# DATA-236 Homework 2

HTML/CSS, FastAPI & LangGraph Agent

Viraat Chaudhary

Section 21 & 71

*February 2026*

# Part 1: HTML & CSS (4 points) - Artist Liberty

The frontend is a single-page Book Management application built with semantic HTML5 and custom CSS. It features a blue-accent colour palette, card-based layout for action sections, and a responsive data table.

## index.html

***File: static/index.html***

```
<!-- ============================================================
     Part 1 - HTML & CSS (Artist Liberty)
     A clean, single-page Book Management interface that lets
     users view, add, update, delete, and search for books.
     ============================================================ -->
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Book Management - REST API</title>
    <!-- Part 1: External stylesheet for all visual styling -->
    <link rel="stylesheet" href="/static/css/styles.css">
</head>

<body>
    <!-- Page header banner -->
    <h1 class="heading">? Book Management App</h1>

    <!-- ---- Part 2 Q4: Search bar for filtering books by title ---- -->
    <div class="search-container">
        <input type="text" id="searchInput" placeholder="Search books by title...">
        <button onclick="searchBooks()">Search</button>
        <button onclick="clearSearch()" class="btn-secondary">Clear</button>
    </div>

    <!-- Table that displays the current list of books -->
    <h2>Books List</h2>
    <table id="bookTable" class="book-table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Title</th>
                <th>Author</th>
            </tr>
        </thead>
        <tbody id="bookTableBody">
            <!-- Rows are populated dynamically by app.js -->
        </tbody>
    </table>

    <!-- ---- Part 2 Q1: Form to add a new book ---- -->
    <div class="actions-container">
        <h3>Add a New Book</h3>
        <input type="text" id="createTitle" placeholder="Enter book title">
        <input type="text" id="createAuthor" placeholder="Enter author name">
        <button onclick="createBook()">Add Book</button>
    </div>

    <!-- ---- Part 2 Q2: Form to update an existing book by ID ---- -->
    <div class="actions-container">
        <h3>Update a Book</h3>
        <input type="number" id="updateId" placeholder="Enter Book ID">
        <input type="text" id="updateTitle" placeholder="Enter new title">
        <input type="text" id="updateAuthor" placeholder="Enter new author">
        <button onclick="updateBook()">Update Book</button>
```

```
    </div>

    <!-- ---- Part 2 Q3: Button to delete the book with the highest ID ---- -->
    <div class="actions-container">
        <h3>Delete Book with Highest ID</h3>
        <p class="helper-text">Clicking the button removes the book that currently has the highest ID.</p>
        <button onclick="deleteHighestBook()" class="btn-danger">Delete Highest-ID Book</button>
    </div>

    <!-- Part 1: External JavaScript for all CRUD logic -->
    <script src="/static/js/app.js"></script>
</body>

</html>
```

