CMPS 3830 Spring 2026

Phase 2 - Auth, Frontend, Deployment

Have this done by Feb 25th, 11:59PM

Phase 2 of CMPS 3830 this semester builds on the “finish” of phase 1 and covers authentication, authorization, react web, react expo, and deployments

Important Information

- All code you and your teammates must be pushed to Github in order for it to be considered for participation
- You will be given a repository based on Phase 1
- The repository will contain automated tests you must pass
- The automated tests are a necessary but not a sufficient requirement for completing the phase
- You are not to change the automated test code in this phase

Initial Setup

You’ll need to follow these steps before you can start. Please do this **ASAP** so we can resolve any issues you may have.

1) Go to the classroom link in GitHub here: **https://classroom.github.com/a/kifbCiT5**
2) Join or create your group - **please make sure you coordinate with your teammates so that you don’t try to create the group at the same time**
    a) **Make sure you enter your team as “g##” where ## is whatever team number you are assigned to (check canvas) - or select the team with your number if it was already created**
    b) Full Guide: INFO - Spring 2026 GitHub Classroom
3) Pull and test run your repository on your computer - make sure it builds and runs

GitHub, Branches, Tests

The repository that you’ll be using in this class, including this phase, is configured such that you must utilize a branch to push code to GitHub. This restriction is commonly used to ensure that all tests pass and code is “good enough” to be consumed by the rest of the team in general. Thus, you need to utilize pull requests to get code into the master branch on GitHub(See: GITHUB - Creating a pull request).

Email cmps3830@gmail.com

In other words: the master branch should always work.

But, because the starting repository has many tests that you are expected to pass, you are essentially being given a repository that is broken. Specifically: A GitHub Action ensures that the tests are run before a merge to master can finish, but that means the tests must pass before any code can be merged. A sort of catch-22

Since this phase may involve a fair amount of code to be done before you are passing all the tests, you may want to create and merge work into a “dev” branch that everyone on your team can share - directly merging into that “dev” branch rather than the “master” branch. This bypasses the requirements for all tests to pass.

Just remember: eventually you’ll need to merge everything into the master branch - so “dev” shouldn’t live too long.

Overall Requirements

- Must be built on the provided starter project that we will provide
- Must add migrations - **you may not delete the existing migration**
- All commits must be pushed
- Do not modify the automated tests
- Be sure that all inputs and outputs of all types are strongly typed and have the correct HTTP status codes
    - E.g. 200, 201, 400, 404, etc
- Be sure that you use the correct HTTP verbs: POST, PUT, GET, DELETE
    - See: https://developer.mozilla.org/en-us/docs/Web/HTTP/Methods
- The tests will be picky about what URLs those actions are on - follow the standards here: https://restfulapi.net/resource-naming/
- The API endpoints, DTOs, and Domain Entities below should be implemented
- The API + Database should be set up and deployed to azure automatically via GitHub Actions
- Never send or receive entity / model types (e.g. “Location”) on your controller endpoints - always send and receive *Dto types
- Configure your project to use asp.net core identity with cookie authentication
- Your code should migrate and seed your database on startup so there is data for the tests to retrieve
    - You need 3 locations
    - You need 3 users - all with the password: Password123!
        - bob - a user
        - sue - a user
        - galkadi - an admin

Recommended Process

To help you get started, we'd recommend working roughly in this order (some steps can be done in parallel - hence you being in a team):

