Course Equivalency Finder App

What This App Does

This application addresses a common challenge in higher education: determining how courses from one institution transfer to another. Students transferring between colleges need to understand which of their completed courses will count toward their degree at the new school. Academic advisors face the same challenge when helping students plan their educational pathways.

The Course Equivalency Finder serves as a centralized system that maps relationships between courses at different institutions. Students can browse course catalogs, identify transfer equivalencies, and create personalized transfer plans that can be saved and shared.

How Everything Fits Together

The application follows a three-tier architecture similar to most modern web applications. The database layer stores and organizes course information and relationships. The backend API layer provides structured access to this data through RESTful endpoints. The frontend presentation layer offers an intuitive interface for browsing and managing course information.

The Database

The application uses SQLite as its database engine, which provides relational data storage without requiring a separate database server. The schema consists of five interconnected tables that model the academic structure:

Institutions represent individual educational organizations such as "Delgado Community College" or "The University of New Orleans." Each institution has a unique identifier and name:

```
CREATE TABLE IF NOT EXISTS Institution (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT UNIQUE NOT NULL
)
```

Departments represent academic divisions within institutions. A "Computer Science" department exists separately at Delgado Community College and The University of New Orleans, even though they share the same name:

```
CREATE TABLE IF NOT EXISTS Department (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    institution_id INTEGER NOT NULL,
    FOREIGN KEY (institution_id) REFERENCES Institution (id),
    UNIQUE(name, institution_id)
)
```

The unique constraint on `(name, institution_id)` prevents duplicate departments within a single institution while allowing different institutions to have departments with the same name.

Courses represent individual academic offerings. Each course requires a code (such as "CS 101"), a descriptive title, and must belong to both a department and an institution:

```
CREATE TABLE IF NOT EXISTS Course (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    code TEXT NOT NULL,
    title TEXT NOT NULL,
    department_id INTEGER NOT NULL,
    institution_id INTEGER NOT NULL,
    FOREIGN KEY (department_id) REFERENCES Department (id),
    FOREIGN KEY (institution_id) REFERENCES Institution (id),
    UNIQUE(code, department_id, institution_id)
)
```

This structure ensures that "CS 101" at Delgado Community College and "CS 101" at The University of New Orleans are treated as distinct courses, which accurately reflects academic reality.

Course Equivalencies define transfer relationships between courses at different institutions:

```
CREATE TABLE IF NOT EXISTS CourseEquivalency (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    source_course_id INTEGER NOT NULL,
    target_course_id INTEGER NOT NULL,
    FOREIGN KEY (source_course_id) REFERENCES Course (id),
    FOREIGN KEY (target_course_id) REFERENCES Course (id),
    UNIQUE(source_course_id, target_course_id)
)
```

This design enables bidirectional equivalency relationships. When "Intro to Psychology" at Delgado Community College is marked as equivalent to "Psychology 101" at The University of New Orleans, the system automatically recognizes the relationship in both directions.

Transfer Plans allow users to save their course selections and planning work:

```
CREATE TABLE IF NOT EXISTS TransferPlan (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    code TEXT UNIQUE NOT NULL,
    plan_name TEXT NOT NULL,
    source_institution_id INTEGER NOT NULL,
    target_institution_id INTEGER NOT NULL,
    selected_courses TEXT NOT NULL,
    plan_data TEXT NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    ...
)
```

The `selected_courses` and `plan_data` fields use JSON formatting to store structured information without requiring additional normalized tables.

The Backend

The Flask backend implements a RESTful API that provides controlled access to the database. It functions as a data access layer, handling queries, validation, and business logic.

Getting Course Information

The API follows a hierarchical pattern that mirrors the academic structure. Users begin by selecting an institution:

```
@app.route('/api/institutions')
def get_institutions():
    cleanup_expired_plans()  # Maintenance task
    conn = get_db_connection()
    rows = conn.execute('SELECT * FROM Institution ORDER BY name').fetchall()
    conn.close()
    return jsonify([dict(r) for r in rows])
```

The `cleanup_expired_plans()` function performs routine maintenance by removing plans older than one year, similar to an automated file retention policy.

