

CS4530, Spring 2022

Assignments / Assignment 2: Conversation Areas

Assignment 2: Conversation Areas DUE FRIDAY FEBRUARY 11, 10:00PM EST

Change Log

- 1/28/22: Initial Release
- 1/29/22: Clarify expected return type of `conversationAreas` route
- 1/31/22: Clarify route specification, clarify behavior of `updatePlayerLocation` (2.1), update handout test for 2.1 to clearly specify the intended location to use is `userLocation.conversationLabel` (diff)
- 2/4/22: Clarify task 2.3 introductory language (diff)
- 2/5/22: Clarify conversation -> conversation area; rework 2.3 text to be clearer (diff)
- 2/6/22: Add note clarifying that Task 1.1 tests can not run on GradeScope until after Task 1.3 is completed; this does not impact Task 1.2

Welcome aboard to the Covey.Town team! We're glad that you're here and ready to join our development team as a new software engineer. We're building an open source virtual meeting application, and are very happy to see that we have so many new developers who can help make this application a reality. Covey.Town is a collective effort of many contributors, and by the end of the semester, you will have the option to submit a pull request to merge a new feature into the codebase, and become a direct contributor yourself.

We understand that some of you may have some web development experience, but don't expect that most of you do, and hence, have created this series of three individual assignments to help you get up to speed with our existing codebase and development environment.

The new feature: Conversation Areas

In the original release of Covey.Town (demoed at <https://spring2021.covey.town>), users connect to a "Town", which provides a 2D arcade-style map that users can walk around in. When two users get close, they are able to see and hear each other through a video call. Since that original release, our lead software engineer, Avery, has been quite busy on making this app more interesting and usable. One problem that Avery observed with the app was that users would focus on exploring the relatively small map, and then not know what to do: how do you have a conversation with someone else when there is nothing to indicate that you want to talk? If two

people are talking, how does a third person know that it is OK to approach them and join the conversation?

Avery has developed an exciting new feature concept for Covey.Town this semester:

Conversation Areas. Avery replaced the “outdoor” map from the original Covey.Town implementation with a new map for users to explore, which has some interesting sights to see and talk about, added text chat, and designed a new *conversation areas* feature, highlighted in the screenshot below:



Each conversation area is a rectangle on the map. In this screenshot, there are three conversation areas: “Foyer Table 4”, “Foyer Table 5”, and “Foyer Table 6”. There are two users who are engaged in a conversation at Foyer Table 5, who have labeled the current conversation topic to be “Talking about class”.

This Assignment

In this assignment, you will implement the backend service that supports the conversation areas feature, creating an implementation that behaves comparably to **our public deployment of covey.town**. Since this is your first time working in this codebase, Avery has sketched out a reasonable design for the Conversation Area API, and has mapped the design into implementation tasks. Avery has also provided a few sanity tests so that you can check your work. There is a complete set of tests available to you on GradeScope. This assignment does *not* involve any frontend development, but we will return to the frontend aspects of the Conversation Area feature in HW4.

Based on past experiences, we project that this assignment could take you up to 20 hours (depending on your prior preparation). We encourage you to start early so that you can post

questions on Piazza, make the most use of our TAs' tutorials, and attend office hours as necessary in order to ensure that you can reach full marks across the board. Please note that while Part 2 looks shorter, this is mainly an artifact of us providing you with less detailed steps of what you need to do: it might take you just as long as Part 1, so do not assume that it will be a quick last few steps!

Learning Objectives

The objectives of this assignment are to:

- Read an architectural diagram and apply that design to an implementation
- Expand an existing API following the coding conventions set out in an existing codebase
- Translate high-level requirements into code
- Become familiar with the Covey.Town codebase, which will be reused in HW3, HW4, and your final project

You are encouraged to write tests for your implementation, however *we will not grade your tests on this assignment*.

Grading

Your code will automatically be evaluated for linter errors and warnings. Submissions that have *any* linter errors will automatically receive a grade of 0. **Do not wait to run the linter until the last minute.**

Each implementation task will be automatically graded by our test suite on GradeScope. You may resubmit your code an unlimited number of times for feedback. Each task is allocated a set number of points based on its complexity. There are no partial marks for passing some, but not all of the tests for a function. GradeScope will provide feedback on which checks your implementation fails.

Your code will be manually evaluated for conformance to our course [style guide](#). This manual evaluation will account for 10% of your total grade on this assignment. We will manually evaluate your code for style on the following rubric:

To receive all 10 points:

- All new names (e.g. for local variables, methods, and properties) follow the naming conventions defined in our style guide
- All public properties and methods (other than getters, setters, and constructors) are documented with JSDoc-style comments that describes what the property/method does, as

defined in our style guide

- The code that you write generally follows the design principles discussed in week one. In particular, your design does not have duplicated code that could have been refactored into a shared method.

We will review your code and note each violation of this rubric. We will deduct two points for each violation, up to a maximum of deducting all 10 style points.

General Requirements

This is an individual assignment.

