Sprint 1 Artifacts

1.1 Web Page is Displayed to the User

Description:
Create and serve a frontend page that loads successfully when the application runs.

User Story:
As a user, I want to open the application and see a homepage so that I know the app is running and accessible.

Acceptance Criteria:

- The homepage loads at `http://localhost:5173`.
- The page displays basic content.
- No frontend errors appear in the browser console.

Tasks:

- Initialize React project using Vite.
- Create `App.jsx` with placeholder UI.
- Configure routing -- Pushed to the Backlog for Sprint 2.
- Run the app locally.

Evidence of Completion:

- Screenshot of homepage rendering.
- Commit: `Bare bones scheduler vite react app`.

Testing / Verification:

- Verified in the browser that the page renders.
- Console inspected for no runtime errors.

Outcome: Successfully implemented — the React frontend builds and runs locally with an initial homepage.

1.2 Backend Responds to HTTP Requests

Description:
Set up a backend server to respond to basic HTTP requests (GET `/`).

User Story:
As a developer, I want a backend endpoint that confirms the API is running so that I can test connectivity from the frontend.

Acceptance Criteria:

- A GET request to `/api/hello` returns 200 OK and a response.
- The server starts successfully with no errors.

Tasks:

- Initialize Django app.
- Define a route `/api/hello` returning a sample message.
- Run the backend server and test using the browser.

Evidence of Completion:

- Screenshot of browser navigating to `http://127.0.0.1:8000/api/hello` → `"Hello World!"`.
- Commit: `Initial handling of course data using the Django REST Framework`. -- earlier commit did this at the path “courses/hello”

Testing / Verification:

- Verified via browser fetch request.

Outcome: Backend is operational and returns successful responses.

2.1 Frontend Displays Data from the Backend

Description:
Connect the frontend to the backend and display API data dynamically.

User Story:
As a user, I want to see backend-provided data displayed on the web page so that I know the frontend and backend are connected.

Acceptance Criteria:

- Fetch data from backend endpoint on page load.
- Data renders visibly (e.g., course list).
- Handles error state -- will be more thoroughly handled in later sprints.

Tasks:

- Use the Django REST Framework to call backend API.
- Display JSON response in a React component.
- Add basic UI formatting.

Evidence of Completion:

- Screenshot of the web page displaying fetched data.
- First Frontend Commit: `Pull data from backend and update data format`.
- Backend Commit: `Update Course model to reflect exportable data from the registrar`.

Testing / Verification:

- Verified via browser Network tab — successful API call and correct render.

Outcome: Frontend successfully displays data retrieved from the backend.

2.2 Backend Returns Domain Specific Data

Description:
Modify backend endpoint to return structured course-related data.

User Story:
As a user, I want the API to return course information so that I can view available courses on the frontend.

Acceptance Criteria:

- API returns a JSON array of courses with expected fields (`course_number`, `title`, `instructor`, etc.).
- Returns status code 200.

Tasks:

- Define `/api/courses` endpoint.
- Implement serializer/schema for course data.
- Test response structure via the browser.

Evidence of Completion:

- Screenshot of `/api/courses` JSON output.
- Commit: `Update Course model to reflect exportable data from the registrar`.

Testing / Verification:

- Tested endpoint manually in the browser.
- Confirmed expected keys and values returned.

Outcome: Backend now serves domain-specific data for frontend use.

2.3 Define Course Data Model

Description:
Create a database model representing course entities.

User Story:
As a developer, I need a Course model so that course data can be stored, retrieved, and modified efficiently.

Acceptance Criteria:

- The `Course` model should include all of the relevant fields present in the excel file exported from the registrar website with the addition of `school`, `department`, and `building`.
- Database migration runs successfully.

Tasks:

- Create model in Django ORM schema.
- Run migrations.
- Validate schema in browser.

Evidence of Completion:

- Screenshot of Django model page.
- Commit: `Update Course model to reflect exportable data from the registrar`.

Testing / Verification:

- Verified that `Course` table appears in the database.
- Confirmed CRUD operations work.

Outcome: Course model defined and integrated with backend.

2.5 Populate DB with Initial Data (Importing)

Description:
Import and seed the database with sample or real course data.

User Story:
As a user, I want to see preloaded course data so that I can interact with the app immediately without manually adding data.

Acceptance Criteria:

- The program is able to read the CSV files from the registrar and add the courses to the database.
- The database contains multiple course entries.
- `/api/courses` returns real records.

Tasks:

- Create import script (e.g., CSV/JSON importer).
- Run script to insert data into DB.
- Verify records exist.

Evidence of Completion:

- Screenshot of Django page with populated data.
- Commit: `added import_courses.py`.

Testing / Verification:

- Verified `/api/courses` returns populated data.
- Checked data count matches expected import.

Outcome: Database seeded with initial course data successfully.

Source: Initial generations by ChatGPT | Edited by Connor A.