Tenant Dashboard Architecture Documentation

System Overview

The Tenant Dashboard is a Flask-based web application that provides automated rental agreement management through OCR text extraction, AI-powered information parsing, and intelligent alert systems. The application is designed to handle scanned PDF rental agreements and extract structured data for property management purposes.

Architecture Components

Core Framework and Dependencies

Flask Web Framework

- Primary web application framework handling HTTP requests and responses
- Template rendering system for HTML views
- File upload handling with Werkzeug utilities
- Route management for application endpoints

External Libraries and Services

- `pytesseract`: OCR engine for text extraction from PDF images
- `pdf2image`: PDF to image conversion utility
- `PIL (Pillow)`: Image processing capabilities
- `openai`: GPT-4o integration for intelligent text parsing
- `werkzeug`: Secure filename handling and file operations

Data Storage Architecture

File-Based Storage System

- `agreements_data.json`: Primary data store for active rental agreements
- `archived_agreements.json`: Secondary storage for deleted/archived agreements
- `uploads/`: Directory for temporary PDF file storage during processing

Data Persistence Strategy

- JSON-based storage for simplicity and human readability
- Automatic backup through archive system
- No external database dependencies

Core Functions and Data Flow

PDF Processing Pipeline

1. File Upload and Validation

```
def allowed_file(filename):

    return "." in filename and filename.rsplit(".", 1)[1].lower() in
ALLOWED_EXTENSIONS

Restricts uploads to PDF files only
Uses secure filename handling to prevent path traversal attacks
```

2. PDF to Image Conversion

```
def extract_text_from_pdf(pdf_path):

    images = convert_from_path(pdf_path)

    extracted_text = ""

    for image in images:

        text = pytesseract.image_to_string(image)

        extracted_text += text + "\n"

    return extracted_text
```

- Converts multi-page PDFs to individual images
- Processes each page through OCR for text extraction
- Concatenates extracted text from all pages

3. AI-Powered Information Extraction

```
def extract_information_with_gpt4o(text):

    prompt = (
        "Extract the following details from this rental agreement and return them as a single JSON object with these keys: "

        '"tenant_name", "place_occupied", "period_of_rent", "rent_amount", "maintenance", "rent_escalation", '

        '"agreement_start_date", "agreement_expiry_date", "lock_in_period", "lock_in_period_end_date", '

        '"rental_period_greater_than_lock_in_period", "next_rent_escalation". '

        "For dates, use YYYY-MM-DD format. If a value is not found, use an empty string. Only return the JSON object, nothing else.\n\n"

        f"{text}"

    )
```

- Uses structured prompting to extract specific rental agreement fields
- Enforces consistent data format through prompt engineering
- Handles missing data gracefully with empty string fallbacks

Alert System Architecture

Alert Status Calculation

```
def calculate_alert_status(lock_in_end_date):

    # Multiple date format parsing

    date_formats = [

        "%Y-%m-%d", "%d/%m/%Y", "%m/%d/%Y", "%d-%m-%Y",

        "%Y/%m/%d", "%m-%d-%Y", "%d.%m.%Y", "%Y.%m.%d",

        "%B %d, %Y", "%d %B %Y", "%b %d, %Y", "%d %b %Y"

    ]
    # Alert logic based on time windows
    if today <= one_month_before:
        return ""  # No alert
    elif one_month_before < today <= lock_in_date:
        return "approaching"  # Green alert
    elif lock_in_date < today <= one_month_after:
        return "grace_period"  # Gray alert
    elif today > one_month_after:
        return "overdue"  # Red alert
```

Alert Status Categories

- **Approaching**: Within one month of lock-in period end (green highlight)
- **Grace Period**: Between lock-in end and one month after (gray highlight)
- **Overdue**: More than one month past lock-in period (red highlight)
- **No Alert**: More than one month away from lock-in period end