After selecting an institution, users can browse its departments:

```
@app.route('/api/departments')
def get_departments():
    institution_id = request.args.get('institution_id')
    conn = get_db_connection()
    rows = conn.execute('SELECT * FROM Department WHERE institution_id = ? ORDER
BY name',
                        (institution_id,)).fetchall()
    conn.close()
    return jsonify([dict(r) for r in rows])
```

Finally, users can view courses within a specific department:

```
@app.route('/api/courses')
def get_courses():
    department_id = request.args.get('department_id')
    conn = get_db_connection()
    rows = conn.execute('SELECT * FROM Course WHERE department_id = ? ORDER BY
code',
                        (department_id,)).fetchall()
```

```
    conn.close()
    return jsonify([dict(r) for r in rows])
```

Finding Course Equivalencies

The equivalency lookup system uses a sophisticated SQL query to find all courses related to a selected course:

```
@app.route('/api/equivalents')
def get_equivalents():
    course_id = request.args.get('course_id')
    conn = get_db_connection()
    rows = conn.execute('''
        SELECT c.*, i.name as institution_name, d.name as department_name
        FROM CourseEquivalency e
        JOIN Course c ON c.id = e.target_course_id
        JOIN Institution i ON i.id = c.institution_id
        JOIN Department d ON d.id = c.department_id
        WHERE e.source_course_id = ?
        UNION
        SELECT c.*, i.name as institution_name, d.name as department_name
        FROM CourseEquivalency e
        JOIN Course c ON c.id = e.source_course_id
        JOIN Institution i ON i.id = c.institution_id
        JOIN Department d ON d.id = c.department_id
        WHERE e.target_course_id = ?
        ORDER BY institution_name, department_name, code
    ''', (course_id, course_id)).fetchall()
```

The UNION operation combines two queries: one finding courses where the selected course serves as the source of an equivalency, and another where it serves as the target. This approach ensures complete bidirectional relationship discovery regardless of how the equivalency was originally entered.

Handling Data Import

Production systems must handle imperfect data gracefully. The CSV import system implements robust error handling and duplicate management:

```
def get_or_create(table, fields, values):
    """Get existing record or create new one, handling duplicates gracefully"""
    where_clause = " AND ".join([f"{field} = ?" for field in fields])
    select_query = f"SELECT id FROM {table} WHERE {where_clause}"

    row = cursor.execute(select_query, values).fetchone()
    if row:
        return row[0]  # Record exists, return its ID

    # Record doesn't exist, attempt to create it
    placeholders = ','.join(['?' for _ in fields])
    insert_query = f"INSERT INTO {table} ({','.join(fields)}) VALUES
```

```
({placeholders})"

    try:
        cursor.execute(insert_query, values)
        return cursor.lastrowid
    except sqlite3.IntegrityError as e:
        if "UNIQUE constraint failed" in str(e):
            # Handle race condition: record created between check and insert
            row = cursor.execute(select_query, values).fetchone()
            if row:
                return row[0]
            else:
                raise Exception(f"Could not create or find record in {table}")
        else:
            raise  # Re-raise other integrity errors
```

This implementation follows the "optimistic concurrency" pattern: attempt the operation and handle conflicts when they occur. If a CSV file references "Delgado Community College" multiple times, the system creates the institution record once and reuses it for subsequent references.

Managing Transfer Plans

Transfer plan management includes unique code generation for plan retrieval:

```
def generate_plan_code():
    """Generate a unique 8-character alphanumeric code"""
    return ''.join(secrets.choice(string.ascii_uppercase + string.digits) for _ in
range(8))
```

The `secrets` module provides cryptographically secure random number generation, ensuring that plan codes cannot be easily guessed or predicted.

