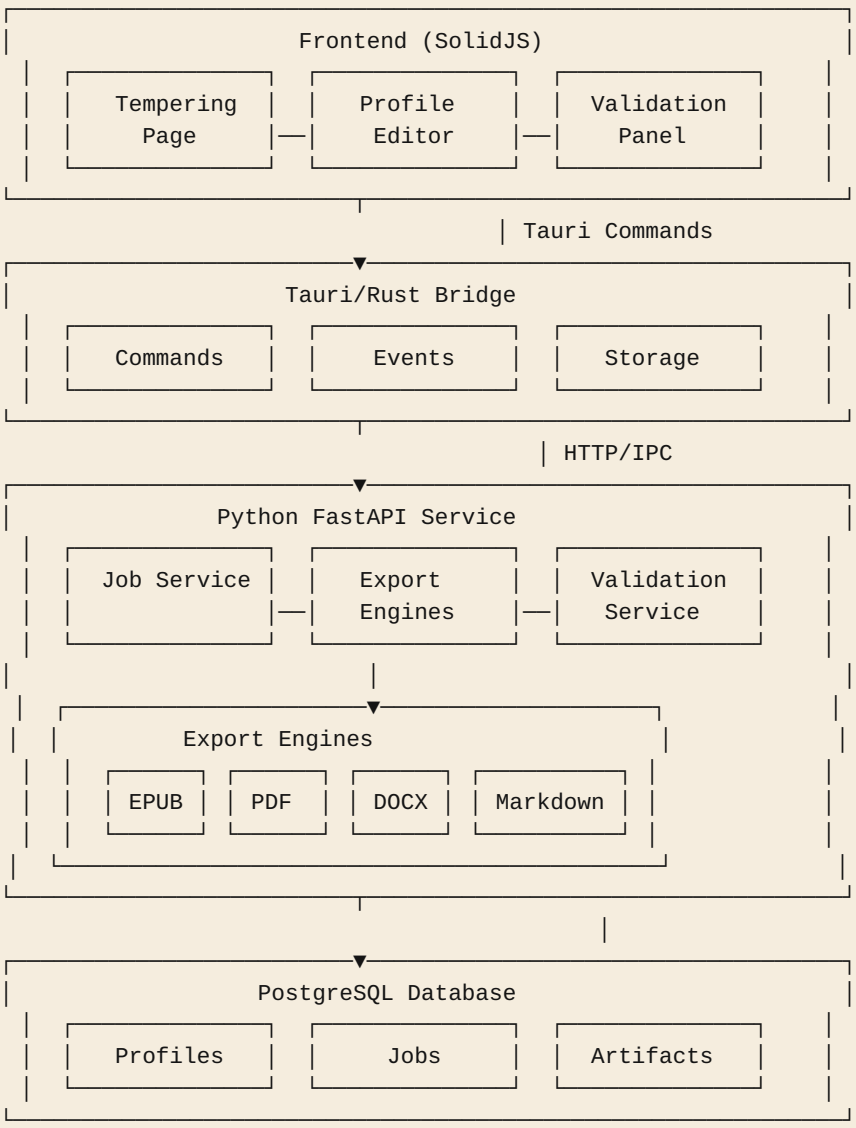
System Design

The **Tempering** module is AuthorForge's export and publishing engine, responsible for transforming manuscript content into professional, publish-ready formats. The name "Tempering" follows the forge metaphor—just as tempering is the final step in metalworking that hardens and finishes steel, this module takes raw manuscript content and forges it into polished, distributable documents.

Core Capabilities

- **Multi-Format Export:** EPUB, PDF, DOCX, Markdown, and HTML
- **Profile-Based Configuration:** Save and reuse export settings
- **Live Preview:** See formatting changes before export
- **Validation Engine:** Pre-flight checks for common issues
- **Asset Management:** Cover images, interior illustrations, maps
- **Template System:** Pre-configured profiles for common use cases

Technology Stack



Export Process Flow

1. **Configuration:** User selects/creates an export profile with formatting options
2. **Validation:** System performs pre-flight checks on content and settings
3. **Content Gathering:** Manuscript content is collected based on scope
4. **Processing:** Content is transformed according to profile settings

5. **Generation:** Format-specific engines create output files
6. **Delivery:** Artifacts are made available for download

> **Design Philosophy:** The Tempering module follows AuthorForge's core principle of providing professional-grade tools without overwhelming complexity. Pre-configured templates handle common scenarios, while power users can fine-tune every aspect of the export process.