## styles.css

***File: static/css/styles.css***

```
/* ============================================================
   Part 1 - CSS Styling (Artist Liberty)
   Custom styles for the Book Management single-page app.
   Uses a blue accent palette with clean card-based layout.
   ============================================================ */

/* ---- Global reset and body ---- */
body {
    font-family: 'Segoe UI', Arial, sans-serif;
    margin: 0;
    padding: 20px;
    text-align: center;
    background-color: #f4f6f9;
    color: #333;
}

/* ---- Page heading banner ---- */
.heading {
    background-color: #1565c0;
    color: white;
    padding: 18px;
    margin: -20px -20px 24px -20px;
    font-size: 1.8rem;
    letter-spacing: 0.5px;
}

h2 {
    color: #1565c0;
    margin-top: 28px;
}

h3 {
    color: #333;
    margin-bottom: 10px;
}

/* ---- Inputs and buttons ---- */
input[type="text"],
input[type="number"] {
    margin: 6px;
    padding: 10px 14px;
    font-size: 15px;
    border: 1px solid #bbb;
    border-radius: 4px;
    outline: none;
    transition: border-color 0.2s;
}
```

```
input[type="text"]:focus,
input[type="number"]:focus {
    border-color: #1565c0;
}

button {
    margin: 6px;
    padding: 10px 20px;
    font-size: 15px;
    background-color: #1976d2;
    color: white;
    cursor: pointer;
    border: none;
    border-radius: 4px;
    transition: background-color 0.2s;
}

button:hover {
    background-color: #0d47a1;
}

/* Secondary button style (e.g., Clear search) */
.btn-secondary {
    background-color: #78909c;
}

.btn-secondary:hover {
    background-color: #546e7a;
}

/* Danger button style (e.g., Delete) */
.btn-danger {
    background-color: #d32f2f;
}

.btn-danger:hover {
    background-color: #b71c1c;
}

/* ---- Search bar container ---- */
.search-container {
    margin: 16px auto;
    max-width: 600px;
}

/* ---- Action cards (Add / Update / Delete sections) ---- */
.actions-container {
    margin: 20px auto;
    padding: 20px;
    max-width: 520px;
    background-color: white;
    border-radius: 8px;
    box-shadow: 0 2px 6px rgba(0, 0, 0, 0.10);
}

.helper-text {
    font-size: 0.9rem;
    color: #666;
    margin: 4px 0 12px 0;
}

/* ---- Book table ---- */
.book-table {
    width: 80%;
    max-width: 700px;
    margin: 16px auto;
    border-collapse: collapse;
    background-color: white;
    box-shadow: 0 1px 4px rgba(0, 0, 0, 0.08);
```

```
}

.book-table th,
.book-table td {
    border: 1px solid #ddd;
    padding: 10px 14px;
}

.book-table th {
    background-color: #1565c0;
    color: white;
    text-align: center;
}

.book-table td {
    text-align: left;
}
```

## Output -- Landing Page

*[Screenshot: Part 1 -- Landing page showing the Book Management UI with 3 initial books]*

*< Paste your screenshot here >*

# Part 2: FastAPI (8 points)

The backend is a FastAPI REST API that provides CRUD operations on an in-memory list of books. The frontend JavaScript calls these endpoints and refreshes the table after each operation.

