

This milestone is to be completed individually. No group work is allowed.

This milestone is intended to familiarize you with using sessions to maintain state in your web application.

For this milestone, you will continue working on your Node.js based application from milestone 2. The milestone builds upon the previous milestone's MVC architecture.

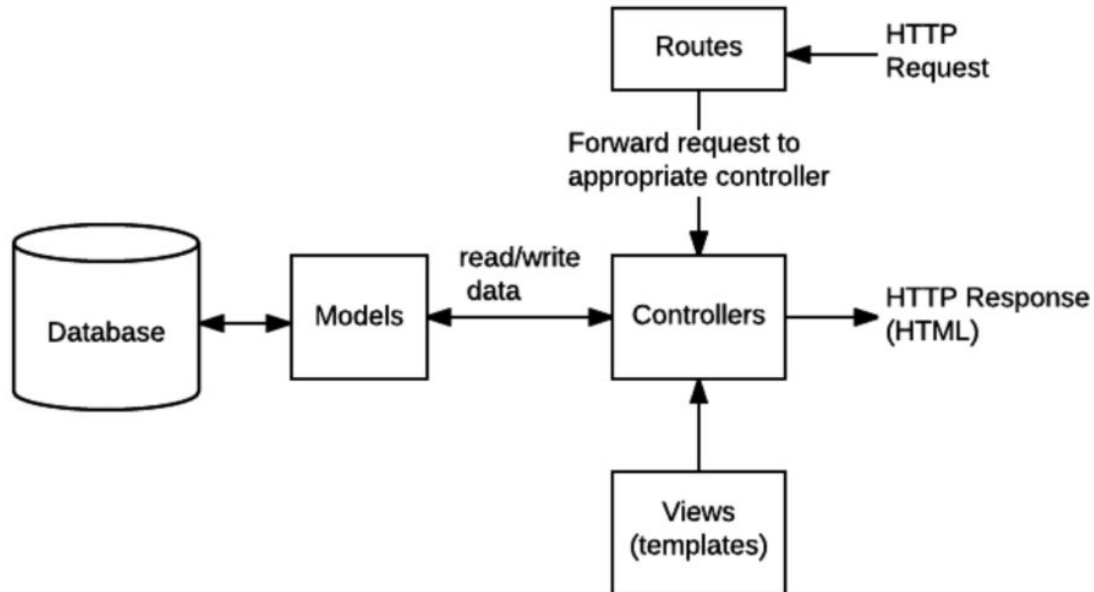
Milestone Specification:

In this milestone, you will add new features to your previous Node.js/Express web application, according to the following specifications:

1. Correct any errors identified or partial/missing functionality from the previous milestone.

If your application developed in milestone 2 has errors and is not complete, it is your responsibility to communicate and coordinate with course staff to help fix those bugs. Keep in mind that you are building in this application in stages. It is important that you have a working application for each stage.

2. All structure, design, and content requirements from previous milestones are mandatory unless explicitly updated in this milestone description.
3. Use JavaScript to implement the business layer of the application (**model**).
4. Use EJS pages to present the **view** to the browser.
5. Use routes to **control** the flow of the application.
6. Functionality that does not follow the milestone specifications will not receive credit.



Milestone Description:

The main task of this milestone is to implement the logic to support user-specific functionality. The application should utilize sessions to handle any updates/actions users take while interacting with the application.

The goal is to implement logic to support the following functionalities:

- user responding (RSVP) to a connection to save to their account (profile)
- user removing (delete) connections they've saved in their account (profile)
- user changing (update) their response. A user should be able to change any response they previously provided.

