Course Equivalency Finder: How It All Works

What This App Does

Imagine you're a student trying to transfer from one college to another, and you need to figure out which of your current courses will count toward your degree at the new school. Or maybe you're an academic advisor helping dozens of students navigate this same challenge every semester. That's exactly what this app was built to solve.

The Course Equivalency Finder is like having a smart course catalog that knows how classes at different schools relate to each other. Students can browse through courses, see what transfers where, and even create personalized transfer plans they can save and share.

How Everything Fits Together

Think of this app like a well-organized library system. You've got your card catalog (the database) that keeps track of all the books (courses), which sections they belong to (departments), and which libraries have them (institutions). Then you have the friendly librarian (the backend) who knows how to find exactly what you're looking for, and a really nice reading room (the frontend) where you can comfortably browse and plan.

The Database

The heart of the app is a SQLite database - nothing fancy, but it gets the job done beautifully. Here's how we've organized all the course information:

Institutions are pretty straightforward - think "University of California" or "Community College of Denver." Each school gets its own record:

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

Departments live inside institutions. So you might have "Computer Science" at both UC Berkeley and UCLA, but they're separate records because they're different departments at different schools:

```
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(name, institution_id)` part is clever - it prevents us from accidentally creating duplicate "Computer Science" departments at the same school, but allows different schools to have their own CS departments.

Courses are where things get interesting. Every course needs a code (like "CS 101"), a title (like "Introduction to Programming"), and it needs to 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 setup means that "CS 101" at UC Berkeley and "CS 101" at UCLA are completely different courses in our system, which is exactly what we want.

Course Equivalencies are the secret sauce that makes this whole thing work. This table creates connections between courses that are considered equivalent for transfer purposes:

```
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)
)
```

The beauty of this design is that equivalencies work both ways automatically. If "Intro to Psychology" at School A is equivalent to "Psychology 101" at School B, then students can transfer in either direction.

Transfer Plans let students save their course selection work for later. Instead of having to rebuild their transfer plan every time they visit the app, they can save it with a shareable code:

```
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 store JSON, which gives us flexibility to save complex information without having to create a bunch of additional tables.

The Backend

The Flask backend acts like a knowledgeable academic advisor who never gets tired of answering questions. It knows how to fetch course information, find equivalencies, and manage transfer plans.

Getting Course Information

When someone wants to browse courses, they start by picking a school. The backend responds with a simple request:

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

Notice that little `cleanup_expired_plans()` call? That's the app being thoughtful - every time someone asks for institutions, it quietly cleans up any transfer plans older than a year. It's like having a filing system that automatically throws away old paperwork.

Once they pick a school, they can browse 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])
```

And then courses within those departments:

```
@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

Here's where the app really shines. When someone clicks on a course, the backend finds all the equivalent courses using a pretty clever SQL query:

```
@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` is the magic here. The first part of the query finds courses where our selected course is the "source" of an equivalency relationship. The second part finds courses where our selected course is the "target." Put them together, and you get all equivalent courses regardless of which direction the relationship was originally entered.

Handling Data Import

Real-world data is messy. CSV files have duplicates, missing information, and all sorts of inconsistencies. The app handles this gracefully with a smart import system.

The key is the `get_or_create` function, which tries to be helpful rather than just crashing when it encounters duplicates:

```
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]  # Found it! Use the existing one.

    # Doesn't exist yet, so let's try 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):
            # Oops, someone else created it between our check and our insert
            # Let's look for it again
            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  # Some other kind of error we don't know how to handle
```

This approach means that if your CSV file lists "University of California" fifty times, the app will create it once and then reuse that record for all the other references. Much better than crashing on the second occurrence!

Managing Transfer Plans

Creating a transfer plan involves generating a unique code that students can use to retrieve their plan later:

```
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 ensures these codes are cryptographically random - no one's going to guess someone else's plan code by accident.

When creating a plan, the backend does some basic validation to make sure all the required pieces are there:

```
@app.route('/api/create-plan', methods=['POST'])
def create_plan():
    try:
        data = request.get_json()
        # Make sure they gave us everything we need
        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 a unique code
        code = generate_plan_code()
        conn = get_db_connection()

        # Make sure it's actually unique (just in case)
        while conn.execute('SELECT id FROM TransferPlan WHERE code = ?',
(code,)).fetchone():
            code = generate_plan_code()
```

The while loop at the end handles the extremely unlikely case where we generate a duplicate code. Better safe than sorry!