Plan creation includes comprehensive validation:

```
@app.route('/api/create-plan', methods=['POST'])
def create_plan():
    try:
        data = request.get_json()

        # Validate required fields
        required_fields = ['plan_name', 'source_institution_id',
'target_institution_id', 'selected_courses']
        for field in required_fields:
            if field not in data:
                return jsonify({'error': f'Missing required field: {field}'}), 400

        # Generate unique code
        code = generate_plan_code()
        conn = get_db_connection()
```

```
        # Ensure uniqueness
        while conn.execute('SELECT id FROM TransferPlan WHERE code = ?',
(code,)).fetchone():
            code = generate_plan_code()
```

The collision detection loop handles the statistically unlikely case of generating a duplicate code, implementing a fail-safe mechanism for code uniqueness.

The Frontend

The React frontend provides an interactive interface for course browsing and transfer planning. The application uses modern functional components with hooks for state management.

State Management

The main application component manages multiple state variables that track user interactions and data:

```
const [institutions, setInstitutions] = useState([]);
const [departments, setDepartments] = useState([]);
const [courses, setCourses] = useState([]);
const [equivalents, setEquivalents] = useState([]);

const [expandedInstitution, setExpandedInstitution] = useState(null);
const [expandedDepartment, setExpandedDepartment] = useState(null);
const [selectedCourse, setSelectedCourse] = useState(null);

const [selectedCourses, setSelectedCourses] = useState([]);
const [planName, setPlanName] = useState('');
const [sourceInstitution, setSourceInstitution] = useState('');
const [targetInstitution, setTargetInstitution] = useState('');
const [currentView, setCurrentView] = useState('browse');
const [isEditMode, setIsEditMode] = useState(false);
```

The state variables are logically grouped: data storage arrays hold course information, UI state variables track user interface interactions, and planning variables manage transfer plan construction.

Progressive Data Loading

The application implements lazy loading to minimize initial load time and reduce unnecessary API calls:

```
useEffect(() => {
  if (expandedInstitution) {
    axios.get(`/api/departments?institution_id=${expandedInstitution}`)
      .then(res => setDepartments(res.data));
  } else {
    setDepartments([]);
    setCourses([]);
    setEquivalents([]);
```

```
  }
}, [expandedInstitution]);
```

When users expand an institution, the system loads its departments. When they collapse it, the system clears dependent data to maintain interface clarity. This pattern continues hierarchically:

```
useEffect(() => {
  if (expandedDepartment) {
    axios.get(`/api/courses?department_id=${expandedDepartment}`)
      .then(res => setCourses(res.data));
  } else {
    setCourses([]);
    setEquivalents([]);
  }
}, [expandedDepartment]);
```

Course Selection Logic

The course selection system implements toggle functionality with clear state management:

```
const toggleCourseSelection = (course) => {
  const isSelected = selectedCourses.some(c => c.id === course.id);
  if (isSelected) {
    setSelectedCourses(selectedCourses.filter(c => c.id !== course.id));
  } else {
    setSelectedCourses([...selectedCourses, course]);
  }
};
```

This function maintains an array of selected courses, adding or removing items based on current selection state. The interface provides visual feedback to indicate selection status.

Component Architecture

The application uses component composition to separate concerns and improve maintainability:

Navigation manages view transitions and displays contextual information:

```
const Navigation = ({
  currentView,
  setCurrentView,
  isEditMode,
  loadedPlan,
  selectedCourses,
  searchCode,
  setSearchCode,
  loadPlan,
```

```
  cancelEdit
}) => {
```

BrowseView handles the hierarchical course browsing interface:

```
const BrowseView = ({
  institutions,
  departments,
  courses,
  equivalents,
  expandedInstitution,
  setExpandedInstitution,
  // Additional props for complete functionality
}) => {
```

Each component receives specific props that define its data dependencies and available actions, promoting loose coupling and clear interfaces.