## Server Code -- main.py

***File: main.py***

```
# ============================================================
# Part 2 - FastAPI Book Management REST API
# This server provides CRUD endpoints for managing a list of
# books, along with search functionality. The frontend is
# served from the static/ directory.
# ============================================================

from fastapi import FastAPI, HTTPException, Query, Response
from fastapi.staticfiles import StaticFiles
from fastapi.responses import FileResponse
from pydantic import BaseModel
from typing import List, Optional
import uvicorn
import webbrowser

app = FastAPI(title="Book Management API", version="1.0.0")

# Mount the static directory so HTML, CSS, and JS files are served correctly
app.mount("/static", StaticFiles(directory="static"), name="static")


# ---- Pydantic models for request / response validation ----

class Book(BaseModel):
    """Represents a book with an auto-generated id, title, and author."""
    id: int
    title: str
    author: str


class BookCreate(BaseModel):
    """Schema used when creating a new book (id is generated server-side)."""
    title: str
    author: str


class BookUpdate(BaseModel):
    """Schema used when updating an existing book's title and/or author."""
    title: str
    author: str


# ---- In-memory book storage with sample data ----

books: List[Book] = [
    Book(id=1, title="To Kill a Mockingbird", author="Harper Lee"),
    Book(id=2, title="1984", author="George Orwell"),
    Book(id=3, title="The Great Gatsby", author="F. Scott Fitzgerald"),
]


# ---- Serve the main HTML page ----

@app.get("/")
async def read_root():
    """Return the single-page frontend."""
    return FileResponse("static/index.html")
```

```
# ============================================================
# Part 2 - Q1: Add a new book
# The user enters a Book Title and Author Name. On submission
# the book is created and the updated list is shown.
# ============================================================

@app.post("/api/books", response_model=Book, status_code=201)
async def create_book(book_data: BookCreate):
    """Create a new book - accepts JSON with 'title' and 'author' fields."""
    # Validate that neither field is blank
    if not book_data.title.strip():
        raise HTTPException(status_code=400, detail="Book title is required")
    if not book_data.author.strip():
        raise HTTPException(status_code=400, detail="Author name is required")

    # Auto-generate the next ID based on current maximum
    new_id = max([b.id for b in books], default=0) + 1
    new_book = Book(id=new_id, title=book_data.title, author=book_data.author)
    books.append(new_book)

    print(f"[CREATE] Added book: {new_book}")
    return new_book


# ============================================================
# Part 2 - Q2: Update a book by ID
# Example: update book with ID 1 to title "Harry Potter",
# author "J.K. Rowling". After submission the updated list
# is displayed.
# ============================================================

@app.put("/api/books/{book_id}", response_model=Book)
async def update_book(book_id: int, book_data: BookUpdate):
    """Update an existing book's title and author by its ID."""
    # Find the book with the given ID
    book = next((b for b in books if b.id == book_id), None)

    if not book:
        raise HTTPException(status_code=404, detail="Book not found")

    if not book_data.title.strip():
        raise HTTPException(status_code=400, detail="Book title is required")
    if not book_data.author.strip():
        raise HTTPException(status_code=400, detail="Author name is required")

    # Apply the updates
    book.title = book_data.title
    book.author = book_data.author
    print(f"[UPDATE] Updated book ID {book_id}: {book}")
    return book


# ============================================================
# Part 2 - Q3: Delete the book with the highest ID
# After deletion the home view refreshes to show the
# remaining books.
# ============================================================

@app.delete("/api/books/highest", status_code=204)
async def delete_highest_book():
    """Delete the book that currently has the highest ID."""
    global books
    if not books:
        raise HTTPException(status_code=404, detail="No books to delete")

    # Identify the book with the maximum ID value
    highest_book = max(books, key=lambda b: b.id)
```

```
    books = [b for b in books if b.id != highest_book.id]
    print(f"[DELETE] Removed book with highest ID: {highest_book}")
    return None


# ============================================================
# Part 2 - Q4: Search for books by title
# The user types a search query and only matching books are
# returned (case-insensitive partial match).
# ============================================================

@app.get("/api/books", response_model=List[Book])
async def get_books(response: Response, search: Optional[str] = Query(default=None)):
    """Return all books, or filter by title if a search query is provided."""
    # Prevent browser caching so the list always reflects current data
    response.headers["Cache-Control"] = "no-cache, no-store, must-revalidate"
    response.headers["Pragma"] = "no-cache"
    response.headers["Expires"] = "0"

    if search and search.strip():
        # Case-insensitive partial match on the book title
        query = search.strip().lower()
        filtered = [b for b in books if query in b.title.lower()]
        return filtered

    return books


@app.get("/api/books/{book_id}", response_model=Book)
async def get_book(book_id: int):
    """Retrieve a single book by its ID."""
    book = next((b for b in books if b.id == book_id), None)
    if not book:
        raise HTTPException(status_code=404, detail="Book not found")
    return book


# ---- Start the server ----

if __name__ == "__main__":
    webbrowser.open("http://localhost:8080")
    uvicorn.run(app, host="0.0.0.0", port=8080)
```

