AutoJobApply Technical Documentation

Table of Contents

Architecture Overview

AutoJobApply is built using a modern microservices architecture with the following components:

Frontend

- React-based Single Page Application (SPA)
- TypeScript for type safety
- Tailwind CSS for styling
- Vite for build tooling
- React Router for navigation
- Axios for API communication

Backend

- FastAPI for high-performance API

- Python 3.11+ for backend logic
- Selenium for web automation
- SQLite for data storage
- Pydantic for data validation

System Components

1. Job Board Integration Layer

```
class JobBoard(ABC):
    async def login(self, email: str, password: str) -> bool
    async def search_jobs(self, keywords: str, location: str) -> I
    async def apply_to_job(self, job_id: str, resume_path: str, co
```

The job board integration layer provides a unified interface for interacting with different job boards. Each job board implementation:

- Handles authentication
- Manages session state
- Implements job search functionality
- Handles job applications
- Manages browser automation

2. Application Service Layer

```
class JobService:
    def __init__(self):
        self.job_boards: Dict[str, Type[JobBoard]] = {
            "linkedin": LinkedInJobBoard,
            # Additional job boards
        }
```

The service layer:

- Manages job board instances

- Handles credential management
- Coordinates job searches
- Manages application processes
- Tracks application status

3. API Layer

```
@router.post("/jobs/search")
async def search_jobs(params: JobSearchParams) -> List[JobResponse
    return await job_service.search_jobs(params)
```

The API layer provides:

- RESTful endpoints for all operations
- Request validation
- Response formatting
- Error handling
- Rate limiting

Job Board Integration

LinkedIn Integration

```
class LinkedInJobBoard(JobBoard):
    BASE_URL = "https://www.linkedin.com"

    async def login(self, email: str, password: str) -> bool:
        # Implementation details
        pass

    async def search_jobs(self, keywords: str, location: str) -> I
        # Implementation details
        pass
```

Key features:

- Automated login process
- Job search with filters
- Easy Apply integration
- Form handling
- Error recovery

Browser Automation

```
def _setup_driver(self) -> webdriver.Chrome:
    chrome_options = Options()
    chrome_options.add_argument("--start-maximized")
    chrome_options.add_argument("--disable-notifications")
    # Additional options
```

Features:

- Headless mode support
- Custom user agent
- Anti-detection measures
- Error handling
- Resource cleanup

Security Implementation

1. Credential Management

- Environment variables for sensitive data
- Encrypted storage for credentials
- Secure session management
- Regular credential rotation

2. API Security

- CORS configuration

- Rate limiting
- Request validation
- Error handling
- HTTPS enforcement

3. Data Protection

- No sensitive data storage
- Secure file handling
- Input sanitization
- Output encoding

Database Schema

Applications Table

```
CREATE TABLE applications (
    id INTEGER PRIMARY KEY,
    job_id TEXT NOT NULL,
    job_board TEXT NOT NULL,
    status TEXT NOT NULL,
    applied_at TIMESTAMP NOT NULL,
    company TEXT NOT NULL,
    position TEXT NOT NULL,
    location TEXT NOT NULL,
    url TEXT NOT NULL
);
```

Settings Table

```
CREATE TABLE settings (
    id INTEGER PRIMARY KEY,
    key TEXT NOT NULL UNIQUE,
    value TEXT NOT NULL,
    updated_at TIMESTAMP NOT NULL
);
```

API Documentation

Job Search

```
 POST /api/v1/jobs/search
Content-Type: application/json

{
    "keywords": "software engineer",
    "location": "San Francisco",
    "job_board": "linkedin"
}
```

Job Application

```
 POST /api/v1/jobs/apply
Content-Type: application/json

{
    "job_id": "123456",
    "job_board": "linkedin"
}
```

Frontend Architecture

Component Structure

```
src/
├── components/
│   ├── Navbar.tsx
│   ├── JobCard.tsx
│   └── SettingsForm.tsx
├── pages/
│   ├── Dashboard.tsx
│   ├── JobSearch.tsx
│   └── Settings.tsx
└── services/
    ├── api.ts
    └── auth.ts
```

State Management

- React Query for server state
- Context API for global state
- Local storage for persistence

Deployment Guide

Backend Deployment

1. Set up Python environment
2. Install dependencies
3. Configure environment variables
4. Set up database
5. Deploy with uvicorn/gunicorn

Frontend Deployment

1. Build production assets
2. Configure environment
3. Deploy to static hosting
4. Set up CDN
5. Configure SSL

Production Considerations

- Use production-grade web server
- Enable HTTPS
- Set up monitoring
- Configure logging
- Implement backup strategy
- Set up CI/CD pipeline

Error Handling

Backend Errors

```
class JobBoardError(Exception):
    def __init__(self, message: str, job_board: str):
        self.message = message
        self.job_board = job_board
        super().__init__(self.message)
```

Frontend Error Handling

```
const handleError = (error: Error) => {
  if (error instanceof ApiError) {
    // Handle API errors
  } else if (error instanceof NetworkError) {
    // Handle network errors
  } else {
    // Handle unexpected errors
  }
};
```

Monitoring and Logging

Backend Logging

```
 logger = logging.getLogger(__name__)
logger.info("Starting job search")
logger.error("Failed to apply to job", exc_info=True)
```

Frontend Monitoring

- Error tracking
- Performance monitoring
- User analytics
- Usage statistics

Future Improvements

1. Additional Job Boards

2. Indeed

3. Glassdoor
4. GitHub Jobs

5. Stack Overflow Jobs

6. Enhanced Features

7. AI-powered job matching

8. Automated interview scheduling

9. Resume optimization

10. Application analytics

11. Performance Optimizations

12. Caching layer

13. Database indexing
14. Query optimization

15. Asset optimization

16. Security Enhancements

17. Two-factor authentication
18. API key rotation
19. Enhanced encryption
20. Security scanning