Python to TypeScript Migration Guide

Overview

This guide explains the key differences between the Python and TypeScript implementations of the DAF420 parser.

Installation & Setup

Python Version

```
pip install -r requirements.txt
python -m parser.cli -i input.dat -o output.csv
```

TypeScript Version

```
npm install
npm run build
npm run parse -- -i input.dat -o output.csv
```

API Comparison

Python API

```python
from parser import Config, PermitParser, CSVExporter

# Initialize
config = Config()
parser = PermitParser(config)

# Parse
permits, stats = parser.parse_file("input.dat")

# Export
exporter = CSVExporter(config)
exporter.export(permits, "output.csv")
```

TypeScript API

```typescript
import { Config, PermitParser, CSVExporter } from './src';

// Initialize
const config = new Config();
const parser = new PermitParser(config);

// Parse (async)
const { permits, stats } = await parser.parseFile('input.dat');

// Export (async)
const exporter = new CSVExporter(config);
await exporter.export(permits, 'output.csv');
```

Key Differences

1. Async/Await

Python (Synchronous)

```python
def parse_file(self, input_path: Path) -> Tuple[Dict, ParseStats]:
    with input_path.open() as f:
        for line in f:
            self._process_line(line)
    return permits, stats
```

TypeScript (Asynchronous)

```typescript
async parseFile(inputPath: string): Promise<{
  permits: Record<string, PermitData>;
  stats: ParseStats;
}> {
  // Async file reading
  return new Promise((resolve, reject) => { ... });
}
```

2. Type Annotations

Python (Runtime Hints)

```python
def validate(self, validator_name: str, value: str, context: str = "") -> bool:
    # Type hints are optional and not enforced
    pass
```

TypeScript (Compile-Time Enforcement)

```typescript
validate(
  validatorName: ValidatorType,  // Enforced enum type
  value: string,
  context: string = ''
): boolean {
  // Return type enforced at compile time
}
```

3. Data Structures

Python

```python
from collections import Counter, defaultdict

stats.records_by_type = Counter()
stats.record_lengths = defaultdict(list)
```

TypeScript

```typescript
stats.recordsByType = new Map<string, number>();
stats.recordLengths = new Map<string, number[]>();
```

4. Null Handling

Python (Optional)

```python
from typing import Optional

def parse_date(value: str) -> Optional[str]:
    if not value:
        return None
```

TypeScript (Union Types)

```typescript
function parseDate(value: string): string | null {
  if (!value) {
    return null;
  }
}
```

5. Class Definitions

Python (Dataclass)

```python
from dataclasses import dataclass, field

@dataclass
class Permit:
    permit_number: str
    daroot: Optional[Dict] = None
    dafield: List[Dict] = field(default_factory=list)
```

TypeScript (Class with Types)

```typescript
class Permit implements PermitData {
  daroot: DaRootRecord | null = null;
  dafield: DaFieldRecord[] = [];

  constructor(public readonly permitNumber: string) {}
}
```

File Structure Comparison

Python Structure

```
refactored_parser/
├── config.yaml
├── parser/
│   ├── __init__.py
│   ├── config.py
│   ├── models.py
│   ├── validators.py
│   ├── parser.py
│   ├── exporter.py
│   └── cli.py
└── tests/
```

TypeScript Structure

```
refactored_parser_ts/
├── config.yaml
├── src/
│   ├── types/          # NEW: Type definitions
│   ├── models/
│   ├── config/
│   ├── validators/
│   ├── parser/
│   ├── exporter/
│   ├── cli/
│   └── utils/
├── dist/               # Compiled output
└── tsconfig.json       # TypeScript config
```

Configuration Loading

Python (Pydantic)

```python
class Config:
    def __init__(self, config_path: Optional[Path] = None):
        with open(config_path, 'r') as f:
            self._raw_config = yaml.safe_load(f)

        self.settings = Settings(**self._raw_config.get('settings', {}))
```