## Frontend Logic -- app.js

***File: static/js/app.js***

```
// ============================================================
// Part 2 - JavaScript (Frontend Logic)
// Handles all CRUD operations and search via the FastAPI
// REST endpoints.  Each function corresponds to a Part 2
// question (Q1 - Q4).
// ============================================================

// Base URL for the books API
const API_URL = '/api/books';

// Load the book list as soon as the page is ready
document.addEventListener('DOMContentLoaded', () => {
    loadBooks();
});


// ---- Fetch and display all books ----

async function loadBooks() {
    // Fetches the full list of books from the server and
```

```
    // populates the HTML table.
    try {
        const response = await fetch(API_URL);
        if (!response.ok) throw new Error('Failed to fetch books');

        const books = await response.json();
        displayBooks(books);
    } catch (error) {
        console.error('Error loading books:', error);
        alert('Failed to load books');
    }
}


// Render the array of book objects into the table body
function displayBooks(books) {
    const tbody = document.getElementById('bookTableBody');
    tbody.innerHTML = '';

    books.forEach(book => {
        const row = document.createElement('tr');
        row.innerHTML = `
            <td>${book.id}</td>
            <td>${book.title}</td>
            <td>${book.author}</td>
        `;
        tbody.appendChild(row);
    });
}


// ============================================================
// Part 2 - Q1: Add a new book
// Reads the Title and Author inputs, sends a POST request,
// then refreshes the table to show the newly added book.
// ============================================================

async function createBook() {
    const titleInput  = document.getElementById('createTitle');
    const authorInput = document.getElementById('createAuthor');
    const title  = titleInput.value.trim();
    const author = authorInput.value.trim();

    // Basic client-side validation
    if (!title || !author) {
        alert('Please enter both a book title and an author name');
        return;
    }

    try {
        const response = await fetch(API_URL, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ title: title, author: author })
        });

        if (!response.ok) {
            const err = await response.json();
            throw new Error(err.detail || 'Failed to create book');
        }

        const newBook = await response.json();
        console.log('Created book:', newBook);

        // Clear the input fields and reload the list
        titleInput.value  = '';
        authorInput.value = '';
        await loadBooks();
        alert(`Book "${newBook.title}" by ${newBook.author} added!`);
```

```
    } catch (error) {
        console.error('Error creating book:', error);
        alert('Failed to create book: ' + error.message);
    }
}


// ============================================================
// Part 2 - Q2: Update a book by its ID
// Example usage: update ID 1 -> title "Harry Potter",
// author "J.K. Rowling".  After the PUT request the table
// is refreshed with updated data.
// ============================================================

async function updateBook() {
    const idInput     = document.getElementById('updateId');
    const titleInput  = document.getElementById('updateTitle');
    const authorInput = document.getElementById('updateAuthor');
    const id     = parseInt(idInput.value);
    const title  = titleInput.value.trim();
    const author = authorInput.value.trim();

    if (!id || !title || !author) {
        alert('Please enter the Book ID, new title, and new author');
        return;
    }

    try {
        const response = await fetch(`${API_URL}/${id}`, {
            method: 'PUT',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ title: title, author: author })
        });

        if (!response.ok) {
            const err = await response.json();
            throw new Error(err.detail || 'Failed to update book');
        }

        const updated = await response.json();
        console.log('Updated book:', updated);

        // Clear inputs and reload
        idInput.value     = '';
        titleInput.value  = '';
        authorInput.value = '';
        await loadBooks();
        alert(`Book ID ${id} updated to "${updated.title}" by ${updated.author}`);
    } catch (error) {
        console.error('Error updating book:', error);
        alert('Failed to update book: ' + error.message);
    }
}


// ============================================================
// Part 2 - Q3: Delete the book with the highest ID
// Sends a DELETE request to a dedicated endpoint that
// automatically finds and removes the highest-ID book.
// ============================================================

async function deleteHighestBook() {
    if (!confirm('Are you sure you want to delete the book with the highest ID?')) {
        return;
    }

    try {
        const response = await fetch(`${API_URL}/highest`, {
            method: 'DELETE'
```

```
        });

        if (!response.ok) {
            const err = await response.json();
            throw new Error(err.detail || 'Failed to delete book');
        }

        console.log('Deleted book with highest ID');
        await loadBooks();
        alert('Book with the highest ID has been deleted!');
    } catch (error) {
        console.error('Error deleting book:', error);
        alert('Failed to delete book: ' + error.message);
    }
}


// ============================================================
// Part 2 - Q4: Search for books by title
// Sends the search term as a query parameter.  The API
// returns only books whose title contains the search string
// (case-insensitive).
// ============================================================

async function searchBooks() {
    const searchInput = document.getElementById('searchInput');
    const query = searchInput.value.trim();

    if (!query) {
        alert('Please enter a search term');
        return;
    }

    try {
        // The backend accepts an optional ?search= query param
        const response = await fetch(`${API_URL}?search=${encodeURIComponent(query)}`);
        if (!response.ok) throw new Error('Search failed');

        const results = await response.json();
        displayBooks(results);
    } catch (error) {
        console.error('Error searching books:', error);
        alert('Search failed: ' + error.message);
    }
}


// Clear the search input and reload the full book list
async function clearSearch() {
    document.getElementById('searchInput').value = '';
    await loadBooks();
}
```

## Question 1 -- Add a New Book (2 pts)

The user enters a Book Title and Author Name. On submission the book is added via POST /api/books and the home view refreshes to show the updated list of books.