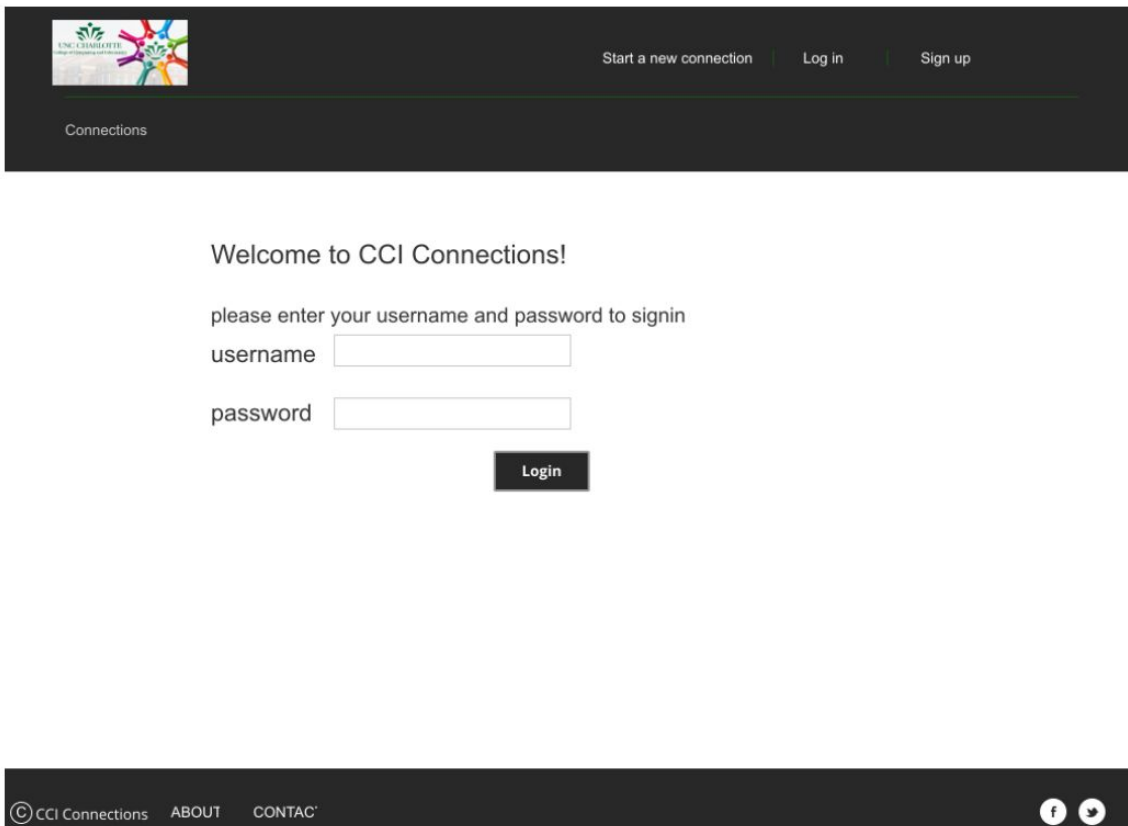
Milestone Tasks and Design

- define and create the new objects needed to represent the business models
- define and create the routes for the business logic (controller) to support each functionality added
- update EJS template views to utilize dynamic content available from the new data models
- update form / button actions and links in the site with the new routes defined

1. Add New EJS Site Pages for Login

Create a new EJS page for the following screen. While it is not necessary to do so, it may be useful to create them first as static design prototype pages (with placeholder data), in order to get the structuring needed, and then convert them to dynamic pages.

Filename: login.ejs – this page is accessible from the User Controller and linked into the application flow from the “Log In/Sign In” links in the user navigation bar. It should provide submission of username (email) and password for a user to login to the application and view their profile of saved connections. The “Log In/Sign In” button should be linked to the **UserController**.



The image shows a login page design for 'CCI Connections'. At the top, there is a dark navigation bar. On the left is the 'CCI CHARLOTTE' logo. On the right are links for 'Start a new connection', 'Log in', and 'Sign up'. Below the navigation bar is a section titled 'Connections'. The main content area has a heading 'Welcome to CCI Connections!' followed by the instruction 'please enter your username and password to sign in'. There are two input fields: 'username' and 'password'. Below the password field is a 'Login' button. At the bottom, there is a dark footer bar containing the text '© CCI Connections ABOUT CONTACT' and social media icons for Facebook and Twitter.

2. Create New Objects for Business Objects (Model)

Create Javascript objects for the following Model elements, with the specified properties and additional functions/methods.

User - represents a user of the application with the following properties:

- User ID
- First Name
- Last Name
- Email Address
- Optional fields
 - Address 1 Field
 - Address 2 Field
 - City
 - State
 - Zip Code
 - Country
- Any other fields you find necessary

UserConnection – represents a connection object saved to the user profile (associates a connection to a user profile) with the following properties:

- Connection
- rsvp
- Any other fields you find necessary (optional)

3. Create a utility class/module to manage a user's profile

UserProfile - represents the user profile. Contains the list of user connections as well as functions to support getting, adding and removing connections from the user profile. In other words, it's a class/module with methods/functions to store and manage the user connections in the user profile.

- properties:
 - User ID or User to associate this UserProfile object to the user
 - a list containing UserConnection objects for this user
- methods/functions
 - addConnection – adds a UserConnection for this connection / rsvp to the user profile. The profile should not allow multiple UserConnections for the same connection, but should update appropriately if one already exists.
 - removeConnection– removes the UserConnection associated with the given connection.
 - updateRSVP- updates an RSVP property for a specified UserConnection
 - getUserConnections – returns a List / Collection of UserConnection from the user profile
- Any other methods/functions you find necessary

4. Create Routes for Business Logic (Controller)

For this milestone, you will implement dynamic user profile functionality. You must use sessions to store data. This means that the user profile will be able to hold multiple connections at a time, and will maintain its content even if you navigate away from the user profile page (savedConnections view). We will be referring to the page with a user's saved connections as the **profile** view (think of this view as the user's dashboard. It displays a summary view of the connections they saved/shared and provides them ways to manage their profile).

Implement the following controller logic to operationalize the business logic. You will need to send parameters as part of the GET or POST http requests from button / form submissions as the context information that tells the controllers how to proceed.

UserController

(This can be split into multiple controllers, **but you must specify in your Milestone3_Info.pdf how you have divided the tasks or risk losing points**).