Stage 1: Complete SolidJS Components

Core Type Definitions

The type system provides strong typing for all export-related data structures, ensuring type safety across the frontend and enabling excellent IDE support.

src/lib/types/tempering.ts

```
export type ExportKind =
  | "manuscript"
  | "novel"
  | "scientific_paper"
  | "blog_post"
  | "newsletter";

export type ExportFormat = "epub" | "docx" | "pdf" | "markdown" | "html";

export type ExportPhase =
  | "cold"          // not started
  | "heating"       // gathering content
  | "hammering"     // applying formatting
  | "quenching"     // finalizing
  | "cooled"        // complete
  | "cracked";      // failed

export const PHASE_DISPLAY = {
  cold: { icon: "❄️", label: "Ready", color: "text-blue-400" },
  heating: { icon: "🔥", label: "Gathering", color: "text-orange-400" },
  hammering: { icon: "🔨", label: "Forging", color: "text-amber-400" },
  quenching: { icon: "💧", label: "Tempering", color: "text-cyan-400" },
  cooled: { icon: "✓", label: "Complete", color: "text-green-400" },
  cracked: { icon: "⚠️", label: "Failed", color: "text-red-400" },
} as const;
```

Profile Structure

Export profiles encapsulate all settings needed to generate a specific format. They're organized into logical groups:

Category	Purpose	Example Settings
Formatting	Text appearance	Font family, size, line spacing, alignment
Layout	Page dimensions	Page size, margins, headers/footers
Structure	Document sections	Title page, TOC, dedication, chapter breaks
Kind Options	Format-specific	Manuscript headers, novel drop caps, blog SEO

Main Tempering Page

The main page provides two modes: **Quick** for rapid exports with saved profiles, and **Detailed** for full control over all settings.

src/routes/tempering/index.tsx

```
export default function TemperingPage() {
  const [viewMode, setViewMode] = createSignal<"quick" | "detailed">("quick");
  const [selectedProfileId, setSelectedProfileId] = createSignal<string | null>(null);

  const {
    profiles,
    selectedProfile,
    createProfile,
    updateProfile,
  } = useExportProfiles(params.projectId);

  const { currentJob, startExport } = useExportJob();
  const { validation, revalidate } = useValidation(() => selectedProfile());

  // ... component implementation
}
```

Key Features

- **Mode Toggle:** Switch between quick and detailed editing
- **Profile Selection:** Dropdown for saved profiles
- **Real-time Validation:** Pre-flight checks before export
- **Progress Monitoring:** Live updates during export
- **Artifact Download:** One-click access to generated files

Stage 2: Component Details & Subsystems

Custom Hooks

Three primary hooks manage state and orchestrate backend communication:

useExportProfiles

src/routes/tempering/hooks/useExportProfiles.ts

```
export function useExportProfiles(projectId: string) {
  const [selectedId, setSelectedId] = createSignal<string | null>(null);

  const [profiles] = createResource(
    () => projectId,
    async (id) => {
      // Load profiles from Tauri backend
      return await invoke<ExportProfile[]>("get_export_profiles", { projectId: id });
    }
  );

  return {
    profiles,
    selectedProfile,
    createProfile,
    updateProfile,
    deleteProfile,
  };
}
```

useExportJob

Manages export job lifecycle with automatic polling for progress updates.

useValidation

Performs pre-flight checks on profile configuration and content, warning about:

- Missing required images (e.g., cover for EPUB)
- Invalid formatting combinations
- Empty or malformed content
- Unsupported format/kind combinations

Panel Components

ProfileEditorPanel

The heart of the configuration interface, providing tabbed access to all profile settings:

- **Basic Tab:** Metadata (author, ISBN, language)
- **Formatting Tab:** Typography and text styling
- **Layout Tab:** Page size and margins
- **Structure Tab:** Document sections and breaks