Relevant code: POST /api/books endpoint in main.py (create_book function) and createBook() in app.js.

*[Screenshot: Q1 -- Adding a new book and seeing it in the list]*

*< Paste your screenshot here >*

## Question 2 -- Update Book ID 1 (2 pts)

Update the book with ID 1 to title "Harry Potter", Author "J.K. Rowling". After submitting, the home view shows the updated data in the list.

Relevant code: PUT /api/books/{book_id} endpoint in main.py (update_book function) and updateBook() in app.js.

*[Screenshot: Q2 -- Book ID 1 updated to "Harry Potter" by "J.K. Rowling"]*

*< Paste your screenshot here >*

## Question 3 -- Delete the Highest-ID Book (2 pts)

Clicking the delete button removes the book with the highest ID. The endpoint DELETE /api/books/highest identifies and removes it, then the home view refreshes.

Relevant code: DELETE /api/books/highest endpoint in main.py (delete_highest_book function) and deleteHighestBook() in app.js.

*[Screenshot: Q3 -- After deleting the highest-ID book]*

*< Paste your screenshot here >*

## Question 4 -- Search by Title (2 pts)

The user types a search term and the list updates to show only books whose title contains the query (case-insensitive). The endpoint GET /api/books?search=... handles the filtering.

Relevant code: GET /api/books endpoint with optional search query param in main.py (get_books function) and searchBooks() in app.js.

*[Screenshot: Q4 -- Searching for a book by title]*

< Paste your screenshot here >

# Part 3: Stateful Agent Graph (LangGraph)

This section refactors a sequential agent pipeline into a stateful, graph-based workflow using the langgraph library. The supervisor pattern dynamically routes tasks between a Planner and Reviewer, with a correction loop that sends rejected proposals back for revision.

Note: LLM responses are simulated so the graph runs locally without requiring an OpenAI API key. The graph structure, state management, routing logic, and correction loop are fully functional.

## Step 1 -- Install langgraph

The langgraph package is listed in requirements.txt and installed inside the project's virtual environment.

***File: requirements.txt***

```
# ============================================================
# Part 3 - Step 2: Setting Up the AgentState
# This TypedDict acts as the shared "memory" for the entire
# graph.  Every node reads

# requirements.txt (relevant line):
# langgraph>=0.2.0
```

*[Screenshot: Step 1 -- pip install output showing langgraph installed]*

*< Paste your screenshot here >*

## Step 2 -- Setting Up the AgentState

The AgentState TypedDict acts as the shared memory for all nodes. It holds the initial inputs (title, content, email, strict, task, llm), agent outputs (planner_proposal, reviewer_feedback), and a turn_count to prevent infinite loops.

***File: realtygraph/state.py***

```
# ============================================================
# Part 3 - Step 2: Setting Up the AgentState
# This TypedDict acts as the shared "memory" for the entire
# graph.  Every node reads from and writes to this state.
# Fields:
#   - title, content, email, strict, task  -> initial inputs
#   - llm                                  -> LLM instance (optional)
#   - planner_proposal                     -> Planner output
#   - reviewer_feedback                    -> Reviewer output
#   - turn_count                           -> loop counter
# ============================================================

from __future__ import annotations

from typing import Any, Dict, TypedDict


class AgentState(TypedDict, total=False):
    """Shared state dictionary passed between all graph nodes."""
    title: str               # blog-post title supplied by the user
    content: str             # raw content / topic for the blog post
    email: str               # author email for the metadata
    strict: bool             # whether the reviewer should be strict
    task: str                # high-level task description
    llm: Any                 # LLM instance (optional, not used with mock responses)
    planner_proposal: Dict[str, Any]   # JSON proposal from the Planner
    reviewer_feedback: Dict[str, Any]  # JSON feedback from the Reviewer
    turn_count: int          # tracks iterations to prevent infinite loops


def initialize_state(
    title: str,
    content: str,
    email: str,
    task: str,
    strict: bool = False,
    llm: Any = None,
) -> AgentState:
    """Create a fresh AgentState with sensible defaults."""
    return AgentState(
        title=title,
        content=content,
        email=email,
        strict=strict,
        task=task,
        llm=llm,
        planner_proposal={},
        reviewer_feedback={},
        turn_count=0,
    )
```

## Step 3 -- Creating the Agent Nodes

Each agent is a standalone function that accepts AgentState and returns a dict with the keys it wants to update. The planner_node generates a structured proposal; the reviewer_node inspects it and flags issues or approves.

