WorkZone: Job Application Management System

Technical Documentation

Information Retrieval Project

May 22, 2025

Contents

1 Introduction

1.1 Project Overview

WorkZone is a comprehensive job application management system designed to streamline the job search and application process. It integrates web scraping, data management, and application automation into a cohesive platform. This document details the system's architecture, implementation choices, and core functionality.

1.2 Motivation

The job application process often involves:

- Searching across multiple job platforms
- Manually tracking applications
- Repeatedly entering the same information
- Managing resume versions

We developed WorkZone to address these pain points by automating and centralizing the entire process, providing a single interface for managing all job search activities.

2 System Architecture

2.1 Component Overview

The system is built around five core components:

System Components

1. **Web Scrapers**: Collection of specialized scrapers for different job platforms
2. **Data Storage**: SQLite database with job and resume information
3. **Automation Engine**: Platform-specific application submission automation
4. **Web Interface**: Streamlit-based user interface
5. **Utilities**: Support functions for logging and data management

2.2 Data Flow

1. Web scrapers collect job listings from multiple sources
2. Jobs are stored in a centralized SQLite database
3. Users search and filter jobs via the Streamlit interface
4. Users upload and manage resumes via the interface
5. The automation engine applies to selected jobs with user credentials

3 Implementation Details

3.1 Web Scrapers

We implemented specialized scrapers for seven different job platforms, each tailored to the specific HTML structure and data patterns of its target site.

3.1.1 Design Choices

We chose to implement individual scraper classes for each platform to maximize flexibility and maintainability:

```
# From crawler.py
scraper_classes = [
    LinkedInScraper,
    FreelancerScraper,
    WuzzufScraper,
    RemoteOKScraper,
```

```
    WeWorkRemotelyScraper,
    UpworkScraper,
    PeoplePerHourScraper
]
```

Listing 1: Scraper Class Structure

Each scraper implements a common interface with a `scrape()` method, allowing for consistent integration with the main crawler.

3.1.2 Concurrent Execution

To optimize data collection, we implemented concurrent scraping using Python's `ThreadPoolExecutor`:

```
# From crawler.py
with concurrent.futures.ThreadPoolExecutor(max_workers=len(
    scraper_classes)) as executor:
    futures = {executor.submit(run_scraper, scraper_class,
    query, db_name): scraper_class.__name__
                for scraper_class in scraper_classes}
    for future in concurrent.futures.as_completed(futures):
        scraper_name = futures[future]
        try:
            future.result()
            logging.info(f"{scraper_name} completed
    successfully")
        except Exception as e:
            logging.error(f"{scraper_name} generated an
    exception: {e}")
```

Listing 2: Concurrent Scraping Implementation

This approach offers several advantages:

- Parallel execution reduces total scraping time
- Each scraper runs independently with its own database connection
- Error handling isolates failures to individual scrapers

3.1.3 Platform-Specific Implementations

Each scraper is tailored to its target platform. For example, the LinkedIn scraper handles authentication, pagination, and job extraction specific to LinkedIn's structure:

```
# From linkedin.py
class LinkedInScraper:
```

```
    def __init__(self, storage, query="software engineer"):
        self.storage = storage
        self.query = query
        self.driver = webdriver.Chrome()

    def scrape(self, max_pages=5):
        # LinkedIn-specific implementation
        # ...
```

Listing 3: LinkedIn Scraper (Excerpt)

3.2 Data Management

3.2.1 Job Model

We created a standardized `Job` class to normalize data across platforms:

```
# From models.py
class Job:
    def __init__(self, title=None, description=None, link=None, company=None,
                 source=None, timestamp=None, location=None):
        self.id = str(uuid.uuid4())
        self.title = title or 'non'
        self.description = description or 'non'
        self.link = link or 'non'
        self.company = company or 'non'
        self.source = source or 'non'
        self.timestamp = timestamp or datetime.now().isoformat()
        self.location = location or 'non'
```

Listing 4: Job Data Model

Key design decisions:

- UUID-based primary key for database integrity
- Default values to ensure data consistency
- ISO format timestamp for standardization
- Common fields across all job platforms