> **Design Pattern:** All panels follow a consistent structure with:
>
> - Header with icon and title
> - Content area with form controls
> - Action buttons at bottom (if applicable)
> - Responsive padding and spacing using Tailwind utilities

ValidationPanel

Displays real-time statistics and warnings:

- Word count and estimated page count
- Chapter and scene counts
- Errors (blocking issues)
- Warnings (non-blocking concerns)
- Info messages (helpful suggestions)

AssetBindingPanel

Manages image uploads and role assignments:

- Drag-and-drop upload area
- Image preview with metadata
- Role assignment (cover, figure, hero image, etc.)
- Chapter/page binding for figures

LivePreviewPanel

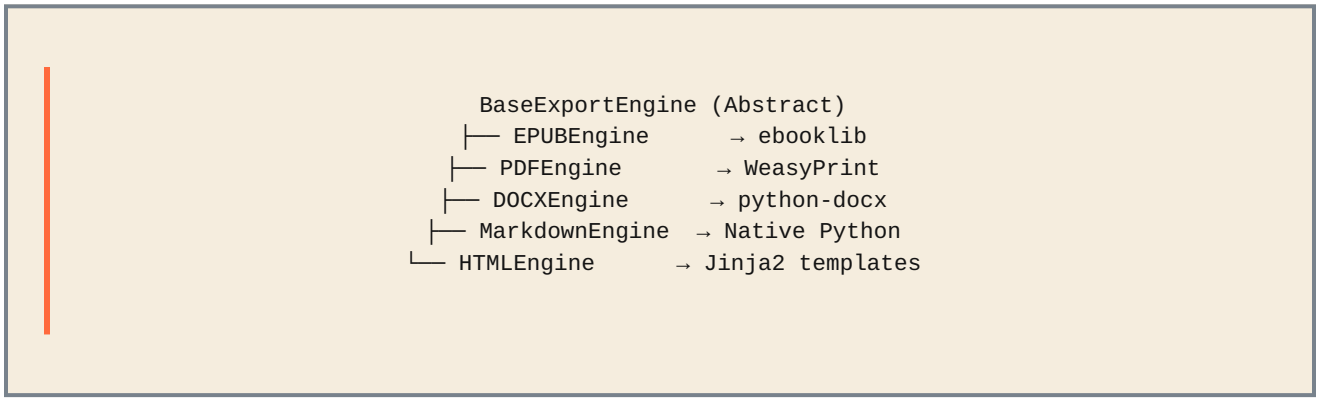
Provides three preview modes with zoom controls:

- **Cover:** Title page with metadata
- **First Page:** Sample of formatted content
- **Spread:** Two-page view for print layout

Stage 3: Python Export Engine

Architecture Overview

The export engine is built as a FastAPI service that Tauri communicates with via HTTP. Each format (EPUB, PDF, DOCX, Markdown) has a dedicated engine that inherits from a common base class.



Base Engine

backend-python/tempering/engines/base.py

```
class BaseExportEngine(ABC):
    """Abstract base class for all export engines"""

    @property
    @abstractmethod
    def format(self) -> ExportFormat:
        """The format this engine produces"""
        pass

    @abstractmethod
    async def generate(self, profile: ExportProfile, content: Dict[str, Any]) -> str:
        """
        Generate the export file

        Args:
            profile: Export profile with all settings
            content: Processed content from ContentProcessor

        Returns:
            Path to the generated file
        """
        pass
```

EPUB Engine

The EPUB engine uses `ebooklib` to create standards-compliant EPUB3 files with proper metadata, table of contents, and CSS styling.

Key Features

- Full metadata support (ISBN, publisher, series, etc.)
- Embedded cover image
- Front matter (title, copyright, dedication)
- Automatic TOC generation
- CSS styling based on profile formatting
- Drop cap support for chapter openings
- Scene break formatting

backend-python/tempering/engines/epub_engine.py

```
async def generate(self, profile: ExportProfile, content: Dict[str, Any]) -> str:
    """Generate EPUB file"""
    book = epub.EpubBook()

    # Set metadata
    self._set_metadata(book, profile)

    # Add cover
    if cover_binding := next((b for b in profile.image_bindings if b.role == "cover"), None):
        await self._add_cover(book, cover_binding)

    # Add chapters
    chapters = []
    for idx, chapter_data in enumerate(content["chapters"]):
        chapter = self._create_chapter(book, profile, chapter_data, idx + 1)
        chapters.append(chapter)

    # Create TOC and spine
    if profile.structure.include_toc:
        book.toc = tuple(chapters)
        book.add_item(epub.EpubNcx())
        book.add_item(epub.EpubNav())

    book.spine = ['nav'] + chapters

    # Write file
    output_path = self.get_output_path(profile)
    epub.write_epub(str(output_path), book, {})

    return str(output_path)
```