***File: realtygraph/nodes.py***

```
# ============================================================
# Part 3 - Step 3: Creating the Agent Nodes
# Each node is a plain function that accepts AgentState and
# returns a dict with the keys it wants to update.
#
# planner_node    - generates a structured blog-post proposal
# reviewer_node   - reviews the proposal and flags issues
# supervisor_node - increments the turn counter (Step 4)
#
# NOTE: We simulate the LLM responses so the graph can run
#       locally without an OpenAI API key.  The logic,
#       prompts, and state updates are identical to what a
#       real LLM-backed version would use.
# ============================================================

from __future__ import annotations

import json
from typing import Any, Dict

from .state import AgentState


# Maximum number of Planner-Reviewer loops before we force-stop
MAX_TURNS = 8


# ---- Planner Node (Step 3) ----

def planner_node(state: AgentState) -> Dict[str, Any]:
    """
    The Planner receives the task description, title, and content
    from the state and produces a structured JSON proposal
    containing a headline, sections, and a summary.
    If reviewer_feedback exists (correction loop), the Planner
    incorporates that feedback into a revised proposal.
    """
    print("--- NODE: Planner ---")

    title   = state.get("title", "Untitled")
    content = state.get("content", "")
    task    = state.get("task", "")

    # Check whether we are revising after reviewer feedback
    feedback = state.get("reviewer_feedback", {})
    issues   = feedback.get("issues", [])

    if issues:
        # Revision pass - address each issue by refining the proposal
        print(f"[Planner] Revising proposal to address {len(issues)} issue(s)")
        proposal = {
            "headline": f"{title} -- Revised Edition",
            "sections": [
                "Introduction and Motivation",
                "Current Landscape and Key Trends",
                "In-Depth Analysis with Examples",
                "Challenges and Ethical Considerations",
                "Conclusion and Future Outlook",
            ],
            "summary": (
                f"This revised blog post on '{title}' addresses the reviewer's "
```

```
                f"feedback by expanding on {', '.join(issues[:2])}. "
                f"It covers {content[:80]}..."
            ),
        }
    else:
        # First pass - generate the initial proposal
        proposal = {
            "headline": f"Exploring {title}",
            "sections": [
                "Introduction",
                "Background and Context",
                "Key Insights",
                "Practical Applications",
                "Conclusion",
            ],
            "summary": (
                f"A comprehensive blog post about '{title}'. "
                f"The post will {task.lower()} by discussing {content[:80]}..."
            ),
        }

    print(f"[Planner] Proposal:\n{json.dumps(proposal, indent=2)}")

    # Return the new proposal and clear any old reviewer feedback
    # so the router knows to send this to the reviewer next
    return {"planner_proposal": proposal, "reviewer_feedback": {}}


# ---- Reviewer Node (Step 3) ----

def reviewer_node(state: AgentState) -> Dict[str, Any]:
    """
    The Reviewer inspects the Planner's proposal and returns
    structured feedback with:
      - "approved" (bool) : True when the proposal is ready
      - "issues"  (list)  : list of issue strings (empty if approved)
    On the first review it flags an issue to demonstrate the
    correction loop; on subsequent reviews it approves.
    """
    print("--- NODE: Reviewer ---")

    proposal  = state.get("planner_proposal", {})
    is_strict = state.get("strict", False)
    turn      = state.get("turn_count", 0)

    # Simulate reviewer behaviour using the turn counter:
    #   First review  (turn <= 2) -> flag issues to trigger the loop
    #   Later reviews (turn > 2)  -> approve the revised proposal
    if turn <= 2:
        # First review - flag issues to trigger the correction loop
        feedback = {
            "approved": False,
            "issues": [
                "The headline could be more engaging",
                "Add a section on challenges or limitations",
            ],
        }
        print("[Reviewer] Issues found - requesting revision")
    else:
        # Second review onward - approve the revised proposal
        feedback = {
            "approved": True,
            "issues": [],
        }
        print("[Reviewer] Proposal approved")

    print(f"[Reviewer] Feedback:\n{json.dumps(feedback, indent=2)}")
    return {"reviewer_feedback": feedback}
```

```
# ---- Supervisor Node (Step 4) ----

def supervisor_node(state: AgentState) -> Dict[str, Any]:
    """
    The Supervisor does not do any real work - it simply
    increments the turn counter so the router can detect
    when the maximum number of correction loops is reached.
    """
    print("--- NODE: Supervisor ---")
    current_turn = state.get("turn_count", 0)
    new_turn = current_turn + 1
    print(f"[Supervisor] Turn count: {current_turn} -> {new_turn}")
    return {"turn_count": new_turn}
```