This controller is responsible for answering requests that pertain to user-specific functionality. These are the requests for accessing a service the application provides to support its users. These are registered users (we are simulating that with are hardcoded data) and these services/functionalities are specific and available to them. An application-user should be able to navigate to a view that lists connections they saved in the application as well as navigate to a view to provide connection RSVP.

Functionality:

- Login a user (add a user to the session without email/password verification) by initializing a UserProfile model and storing it in the session.
- This controller should continue to store a user's current profile contents in the session as long as the session is not destroyed
- Save a user's rsvp for a connection - add a new connection to the user profile
- Update a user's rsvp for a connection - update the value for the RSVP for a specific user connection already in the profile
- Delete a user's rsvp for a connection - delete a specific user connection already in the profile
- Display a user's list of saved connections - list all user connections on the savedConnections view
- Logout a user (remove a user from the session)

Notes:

- This controller is responsible for handling the following requests:

- for a request to login, this controller initializes a UserProfile, saves it to the session (start session tracking) and responds by displaying the profile view. When a session is first started the user profile will be empty.
- for a request to rsvp to a connection this controller responds by adding the requested connection to the user profile if it is not already in the profile or updating it if it already exists and sends the updated data to the profile view for display. (this would be the Yes, No, Maybe buttons action)
- for a request to update a single connection from the user profile this controller responds by retrieving the requested connection model and sending that data to the connection view for display. (this would be the update button link action)
- for a request to delete a single connection this controller responds by deleting the requested connection model from the user profile and sends the updated data to the profile view for display. (this would be the delete button link action)
- for requests to sign-out the user, this controller responds by removing the user information saved in the session and displaying the main page of the application (index.ejs).
- since this controller is responsible for functionality that is specific to a user it has to make sure that a user is logged in to the application. This is established through the session object. Once a user completes the login form we are bypassing an actual login functionality here and assuming this is a new user and a session attribute is used to track this user. Any time a request comes into this controller it has to check and verify this session attribute before attempting to fulfill the request.
- updates to the user profile will only be reflected in the session. As soon as the user signs out their profile is deleted as the session data is destroyed.

Bonus (10 points):

This requirement is a bonus. To receive credit you must include a section in your documentation (Milestone3_Info.pdf) detailing your design and implementation (how you fulfilled this requirement). Your documentation has to be clear and convey a good understanding of the design.

- as mentioned in milestone 2 you should keep in mind that users can interact (send requests and receive responses) from any application on the world wide web without necessarily using the webpages that are tied to the application. A user can simply send a request to a web-based application using command-line for

example. Do not rely on the user only using the links that are implemented in the application views. One way this controller can verify requests coming are from the views provided is by checking the requests for specific parameters (data) that would indicate to the controller the view that was responsible for sending a request. One way this can be done by including hidden fields in the view that the controller checks and would validate the authenticity of the requested action. Since this controller is responsible for managing connections it should keep track of the connections that are displayed on any view and the connections that belong to the user. So for any connection that is displayed on the view, you can use a hidden field named "viewConnections" with the connection code as value (which should be unique). The controller can then check for this parameter to get a list of all connections displayed on the view. For any action that is requested for a specific connection, it can then check that this connection is also in the connection list. If it is not then the controller should have a mechanism for handling such cases (for example, this can be as simple as redisplaying the all connections view).

Visit this page for [Sample Pseudocode for this controller.](#)

5. Update EJS Template Views to Include Dynamic Content From Data Models

Each of the principal views for connections will now receive some kind of dynamic data. Update the EJS views to replace all of the placeholder information with the dynamic data.

Also, if the user has been added to the session as part of a sign in, we consider them to have logged in to their account, and so all the page headers should now display the user name when they're signed in.

6. Update Form / Button Actions and Links in the Site to Dispatch Correctly

Each of the places in the site where a user can take an action that uses the dynamic data from this milestone should be updated to make the appropriate GET or POST request with the necessary parameters or form data.

Remember that requests for resources such as links should be using GET method and any requests that sends data such as form submissions must use POST. Updates must be made in all appropriate places where action would reasonably be indicated from the first milestone prototypes. For example: the links to a connection in saved connections

view will need to be dynamic to load the correct connection details and the user's current RSVP status.

Note: since we don't have user data you can hard-code the user object or use the username that is entered in the login form to represent a user in the session.

Milestone Submissions

What to submit using Canvas (Email submissions will NOT be accepted):

1. **Milestone3.zip** - An archive of the entire web application (project) stored in a standard ZIP File, you must ensure that all the files are included as part of the archive.
2. **Milestone3Info.pdf** – PDF document with the following milestone information:
 1. Screenshots of all views of your application displayed in your browser.
 2. Explanation of additional features, if any.
 3. Explanation of status, stopping point, and issues if incomplete.
 4. Discuss the easy and challenging parts of the milestone. How did you overcome all or some of the challenges?
 5. Optional Bonus - If you completed the bonus requirement detail your design and document your implementation.
 - 6.