PDF Engine

Uses WeasyPrint to convert HTML+CSS into professional PDF output with proper page breaks, margins, and print-ready formatting.

Print-Specific Features

- Mirror margins for left/right pages
- Recto/verso page breaks for chapters
- Running headers with chapter titles
- Roman numerals for front matter
- Proper orphan/widow control
- Bleed settings for print

DOCX Engine

Generates Microsoft Word-compatible files using `python-docx`, preserving formatting and enabling further editing.

Manuscript Mode

When generating manuscripts, the engine applies industry-standard formatting:

- 12pt Courier font (monospaced)
- Double-spacing throughout
- 1-inch margins on all sides
- Header with author name and page number
- Word count on title page
- Contact information block

Content Processor

The ContentProcessor is responsible for gathering and preparing manuscript content for export, handling different scope modes:

Scope Mode	Description	Use Case
entire_project	All chapters and scenes	Full manuscript export
chapters	Specific chapter selection	Partial export, sample chapters
scenes	Individual scene selection	Excerpt generation
document	Single document	Blog post, short story
range	Chapter range	Book sections (Part I, etc.)

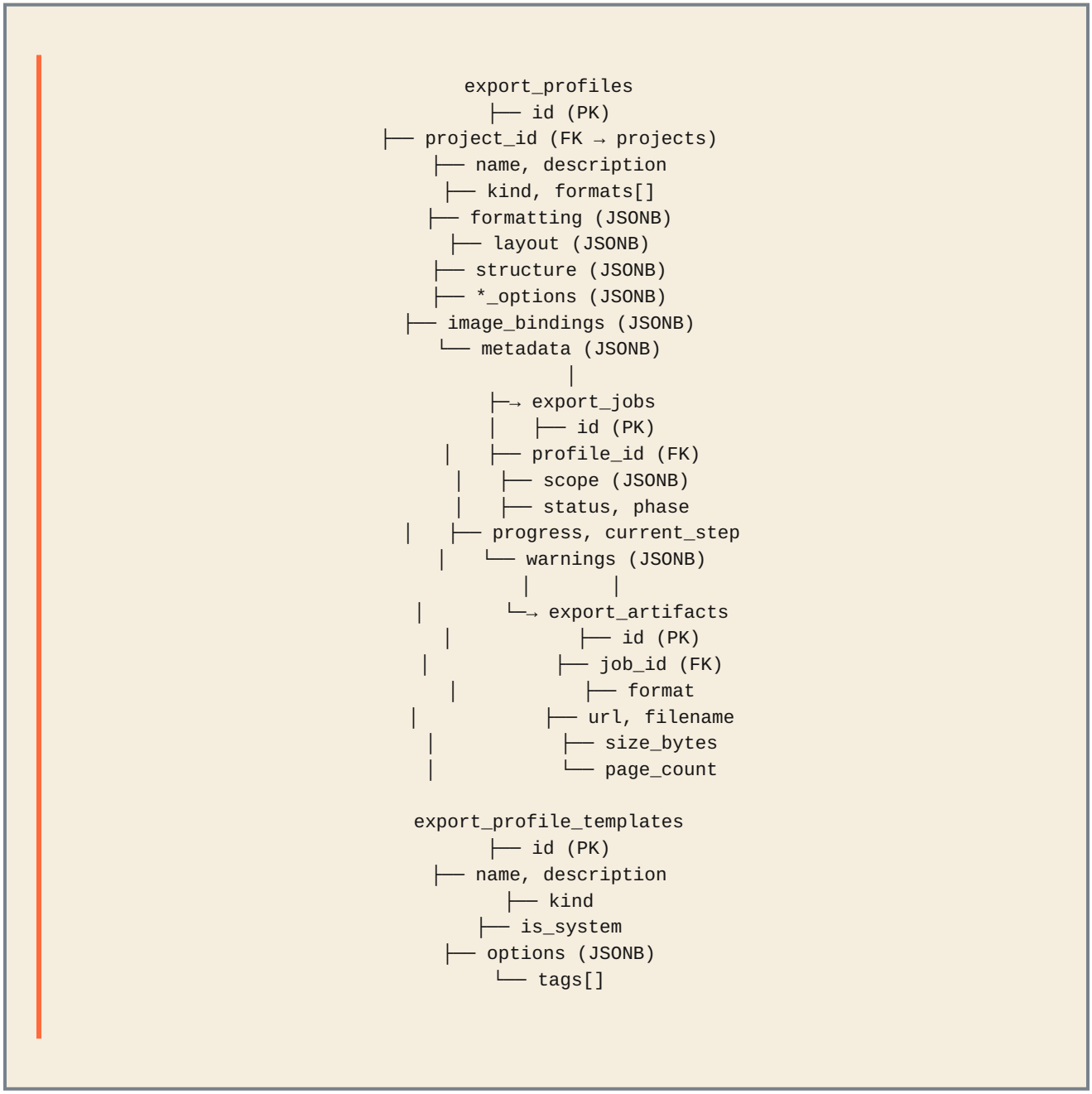
> **Integration Note:** The ContentProcessor needs to interface with your Smithy content storage system. The current implementation includes mock data for testing. You'll need to implement the actual database queries or file system reads to load chapter/scene content from your project structure.