Data Management Functions

Agreement Lifecycle Management

```
def add_unique_id(agreement):
    agreement["id"] = datetime.now().strftime("%Y%m%d_%H%M%S_%f")[:-3]
    agreement["upload_timestamp"] = datetime.now().isoformat()
    return agreement
def archive_agreement(agreement):
    agreement["archived_timestamp"] = datetime.now().isoformat()
    archived.append(agreement)

    save_archived_agreements(archived)
```

Unique Identifier System

- Timestamp-based IDs with millisecond precision
- Format: `YYYYMMDD_HHMMSS_mmm`
- Prevents ID collisions in high-frequency upload scenarios

Application Routes and Endpoints

Main Dashboard Route

```
@app.route("/", methods=["GET", "POST"])

def dashboard():

    agreements = load_agreements()



    # Update alert statuses for all agreements

    for agreement in agreements:

        agreement["alert_status"] = calculate_alert_status(

            agreement.get("lock_in_period_end_date", "")

        )



    if request.method == "POST":

        # File upload and processing logic

        file = request.files["file"]
        if file and allowed_file(file.filename):

            # Process uploaded PDF

            filename = secure_filename(file.filename)

            filepath = os.path.join(app.config["UPLOAD_FOLDER"], filename)

            file.save(filepath)


            # Extract and process data

            full_text = extract_text_from_pdf(filepath)

            data = extract_information_with_gpt4o(full_text)

            data = add_unique_id(data)


            # Save to storage

            agreements.append(data)

            save_agreements(agreements)


    return render_template("dashboard.html", agreements=agreements)
```

Route Functionality

- GET: Displays current agreements with updated alert statuses
- POST: Handles PDF uploads and processes new agreements
- Automatic alert status recalculation on each request

Archive Management Routes

Archive View

```
@app.route("/archive")

def archive():

    archived_agreements = load_archived_agreements()

    archived_agreements.sort(key=lambda x: x.get("archived_timestamp", ""),
reverse=True)

    return render_template("archive.html", agreements=archived_agreements)
```

Agreement Deletion (Archive)

```
@app.route("/delete_agreement/<agreement_id>", methods=["POST"])

def delete_agreement(agreement_id):

    agreements = load_agreements()

    agreement_to_archive = None



    # Find and archive agreement

    for agreement in agreements:

        if agreement.get("id") == agreement_id:

            agreement_to_archive = agreement

            break



    if agreement_to_archive:

        archive_agreement(agreement_to_archive)

        agreements = [a for a in agreements if a.get("id") != agreement_id]
        save_agreements(agreements)


    return redirect("/")
```

Agreement Restoration

```
@app.route("/restore_agreement/<agreement_id>", methods=["POST"])

def restore_agreement(agreement_id):

    # Restore logic with timestamp management

    if "archived_timestamp" in agreement_to_restore:

        del agreement_to_restore["archived_timestamp"]

    agreement_to_restore["restored_timestamp"] = datetime.now().isoformat()
```

Testing and Development Routes

Test Alert System

```
@app.route("/test_alert")

def test_alert():

    test_agreements = [

        # Predefined test data with various alert statuses

        # Tests approaching, overdue, and grace period scenarios

    ]


    for i, agreement in enumerate(test_agreements):

        agreement["alert_status"] = calculate_alert_status(
            agreement["lock_in_period_end_date"]

        )

        agreement["id"] = f"test_{i+1}"



    return render_template("dashboard.html", agreements=test_agreements)
```

Data Models and Schema

Agreement Data Structure

```
{

  "tenant_name": "string",

  "place_occupied": "string",

  "period_of_rent": "string",

  "rent_amount": "string",

  "maintenance": "string",

  "rent_escalation": "string",

  "agreement_start_date": "YYYY-MM-DD",

  "agreement_expiry_date": "YYYY-MM-DD",

  "lock_in_period": "string",

  "lock_in_period_end_date": "YYYY-MM-DD",

  "rental_period_greater_than_lock_in_period": "string",

  "next_rent_escalation": "YYYY-MM-DD",
  "alert_status": "string",

  "id": "timestamp_id",

  "upload_timestamp": "ISO_timestamp"

}
```