Please post any questions about this assignment on Piazza. We have many sections of this class, and we want to make sure that we respond to your questions the same way, regardless of which section you are in.

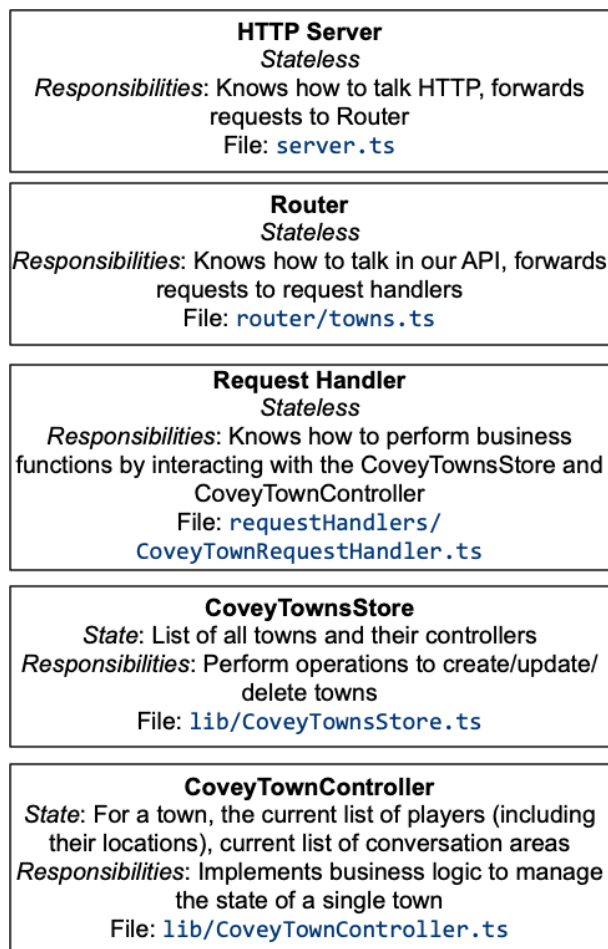
You may not make changes to `package.json` or to the lint configuration. You may not use `ts-ignore` or `eslint-disable` annotations.

Part 1: Base implementation of TownService's ConversationArea API [60 points total]

Your first major objective for this assignment will be to create a base implementation of the ConversationArea API in the backend TownService. By the end of this part of the assignment, you will have implemented a `createConversationArea` API endpoint, which will create a new `ConversationArea` on the server, and extended the existing `TownJoinHandler` so that when a player joins a town and receives the initial state of the town, the list of `ConversationArea`s will be included in that response.

Start by [downloading this starter code](#). Extract this archive and run `npm install` to fetch the dependencies. Avery has provided you with some very basic sanity tests that you can extend for testing your implementation as you go.

Each `ConversationArea` exists within a single `Town` in the overall `Covey.Town` data model. Hence, you will implement the `ConversationArea` API in the context of the existing service that implements the logic to manage each town. The diagram below outlines the major components in the existing `TownService`.



Referring to the diagram above, you'll start by implementing the ConversationArea API at the *Router* level, defining the API in terms of an HTTP endpoint (Task 1.1). Then, you will implement the business logic to create new ConversationAreas in a request handler, and in CoveyTownController (Task 1.2). Lastly, you'll update the existing request handler that processes a player's request to join a town so that this response also includes the list of current ConversationAreas (Task 1.3).

Task 1.1: Add an HTTP route to create conversation areas [5 points]

Your first task is to add a request handler to the `TownsService` to receive client requests to create a new conversation. In the Towns Service architecture diagram above, this task will modify the **Router** to know about our new REST API endpoint.

Avery has already decided on the API's specification, and in fact has already also created a client that conforms to that specification - so you will be able to start testing your implementation right away. Here is the specification of the new route that you need to add for this API:

- URL: `/towns/:townID/conversationAreas` (where `:townID` is a parameter that specified the town in which the conversation area should be created)
- HTTP Method: `post`

- Body: JSON, the body matches the type `ConversationAreaCreateRequest` , defined in `src/client/TownsServiceClient.ts` and reproduced below

```
/**
 * Payload sent by the client to create a new conversation area
 */
export interface ConversationAreaCreateRequest {
  coveyTownID: string;
  sessionToken: string;
  conversationArea: ServerConversationArea;
}
```

Avery's sanity test for this task uses their API client to make a request to create a conversation area. You can run it right now, by running the command `npm test CoveyTownConversationAPI` . Before you implement this task, you should expect to see the tests fail, with a message `Request failed with status code 404` (404 is the error code that indicates that an address does not exist).

The route. Following the example routes that already exist in `router/towns.ts` , create the new route as specified above in the file `router/towns.ts` . Your route should follow the conventions in the other routes: forward the request on to the `conversationAreaCreateHandler` and record the response. If `conversationAreaCreateHandler` throws no error, then return the response from the handler as JSON, with the HTTP status code `OK` . If `conversationAreaCreateHandler` throws an error, log the error and return a response with HTTP status `INTERNAL_SERVER_ERROR` , a JSON response of `{message: 'Internal server error, please see log in server for more details'}` .