The Frontend

The React frontend is where users actually interact with all this course data. It's designed to feel natural and responsive, like browsing through a well-organized course catalog.

State Management

The main App component manages a lot of different pieces of information:

```
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);
```

This might look like a lot, but each piece serves a specific purpose. The first group (`institutions`, `departments`, etc.) holds the course data. The second group tracks what the user has expanded or selected in the browsing interface. The third group manages the transfer plan they're building.

Smart Data Loading

Instead of loading everything at once (which would be slow and wasteful), the app loads data as users need it:

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

When someone expands an institution, we load its departments. When they collapse it, we clear out the departments, courses, and equivalencies to keep the interface clean. This pattern continues down the hierarchy:

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

Course Selection: Building a Transfer Plan

When users find courses they want to include in their transfer plan, they can add them to their selection. The logic is straightforward but important:

```
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 checks whether a course is already selected. If it is, clicking removes it. If it isn't, clicking adds it. The visual feedback in the interface makes this behavior clear to users.

Component Organization

Instead of cramming everything into one giant component, the app splits different features into focused components:

Navigation handles switching between different views and shows contextual information:

```
const Navigation = ({
  currentView,
  setCurrentView,
  isEditMode,
  loadedPlan,
  selectedCourses,
  searchCode,
  setSearchCode,
  loadPlan,
  cancelEdit
}) => {
```

BrowseView manages the hierarchical course browsing:

```
const BrowseView = ({
  institutions,
  departments,
  courses,
  equivalents,
  expandedInstitution,
  setExpandedInstitution,
  // ... lots more props
}) => {
```

Each component receives exactly the data and functions it needs through props. This makes the code easier to understand and maintain.

Edit Mode

One of the trickier parts of the app is allowing users to edit existing transfer plans. The challenge is making it clear what's happening without confusing the interface.

When someone starts editing a plan, the app switches to "edit mode" and populates the form with existing data:

```
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 interface shows clear indicators that editing is happening:

- Navigation buttons change their labels
- Special banners appear explaining what's being edited
- The course browse view shows how many courses are currently selected

If someone tries to cancel editing after making changes, the app checks for unsaved modifications and confirms their intention:

```
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 kind of attention to detail makes the difference between an app that's technically functional and one that's actually pleasant to use.

The Import Feature

Academic data doesn't come in perfect, clean formats. Course catalogs are maintained by different people using different systems, and equivalency information is often stored in spreadsheets that have evolved over years.

The CSV import feature acknowledges this reality. It doesn't just try to import data and crash when things go wrong. Instead, it processes each row individually, keeps track of what works and what doesn't, and gives detailed feedback:

```
# 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 the row ...
        rows_processed += 1

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

At the end, users get a comprehensive report of what happened, including specific error messages for rows that couldn't be processed. This makes it possible to fix data problems and try again, rather than just wondering why the import "didn't work."

Development and Deployment

The app uses modern but straightforward tools. Vite handles the frontend development experience - it's fast, reliable, and doesn't require a lot of configuration:

```
// 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,
      }
    }
  }
})
```

The proxy configuration means that during development, API requests automatically get forwarded to the Flask backend. No CORS headaches, no configuration complexity.

For the backend, Flask keeps things simple and predictable. The entire API fits in one Python file, uses standard SQLite (no database server to configure), and handles all the CRUD operations you'd expect.

Why This Architecture Works

This app succeeds because it matches the complexity of the solution to the complexity of the problem. Course equivalency tracking isn't rocket science, but it does involve a lot of relationships between different pieces of data. The relational database handles those relationships naturally. The hierarchical browsing interface matches how people think about course catalogs. The transfer plan feature solves a real workflow problem without adding unnecessary complexity.

Most importantly, the app acknowledges that data is messy and users make mistakes. It handles duplicates gracefully, provides clear feedback when things go wrong, and gives users ways to recover from errors. That's the difference between a technical demo and a tool people actually want to use.

The component-based frontend architecture means that each piece of functionality can be understood and maintained independently. Need to change how course browsing works? You can focus on just the BrowseView component. Want to add features to transfer plans? The plan management components are self-contained.

This isn't the most cutting-edge tech stack in the world, but it's reliable, understandable, and gets the job done without getting in the way. Sometimes that's exactly what you need.