Stage 4: Database Schema & Migrations

Database Design

The database schema uses PostgreSQL with four core tables that track profiles, jobs, artifacts, and templates.

Schema Overview



JSONB Columns

PostgreSQL's JSONB type provides flexible storage for complex nested structures while maintaining query performance and data validation:

Column	Contains	Why JSONB
`formatting`	Font, spacing, alignment	Flexible, evolving structure
`layout`	Page size, margins	Variable by format
`structure`	TOC, front/back matter	Boolean flags and settings
`*_options`	Kind-specific settings	Only one set populated per profile

SQLAlchemy Models

The ORM layer uses SQLAlchemy with async support for efficient database operations.

backend-python/tempering/db/models.py

```
class ExportProfileDB(Base):
    __tablename__ = "export_profiles"

    id = Column(String(50), primary_key=True)
    project_id = Column(String(50), nullable=False, index=True)

    name = Column(String(255), nullable=False)
    description = Column(Text, nullable=True)

    kind = Column(SQLEnum(ExportKind), nullable=False)
    formats = Column(ARRAY(SQLEnum(ExportFormat)), nullable=False)

    formatting = Column(JSON, nullable=False)
    layout = Column(JSON, nullable=False)
    structure = Column(JSON, nullable=False)

    # Relationships
    jobs = relationship("ExportJobDB", back_populates="profile")
```

Repository Pattern

Repositories abstract database access, providing clean APIs for CRUD operations:

backend-python/tempering/db/repositories.py

```
class ProfileRepository:
    async def create(self, profile: ExportProfile) -> ExportProfile:
        """Create new export profile"""

    async def get_by_id(self, profile_id: str) -> Optional[ExportProfile]:
        """Get profile by ID"""

    async def list_by_project(self, project_id: str) -> List[ExportProfile]:
        """List profiles for a project"""

    async def update(self, profile_id: str, updates: dict) -> bool:
        """Update profile"""

    async def delete(self, profile_id: str) -> bool:
        """Delete profile"""
```

Migration Management

Database migrations are handled with Alembic for version-controlled schema changes:

```
# Create a new migration
alembic revision --autogenerate -m "Add export templates"

# Apply migrations
alembic upgrade head

# Rollback
alembic downgrade -1
```

Seed Data

The initial migration includes five system templates covering common use cases:

- **Standard Manuscript:** Courier, double-spaced, industry format
- **Trade Paperback 6x9:** Garamond, print-ready novel
- **Kindle/EPUB:** Optimized for e-readers
- **Blog Post (Markdown):** YAML front matter for static sites
- **APA Research Paper:** Academic paper formatting

Database Setup:

```
# Start PostgreSQL
docker-compose up -d postgres

# Initialize database
python scripts/manage_db.py init

# Seed templates
python scripts/manage_db.py seed
```

Integration Guide

Wiring Everything Together

This section explains how to connect all the pieces: SolidJS frontend, Tauri commands, Python service, and PostgreSQL database.