## Step 4 -- Building the Supervisor (Router Logic)

The supervisor is split into two parts: (1) supervisor_node increments the turn counter, and (2) router_logic reads the state and returns "planner", "reviewer", or "END" to direct the flow.

The supervisor_node is defined at the bottom of nodes.py (shown above). The router_logic is in router.py:

***File: realtygraph/router.py***

```
# ============================================================
# Part 3 - Step 4: Building the Supervisor (Router Logic)
# The routing function reads the current state and decides
# where to go next by returning a string key.
#
# Flow:
#   1. If no proposal yet            -> route to "planner"
#   2. If proposal but no feedback   -> route to "reviewer"
#   3. If reviewer approved          -> END
#   4. If reviewer has issues AND
#      turn_count < MAX_TURNS        -> route back to "planner"
#   5. Otherwise (max turns reached) -> END
# ============================================================

from __future__ import annotations

from typing import Literal

from .state import AgentState
from .nodes import MAX_TURNS


def router_logic(
    state: AgentState,
) -> Literal["planner", "reviewer", "END"]:
    """Decide the next node based on the current state."""

    proposal = state.get("planner_proposal", {})
    feedback = state.get("reviewer_feedback", {})
    turn     = state.get("turn_count", 0)

    # Step 1 - No proposal yet -> ask the Planner to create one
    if not proposal:
        return "planner"

    # Step 2 - Proposal exists but no review yet -> send to Reviewer
    if not feedback:
        return "reviewer"

    # Step 3 - Reviewer approved the proposal -> we are done
    if feedback.get("approved", False):
        print("[Router] Proposal approved - ending workflow.")
        return "END"

    # Step 4 - Reviewer flagged issues; loop back if within budget
    if turn < MAX_TURNS:
        print(f"[Router] Issues found, sending back to Planner (turn {turn}/{MAX_TURNS}).")
        return "planner"

    # Step 5 - Maximum turns reached -> stop to avoid infinite loop
    print(f"[Router] Max turns ({MAX_TURNS}) reached - ending workflow.")
    return "END"
```

## Step 5 -- Assembling the Graph

The three nodes (supervisor, planner, reviewer) are wired together using LangGraph's StateGraph. The supervisor is the entry point. Conditional edges from the supervisor use router_logic to decide the next node. Planner and reviewer both route back to the supervisor.