Metadata Fields

- **id**: Unique identifier for agreement tracking
- **upload_timestamp**: When the agreement was processed
- **archived_timestamp**: When the agreement was archived (if applicable)
- **restored_timestamp**: When the agreement was restored (if applicable)

Frontend Architecture

Template System

- **dashboard.html**: Main application interface with agreement table
- **archive.html**: Archive management interface
- Bootstrap 5.3.0 for responsive design and styling
- Bootstrap Icons for visual elements

Alert Visualization

```
.alert-approaching {

    background-color: #d4edda !important;

    color: #155724;

}

.alert-grace_period {

    background-color: #e2e3e5 !important;

    color: #383d41;

}
```

```
.alert-overdue {

    background-color: #f8d7da !important;

    color: #721c24;

}
```

Dynamic Styling

- CSS classes applied based on alert status
- Color-coded visual indicators for different alert levels
- Responsive table design for mobile compatibility

Error Handling and Logging

Comprehensive Logging System

```
logging.basicConfig(level=logging.DEBUG)

# Debug logging throughout the application

logging.debug(f"Loaded {len(agreements)} existing agreements")

logging.error(f"Error loading agreements: {e}")

logging.warning(f"Agreement with ID {agreement_id} not found")
```

Log Levels and Usage

- DEBUG: Detailed information for development and troubleshooting
- ERROR: Application errors and exception details
- WARNING: Non-critical issues and edge cases

Exception Handling

```
try:

    # Operation logic

    pass
```

```
except Exception as e:

    logging.error(f"Error description: {e}")

    # Fallback behavior or user notification
```

Error Recovery Strategies

- Graceful degradation when operations fail
- User-friendly error messages
- Automatic retry mechanisms where appropriate

Configuration and Environment

Environment Variables

```
openai.api_key = os.getenv("OPENAI_API_KEY")
```

Required Configuration

- OPENAI_API_KEY: API key for GPT-4o integration
- No additional configuration files required

File Paths and Directories

```
UPLOAD_FOLDER = "uploads"

DATA_FILE = "agreements_data.json"

ARCHIVE_FILE = "archived_agreements.json"
```

Directory Structure

- `uploads/`: Temporary PDF storage during processing
- `templates/`: HTML template files
- `static/`: CSS, JavaScript, and image assets

Performance Considerations

File Processing Optimization

- PDF processing occurs only during upload
- OCR results cached in memory during session
- No repeated processing of existing agreements

Memory Management

- Agreements loaded on-demand from JSON files
- Automatic cleanup of temporary upload files
- Efficient data structures for large agreement collections

Security Features

File Upload Security

- File type validation (PDF only)
- Secure filename handling with `secure_filename()`
- Path traversal prevention

Data Validation

- Input sanitization through AI parsing
- Structured data output validation
- No direct user input to database operations

Deployment and Scaling

Development Server

```
if __name__ == "__main__":

    if not os.path.exists(UPLOAD_FOLDER):

        os.makedirs(UPLOAD_FOLDER)

    app.run(debug=True)
```

Production Considerations

- Debug mode disabled for production
- WSGI server configuration required
- Environment variable management for API keys
- File storage optimization for high-volume scenarios

Scalability Factors

- File-based storage suitable for small to medium deployments
- Database migration path available for larger scale
- Stateless application design for horizontal scaling
- Caching strategies for improved performance

Development Workflow

Code Organization

- Single-file application for simplicity
- Modular function design for maintainability
- Clear separation of concerns between data, logic, and presentation

Testing Strategy

- Built-in test route for alert system validation
- Comprehensive logging for debugging
- Error handling for robust operation

Extension Points

- Additional alert types can be added to `calculate_alert_status()`
- New data fields can be integrated into AI extraction
- Additional routes can be added for enhanced functionality

This architecture provides a solid foundation for rental agreement management with clear separation of concerns, robust error handling, and extensible design patterns suitable for both development and production environments.