3.2.2 Data Storage

We chose SQLite for data persistence due to its simplicity, portability, and performance:

```python
# From models.py
class DataStorage:
    def __init__(self, output_format='sqlite', db_name='jobs.db'):
        self.jobs = []
        self.output_format = output_format
        self.db_name = db_name
        self.conn = sqlite3.connect(self.db_name)
        self.create_table()
```

Listing 5: DataStorage Implementation

The `DataStorage` class provides an abstraction layer over the database operations:

- Table creation and schema management
- Job insertion with duplicate handling
- Connection management
- Support for multiple output formats (CSV, JSON)

3.2.3 Deduplication

To prevent duplicate job listings, we implemented a deduplication utility:

```python
# From utils.py
def deduplicate_jobs(conn):
    with conn:
        conn.execute("""
            DELETE FROM jobs
            WHERE rowid NOT IN (
                SELECT MIN(rowid)
                FROM jobs
                GROUP BY link
            )
        """)
```

Listing 6: Deduplication Implementation

This SQL query efficiently removes duplicates while keeping the earliest record (by rowid) for each unique job link.

3.3 Automation Engine

3.3.1 Platform-Specific Appliers

We developed automation modules for three major platforms:

- LinkedIn
- Wuzzuf
- Freelancer

Each automation module handles the specific workflow of its target platform:

```
# From job_applier.py
def apply_to_job(job, resume_path, labels, credentials):
    source = job['source'].lower()
    if source == 'linkedin':
        return apply_to_linkedin_job(job['link'], resume_path, labels,
                                     credentials['linkedin_email'],
                                     credentials['linkedin_password'])
    elif source == 'wuzzuf':
        return apply_to_wuzzuf_job(job['link'], resume_path, labels,
                                   credentials['wuzzuf_email'],
                                   credentials['wuzzuf_password'])
    # ...
```

Listing 7: Job Application Automation

3.3.2 Automation Techniques

The automation modules use Selenium WebDriver to interact with job platforms:

- Automated form filling
- Resume upload
- Authentication handling
- Application status tracking

```
# From linkedin_applier.py
def apply_to_linkedin_job(job_url, resume_path, labels, email, password):
    driver = webdriver.Chrome()
    try:
```

```
        # Login to LinkedIn
        driver.get("https://www.linkedin.com/login")
        # ... authentication logic ...

        # Navigate to job
        driver.get(job_url)
        # ... application logic ...

        return True, "Application submitted successfully."
    except Exception as e:
        return False, f"Failed to apply: {str(e)}"
    finally:
        driver.quit()
```

Listing 8: LinkedIn Application Automation (Excerpt)

3.4 User Interface

3.4.1 Framework Selection

We chose Streamlit for the user interface due to its:

- Rapid development capabilities
- Interactive widgets
- Python integration
- Minimal frontend code requirements

3.4.2 Main Application Structure

The main application provides navigation and authentication:

```
# From main.py
import streamlit as st
from streamlit_option_menu import option_menu
from auth import login, logout, get_current_user

st.set_page_config(page_title="Job Application Manager")

# User authentication
user = get_current_user()
if not user:
    login()
    st.stop()

# Navigation menu
```

```
selected = option_menu(
    "Main Menu",
    ["Job Search", "Saved Jobs", "Applications", "Resume Manager",
     "Analytics", "Settings"],
    icons=["search", "bookmark", "check2-circle", "file-earmark-person",
           "bar-chart", "gear"],
    menu_icon="cast",
    default_index=0,
    orientation="horizontal"
)

# Page routing
if selected == "Job Search":
    from my_pages.job_search import job_search_page
    job_search_page()
# ... other pages ...
```