- Setup asp.net core identity with cookie authentication - likely best done before API development - since the tests need authentication/authorization to work
    - Tutorial: INFO - Spring 2026 Setup ASP.Net Identity Core
- Local API Development - loop over this process until you have all the endpoints
    - Pick an API endpoint you want to implement (E.g. GET /api/locations)
        i) Generally: List/Create/Update/Delete is a good order
    - Create any needed DTOs (if missing) for that endpoint (e.g. LocationDto)
    - Create the Controller (if missing) for that API endpoint
    - Create any needed Domain Entities (if missing) for that API endpoint (e.g. Location)
    - Add a migration if required (e.g. you had to create one or more Domain Entities)
    - Add seeded data if required for that endpoint
    - Create the Action for that endpoint
        i) Use DbContext to get / map the DTOs as needed for the endpoint
    - Run a test using swagger
    - Run the automated tests and see if that endpoint has any failures
- **While Local Development is occurring above** - someone can setup the infrastructure required
    - Setup Azure App Service
        i) Tutorial: INFO - Spring 2026 Setup Azure App Service
        ii) Quick Link: https://383-bot.azurewebsites.net/api/set-email
    - Setup Azure SQL Database
        i) Tutorial: INFO - Spring 2026 Setup Azure SQL Database
    - Setup GitHub secrets
        i) Tutorial: INFO - Spring 2026 Deployment Secret
    - Setup GitHub Action To Deploy ASP.NET Core Automatically
        i) Tutorial: INFO - Spring 2026 GitHub Action ASPNET Deployment
- **While Local Development is occurring above** - someone can setup the react web integration
    - Tutorial: INFO - Spring 2026 Setup React Web
- **While Local Development is occurring above** - someone can setup the react expo integration
    - Tutorial: INFO - Spring 2026 Setup React Expo
- Create a PR, merge your code, and make sure the deployment works
    - Troubleshooting: INFO - Spring 2026 Troubleshooting Issues
- **Go read:**
    - **Quick Start • React (reactjs.org)**

DTOs

These are the expected DTOs

- LocationDto
  - Id : int
  - Name : string
  - Address : string
  - TableCount: int
  - ManagerId : int? (note: nullable)
- LoginDto
  - UserName : string
  - Password: string
- UserDto
  - Id : int
  - UserName : string
  - Roles : string[]
- CreateUserDto
  - UserName : string
  - Roles : string[]
  - Password: string

API Endpoints

These are the endpoints expected

- Locations
  - List All
    - Return a list of all locations
    - E.g. GET /api/locations
    - Returns HTTP 200
  - Get by id
    - Find a location by its unique Id and return the details
    - E.g. GET /api/locations/1
    - Returns HTTP 200/404
  - Create
    - Name must be provided
    - Name cannot be longer than 120 characters
    - Must have an address
    - Must have at least 1 table
    - ***new*** Must have a valid ManagerId (can be null if there is no manager)
    - Return the created Dto (hint: and its location)

        - E.g. POST /api/locations
        - Returns HTTP 201/400/401/403
        - ***New*** Admins may always use this
    - Update
        - Name must be provided
        - Name cannot be longer than 120 characters
        - Must have an address
        - Must have at least 1 table
        - ***new*** Must have a valid ManagerId (can be null if there is no manager - only admins can change the value of the ManagerId)
        - Return the updated Dto
        - E.g. PUT /api/locations/{id}
        - Returns HTTP 200/404/400/403
        - ***New*** Admins may always use this
        - ***New*** The users may call this only if they are the manager (i.e. their user id is the ManagerId)
    - Delete
        - Delete a location based on matching the Id
        - Do not delete anything if there is not a matching Id
        - E.g. DELETE /api/locations/{id}
        - Returns HTTP 200/404/403
        - ***New*** Admins may always use this
- Authentication ***new***
    - Login
        - Bad request if invalid password / username
        - Note: You will need to use “async / await”: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/
        - Accepts LoginDto
        - Returns UserDto
        - E.g. POST /api/authentication/login
        - Returns HTTP 200/400
    - Me
        - Gets the details of the currently logged in user
        - Returns UserDto
        - E.g. GET /api/authentication/me
        - Returns HTTP 200/401
    - Logout
        - Only authenticated users can use this
        - This logs the user out
        - Returns no data
        - This is a POST
        - E.g. POST /api/authentication/logout

        - Returns HTTP 200/401
- Users ***new***
    - Create
        - Only admins can do this
        - Only allow roles that exist to be sent
        - Must provide at least one role
        - Only allow unique usernames
        - Accepts CreateUserDto
        - Returns UserDto
        - E.g. POST /api/users
        - Returns HTTP 200/400/401/403

Domain Entities

These are the tables expected in SQL / Entity Framework Core “Entity” types

- Location - where people go to watch a movie
    - Id : int
    - Name : string - required, max length 120 characters
    - Address : string - required
    - TableCount: int - required, minimum 1
    - **Manager : User** (optional, can be null)
- User - inherits from IdentityUser<int>
    - Roles : ICollection<UserRole>
- UserRole - inherits from IdentityUserRole<int>
    - Role : Role
    - User : User
- Role - inherits from IdentityRole<int>
    - Users : ICollection<UserRole>