1. Start the Python Service

```
cd backend-python
pip install -r requirements.txt --break-system-packages

# Start PostgreSQL
docker-compose up -d postgres

# Initialize database
python scripts/manage_db.py init
python scripts/manage_db.py seed

# Run FastAPI server
uvicorn tempering.main:app --reload --port 8000
```

2. Configure Tauri Commands

Add Tauri commands to bridge the frontend and Python service:

src-tauri/src/commands/tempering.rs

```
#[tauri::command]
pub async fn get_export_profiles(project_id: String) -> Result<Vec<ExportProfile>, String> {
    let client = reqwest::Client::new();
    let response = client
        .get(format!("http://localhost:8000/api/profiles?project_id={}", project_id))
        .send()
        .await
        .map_err(|e| e.to_string())?;

    response.json()
        .await
        .map_err(|e| e.to_string())
}

#[tauri::command]
pub async fn create_export_job(
    request: CreateExportJobRequest
) -> Result<ExportJob, String> {
    let client = reqwest::Client::new();
    let response = client
        .post("http://localhost:8000/api/jobs")
        .json(&request)
        .send()
        .await
        .map_err(|e| e.to_string())?;

    response.json()
        .await
        .map_err(|e| e.to_string())
}
```

3. Update Tauri Config

src-tauri/tauri.conf.json

```
{
  "tauri": {
    "allowlist": {
      "http": {
        "all": true,
        "scope": ["http://localhost:8000/**"]
      }
    }
  }
}
```

4. Add Route to SolidStart

src/app.tsx

```
import TemperingPage from "~/routes/tempering/index";

const routes = [
  // ... other routes
  {
    path: "/tempering",
    component: TemperingPage,
  },
];
```

5. Test the Integration

```
# Terminal 1: Start PostgreSQL
docker-compose up postgres

# Terminal 2: Start Python service
uvicorn tempering.main:app --reload

# Terminal 3: Start Tauri dev
cd /path/to/authorforge
bun run tauri dev
```

Deployment Checklist

- ✓ PostgreSQL configured with proper credentials
- ✓ Python service running on accessible port
- ✓ Tauri HTTP allowlist includes service URL
- ✓ Database migrations applied
- ✓ System templates seeded
- ✓ Export output directory writable
- ✓ Required fonts installed (Garamond, Courier, etc.)

Environment Configuration

.env

```
DATABASE_URL=postgresql+asyncpg://authorforge:password@localhost:5432/authorforge
TEMPERING_SERVICE_URL=http://localhost:8000
EXPORT_OUTPUT_DIR=/path/to/exports
```

Production Considerations

Security

- Use HTTPS for production Python service
- Implement authentication/authorization
- Validate all user inputs
- Sanitize file paths to prevent directory traversal
- Rate limit export job creation

Performance

- Use connection pooling for database
- Implement job queue (Celery or similar)
- Cache frequently-used templates
- Optimize image processing
- Consider CDN for artifact delivery

Monitoring

- Log all export job starts/completions
- Track export success/failure rates
- Monitor service health
- Alert on failed exports
- Measure average export times per format

Next Steps:

1. Test each export format with sample content
2. Validate EPUB output with epubcheck
3. Verify PDF page breaks and margins
4. Test DOCX compatibility with Microsoft Word
5. Integrate with your existing Smithy content storage
6. Add user-facing documentation
7. Implement error recovery and retry logic

Support & Resources

- **SolidJS Docs:** https://docs.solidjs.com
- **Tauri Docs:** https://tauri.app/v1/guides/
- **FastAPI Docs:** https://fastapi.tiangolo.com
- **WeasyPrint Docs:** https://weasyprint.org
- **ebooklib Docs:** https://github.com/aerkalov/ebooklib
- **python-docx Docs:** https://python-docx.readthedocs.io

> ⚒️ May your exports be swift and your books be legendary ⚒️