TypeScript (Manual Validation)

```
class Config {
  constructor(configPath?: string) {
    const configContent = fs.readFileSync(finalPath, 'utf8');
    this.rawConfig = yaml.load(configContent) as RawConfigData;

    this.settings = {
      minRecordLength: 10,
      strictMode: false,
      encoding: 'latin-1',
      ...this.rawConfig.settings
    };
  }
}
```

Error Handling

Python

```
try:
    parsed = self._parse_record(record, rec_type, line_num)
except Exception as e:
    self._log_malformed(line_num, f"Parse error: {e}")
    if self.strict_mode:
        raise
```

TypeScript

```
try {
  const parsed = this.parseRecord(record, recType, lineNumber);
} catch (error) {
  this.stats.logMalformed(lineNumber, `Parse error: ${String(error)}`);
  if (this.strictMode) {
    throw error;
  }
}
```

Testing

Python (unittest)

```
import unittest

class TestParser(unittest.TestCase):
    def test_parse_date(self):
        result = parse_date("20200101")
        self.assertEqual(result, "01/01/2020")
```

TypeScript (Jest)

```typescript
import { parseDate } from './utils';

describe('parseDate', () => {
  it('should parse date correctly', () => {
    const result = parseDate('20200101');
    expect(result).toBe('01/01/2020');
  });
});
```

CLI Differences

Python

```bash
python -m parser.cli -i input.dat -o output.csv -v
```

TypeScript

```bash
npm run parse -- -i input.dat -o output.csv -v
# or after build:
node dist/cli/index.js -i input.dat -o output.csv -v
```

Performance Comparison

Aspect	Python	TypeScript
Startup	Faster (interpreted)	Slower (requires Node.js)
Runtime	Slower (GIL)	Faster (V8 JIT)
Memory	Higher (Python objects)	Lower (optimized V8)
I/O	Synchronous by default	Asynchronous by default

Type Safety Benefits

Python (Runtime)

```python
# No compile-time checking
permit_number = data["permit_number"]  # Could fail at runtime
```

TypeScript (Compile-Time)

```
// Compile-time type checking
interface DaPermitRecord {
  permit_number?: string;
}

const permitNumber: string | undefined = data.permit_number;
// TypeScript ensures type safety
```

Common Pitfalls

1. Async/Await Forgetting

❌ **Wrong**

```
const result = parser.parseFile('input.dat');  // Missing await
```

✅ **Correct**

```
const result = await parser.parseFile('input.dat');
```

2. Null Checking

❌ **Wrong**

```
const name = permit.dapermit.lease_name;  // May be null
```

✅ **Correct**

```
const name = permit.dapermit?.lease_name || '';
```

3. Type Assertions

❌ **Wrong**

```
const permitNum = data.permit_number as string;  // Unsafe
```

✅ **Correct**

```
const permitNum = (data.permit_number as string | undefined) || '';
```

Migration Checklist

- [ ] Install Node.js and npm
- [ ] Run `npm install` to install dependencies
- [ ] Copy `config.yaml` from Python version
- [ ] Update any custom validation rules

- [ ] Run `npm run build` to compile
- [ ] Run `npm test` to verify
- [ ] Update CI/CD pipelines
- [ ] Update documentation

Advantages of TypeScript Version

1. ✅ **Type Safety**: Catch errors at compile time
2. ✅ **IDE Support**: Better autocomplete and refactoring
3. ✅ **Performance**: Faster runtime with V8 engine
4. ✅ **Async I/O**: Better for large files
5. ✅ **Modern Syntax**: ES2020+ features
6. ✅ **Documentation**: Types serve as docs

When to Use Which Version

Use Python Version If:

- Working in Python ecosystem
- Need Pydantic validation
- Prefer synchronous code
- Team is Python-focused

Use TypeScript Version If:

- Working in Node.js ecosystem
- Need strong type safety
- Want better IDE support
- Prefer async/await
- Team is TypeScript-focused

Support

For issues or questions:
- Python version: See original documentation
- TypeScript version: See TypeScript-specific README.md and ARCHITECTURE.md