Edit Mode Implementation

The edit mode system provides a complete workflow for modifying existing transfer plans:

```
const startEditMode = () => {
  if (loadedPlan) {
    setPlanName(loadedPlan.plan_name);
    setSourceInstitution(loadedPlan.plan_data.source_institution_id);
    setTargetInstitution(loadedPlan.plan_data.target_institution_id);
    setSelectedCourses(loadedPlan.selected_courses);
    setIsEditMode(true);
    setCurrentView('edit-plan');
  }
};
```

The system populates form fields with existing data and switches to an editing interface with appropriate visual indicators.

Change detection prevents data loss from accidental navigation:

```
const cancelEdit = () => {
  const hasChanges =
    planName !== loadedPlan?.plan_name ||
    sourceInstitution !== loadedPlan?.plan_data?.source_institution_id ||
    targetInstitution !== loadedPlan?.plan_data?.target_institution_id ||
    selectedCourses.length !== loadedPlan?.selected_courses?.length;

  if (hasChanges) {
    const confirmCancel = window.confirm('You have unsaved changes. Are you sure
```

```
  you want to cancel editing?');
      if (!confirmCancel) return;
    }

    setIsEditMode(false);
    setCurrentView('view-plan');
  };
```

This implementation compares current form state with original data to detect modifications and requests user confirmation before discarding changes.

The Import Feature

Academic institutions maintain course data in various formats and systems. The CSV import feature accommodates this reality through robust error handling and detailed reporting:

```
# Process each row in the CSV
rows_processed = 0
rows_skipped = 0
errors = []

for row_num, row in enumerate(reader, start=1):
    try:
        # Validate required fields
        required_fields = [
            'source_institution', 'target_institution',
            'source_department', 'target_department',
            'source_code', 'source_title',
            'target_code', 'target_title'
        ]

        missing_fields = [field for field in required_fields if not row.get(field,
'').strip()]
        if missing_fields:
            rows_skipped += 1
            errors.append(f"Row {row_num}: Missing fields: {',
'.join(missing_fields)}")
            continue

        # Process valid rows
        rows_processed += 1

    except Exception as e:
        rows_skipped += 1
        errors.append(f"Row {row_num}: {str(e)}")
```

The system processes each row independently, collecting detailed error information for problematic records while successfully importing valid data. Users receive comprehensive reports that enable data correction and reprocessing.

Development and Deployment

The application uses modern development tools optimized for developer productivity and deployment simplicity:

```
// vite.config.js
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:5000',
        changeOrigin: true,
      }
    }
  }
})
```

Vite's proxy configuration routes API requests to the Flask backend during development, eliminating cross-origin resource sharing (CORS) complications and simplifying the development environment.

The Flask backend maintains simplicity through a single-file architecture using SQLite for data persistence, avoiding the complexity of separate database server management while providing full relational database capabilities.

Why This Architecture Works

The application succeeds by aligning technical complexity with problem complexity. Course equivalency tracking requires managing many-to-many relationships between academic entities, which relational databases handle naturally. The hierarchical browsing interface matches users' mental models of academic organization. The transfer planning feature addresses a real workflow need without introducing unnecessary complexity.

The system emphasizes data integrity and error recovery. Robust duplicate handling, comprehensive validation, and detailed error reporting distinguish a production-ready tool from a proof-of-concept demonstration. Users can recover from errors and understand system behavior, building confidence in the application's reliability.

The component-based frontend architecture enables independent development and maintenance of features. Course browsing functionality can be modified without affecting transfer plan management, and new features can be added through additional components without restructuring existing code.

The technology choices prioritize reliability and maintainability over cutting-edge features. SQLite provides enterprise-grade data storage without operational complexity. Flask offers a well-documented, stable web framework. React enables modern user interfaces with extensive community support. This combination

delivers robust functionality while remaining approachable for development teams with varying experience levels.