Listing 9: Main Streamlit Application

3.4.3 Advanced Search Implementation

We implemented an advanced search interface with multiple filtering options:

```
# From job_search.py
# Advanced Filters
with st.expander("Advanced Search Options", expanded=True):
    col1, col2, col3 = st.columns(3)
    with col1:
        title_filter = st.text_input("Job Title")
        # Multi-source selection
        all_sources = sorted(df["source"].unique().tolist())
        source_filter = st.multiselect("Source(s)", ["All"] + all_sources,
                                       default=["All"])
    # ... other filters ...

    # Date range filter
    parsed_timestamps = pd.to_datetime(df["timestamp"], errors="coerce",
                                       format='mixed')
    min_date = parsed_timestamps.min()
    max_date = parsed_timestamps.max()
    date_range = st.slider(
        "Date Range (Timestamp)",
        min_value=min_date,
        max_value=max_date,
        value=(min_date, max_date),
```

```
        format="YYYY-MM-DD"
    )
```

Listing 10: Advanced Search Implementation

3.4.4 Resume Management

The resume manager page enables:

- PDF/DOCX resume upload
- Resume preview
- Metadata management
- Information categorization

```
# From resume_manager.py
def resume_manager_page():
    st.header("Resume Manager")
    user = get_current_user()

    # ... database setup ...

    # Upload resume section
    uploaded_file = st.file_uploader("Upload or Update Resume",
                                     type=["pdf", "docx"],
                                     key=f"resume_upload_{user}")

    # ... resume processing ...

    # Display form for labels
    st.subheader("Resume Information")

    # Group labels by category
    categories = {
        "Personal Information": ["name", "age", "national_id", ...],
        "Professional Summary": ["overview", "career_objective"],
        # ... other categories ...
    }

    # Create tabs for different categories
    category_tabs = st.tabs(list(categories.keys()))
```

Listing 11: Resume Manager Implementation (Excerpt)

4 Technical Challenges and Solutions

4.1 Challenge: Concurrent Database Access

Problem: Multiple scrapers attempting to write to the same database simultaneously caused connection conflicts.

Solution: We implemented a separate database connection for each scraper thread:

```
# From crawler.py
def run_scraper(scraper_class, query, db_name):
    storage = DataStorage(output_format='sqlite', db_name=
    db_name)
    scraper = scraper_class(storage, query=query)
    scraper.scrape(max_pages=5)
    del storage  # Ensure connection is closed
    return scraper_class.__name__
```

Listing 12: Independent Database Connections

4.2 Challenge: Duplicate Job Listings

Problem: The same job might appear on multiple platforms or in multiple search results.

Solution: We implemented SQL-based deduplication using the job URL as the unique identifier:

```
DELETE FROM jobs
WHERE rowid NOT IN (
    SELECT MIN(rowid)
    FROM jobs
    GROUP BY link
)
```

Listing 13: Deduplication SQL

4.3 Challenge: Widget Key Conflicts in Streamlit

Problem: Multiple identical widgets caused ‘DuplicateWidgetID‘ errors in the UI.

Solution: We implemented dynamic key generation for all widgets:

```
# Unique file uploader key
uploaded_file = st.file_uploader(
    "Upload or Update Resume",
    type=["pdf", "docx"],
    key=f"resume_upload_{user}"
```

```
)

# Unique button keys with UUID
unique_id = str(uuid.uuid4())
st.button("Apply Now", key=f"apply_{row['id']}_{idx}_{
    unique_id}")
```

Listing 14: Dynamic Widget Keys

4.4 Challenge: Date Parsing in Timestamp Filtering

Problem: Inconsistent timestamp formats caused parsing errors.

Solution: We implemented robust timestamp parsing:

```
parsed_timestamps = pd.to_datetime(df["timestamp"],
                                   errors="coerce",
                                   format='mixed')
```

Listing 15: Robust Timestamp Parsing

5 Conclusion and Future Work

5.1 Achievements

The WorkZone system successfully:

- Integrates job data from seven major platforms
- Provides advanced search and filtering capabilities
- Manages resume uploads and metadata
- Automates application submission for three major platforms
- Centralizes job application tracking

5.2 Future Enhancements

Potential future developments include:

- Machine learning for job matching and recommendations
- Additional platform integrations
- Enhanced resume parsing and optimization
- API development for mobile applications
- Interview scheduling and preparation tools

5.3 Lessons Learned

This project reinforced several key principles:

- The importance of modular design for system extensibility
- Effective error handling for robust scraping
- Database optimization techniques for performance
- UI/UX considerations for complex data interaction
- The power of automation in streamlining repetitive tasks