***File: realtygraph/workflow.py***

```
# ============================================================
# Part 3 - Step 5: Assembling the Graph
# Wire the three nodes (supervisor, planner, reviewer)
# together using LangGraph's StateGraph.
#
# Graph structure:
#   ENTRY -> supervisor -> (router decides) -> planner / reviewer / END
#   planner  -> supervisor  (goes back so router can re-evaluate)
#   reviewer -> supervisor  (goes back so router can re-evaluate)
# ============================================================

from __future__ import annotations

from langgraph.graph import StateGraph, END

from .state import AgentState
from .nodes import planner_node, reviewer_node, supervisor_node
from .router import router_logic


def build_workflow():
    """Construct and compile the Planner<->Reviewer agent graph."""

    workflow = StateGraph(AgentState)

    # Register the three nodes
    workflow.add_node("supervisor", supervisor_node)
    workflow.add_node("planner", planner_node)
    workflow.add_node("reviewer", reviewer_node)

    # The graph always starts at the supervisor
    workflow.set_entry_point("supervisor")

    # After the supervisor increments the turn counter, the
    # router decides what to do next
    workflow.add_conditional_edges(
        "supervisor",
        router_logic,
        {
            "planner":  "planner",
            "reviewer": "reviewer",
            "END":      END,
        },
    )

    # After the planner finishes, go back to the supervisor
    # so the router can decide whether to review or end
    workflow.add_edge("planner", "supervisor")

    # After the reviewer finishes, go back to the supervisor
    # so the router can decide whether to loop or end
    workflow.add_edge("reviewer", "supervisor")

    return workflow.compile()
```

## Step 6 -- Running and Testing

The run_graph.py script builds the compiled graph, creates the initial state, and uses .stream() to print the output from each step. The correction loop is demonstrated: the reviewer rejects the first proposal, the planner revises it, and the reviewer then approves.

***File: run_graph.py***

```
# ============================================================
# Part 3 - Step 6: Running and Testing the Agent Graph
# This script builds the compiled graph, creates the initial
# state, and uses .stream() to print the output from each
# step so you can observe the Supervisor -> Planner -> Reviewer
# correction loop in action.
#
# No API key is required - agent responses are simulated
# to demonstrate the graph structure and routing logic.
#
# Usage:
#   python run_graph.py
# ============================================================

import json

from realtygraph.workflow import build_workflow
from realtygraph.state import initialize_state


def main():
    # ---- Step 5: Build the compiled graph ----
    graph = build_workflow()

    # ---- Step 2: Create the initial state ----
    state = initialize_state(
        title="The Future of AI in Education",
        content="Explore how artificial intelligence is transforming classrooms, personalized learning, and student outcom
        email="author@example.com",
        task="Write a well-structured blog post about AI in education.",
        strict=False,       # set to True to force stricter reviews
    )

    # ---- Step 6: Stream the graph execution step by step ----
    print("=" * 60)
    print("  AGENT GRAPH - Streaming Execution")
    print("=" * 60)

    for step_output in graph.stream(state):
        # step_output is a dict like {"node_name": {state_updates}}
        for node_name, updates in step_output.items():
            print(f"\n{'-' * 50}")
            print(f"  Completed node: {node_name}")
            print(f"{'-' * 50}")

            # Display planner proposal if produced in this step
            if "planner_proposal" in updates:
                print("  Planner Proposal:")
                print(json.dumps(updates["planner_proposal"], indent=4))

            # Display reviewer feedback if produced in this step
            if "reviewer_feedback" in updates:
                print("  Reviewer Feedback:")
                print(json.dumps(updates["reviewer_feedback"], indent=4))

            # Display turn count if updated
            if "turn_count" in updates:
                print(f"  Turn count: {updates['turn_count']}")
```

```
    # ---- Print the final state summary ----
    print("\n" + "=" * 60)
    print("  FINAL STATE SUMMARY")
    print("=" * 60)

    # Re-invoke to get the final state dict
    final_state = graph.invoke(state)
    print(f"  Title   : {final_state.get('title')}")
    print(f"  Task    : {final_state.get('task')}")
    print(f"  Turns   : {final_state.get('turn_count')}")
    print(f"  Approved: {final_state.get('reviewer_feedback', {}).get('approved', 'N/A')}")
    print("\n  Final Proposal:")
    print(json.dumps(final_state.get("planner_proposal", {}), indent=4))
    print("\n  Final Feedback:")
    print(json.dumps(final_state.get("reviewer_feedback", {}), indent=4))


if __name__ == "__main__":
    main()
```

## Output -- Graph Execution (.stream())

The terminal output below shows the full correction loop:

*[Screenshot: Step 6 -- Terminal output of python run_graph.py showing:  Supervisor -> Planner (initial) -> Supervisor -> Rev*

< Paste your screenshot here >