The router interfaces with the HTTP server through the variable `app` , which in our case, is a library called **ExpressJS**. For tips on working with Express, see the **ExpressJS routing docs**



Check your work: When you run the sanity test, it should still fail, but this time with the message "Error processing request: This feature is not yet implemented" (the default behavior of `conversationAreaCreateHandler`). If you get this message, then you have successfully completed this first task!

Note: Due to the complexity of writing these tests to work with partial implementations, the GradeScope tests for Task 1.1 *will not pass* until after you get to Task 1.3. The Task 1.2 tasks do *not* have this dependency: so once you are fairly certain that Task 1.1 is implemented correctly,

continue with the rest of Task 1, and don't expect the GradeScope tests for Task 1.1 to pass until after you implement 1.3.

Task 1.2: Implement business logic to create a conversation area [45 points total]

With the top of the architecture stack implemented, it's now time to implement some business logic to actually create a conversation area. This task will require you to implement three methods, and you will receive partial credit for this task at the level of each of the implementation methods:

- `conversationAreaCreateHandler` [15 points]
- `addConversationArea` [30 points]

TASK 1.2A CONVERSATIONAREACREATEHANDLER [15 POINTS]

When the `conversationAreaCreateHandler` is called, it should:

- 1 Validate that the supplied session token is a valid token for the specified town
- 2 Delegate the actual responsibility for creating the conversation area to the `CoveyTownController`, invoking the `addConversationArea` method on the appropriate town controller.

If the token is valid and the conversation area is successfully created, set `return isOK` parameter on the returned response to `true`, the `response` property to `{}`, and the `message` to `undefined`. If not successful for any reason, set the `return isOK` to `false`, and set the message to exactly the string: `Unable to create conversation area <conversationAreaLabel> with topic <conversationAreaTopic>` (replacing the values in `<>` with the parameters that were passed). For example, if a request to create a conversation area with label `foo` and topic `bar` fails, the returned message should be `Unable to create conversation area foo with topic bar`.




Check your work: When you run the API sanity test suite (`npm test CoveyTownConversationAPI`), you should now see "Executes without error when creating a new conversation" succeed. Other tests may fail.

TASK 1.2B ADDCONVERSATIONAREA [30 POINTS]

Implement the method `addConversationArea` method in `CoveyTownController`. Recall from the architectural diagram above that the `CoveyTownController` is responsible for keeping track of all of the state regarding a single town, including its conversation areas. Avery has already added a private field, `_conversationAreas` to `CoveyTownController`. Your task is to implement the `addConversationArea` method, which should have the following behavior:

- 1 Check that the `topic` is defined: it is not permitted to create a conversation area an empty string as the topic, if this is the case return `false`.
- 2 Check to see if there is an existing conversation area with the requested `label`, and if one already exists, return `false`.
- 3 Check to see if the `boundingBox` of the new conversation area overlaps with any existing, and if so, return `false`.
 - Boxes are allowed to be *adjacent* that is, two conversation areas may share boundary points.
 - The `x`, `y` position of the box denotes the *center* of the box on the map, `height` and `width` represent the overall height and width of the box.
- 4 Any players who are in the region defined by the `boundingBox` of the new conversation area should be added to it as `occupants`, and those players should have their `_activeConversationArea` property set to that new conversation area.
 - A player is defined as inside of a box if the `x`, `y` position of the player is anywhere within the bounding box. A player who overlaps only with the edge of a conversation area's bounding box is not in the box.
 - This behavior *only* applies when a conversation area is created. After the conversation area is created, the server does *not* set the `_activeConversationArea` property on any player.
- 5 Notify all listeners that are subscribed to this town that the newly created conversation area was created, by invoking `onConversationAreaUpdated(theNewConversationArea)` on each.


This is a much bigger task than the first one. Note that you will undoubtedly find it useful to add new helper methods (private or public), perhaps in `CoveyTownController`, `Player`, or both. You *must not* add additional fields to track the conversation area's state: the data model that Avery defined is the data model that you must use!

 Check your work: Avery has also provided a single unit test for `addConversationArea`, which you can run with `npm test -- -t 'CoveyTownController addConversationArea'`. You can also run this test directly in VSCode. Of course, you may find it useful to write additional tests (either writing new tests, or modifying the behavior of this one to check more interesting behaviors). There is also a complete test suite for this task on GradeScope.

Task 1.3: Include current conversation areas in join response [10 points]

If you completed task 1.2 correctly, then all players who are in the town when a new conversation is made will be notified of the new conversation area. The last task to complete before we can move on to frontend implementation of this feature is to update the backend service to include the list of all current conversation areas when a new player joins the town.

In the file `CoveyTownRequestHandler.ts`, examine the method `townJoinHandler` and the type `TownJoinResponse`. Your objective for this task is to update the `TownJoinResponse` type to include a new field, `conversationAreas`, of type `ServerConversationArea[]`, and to update the `townJoinHandler` to include the current list of conversation areas for the town that the user has joined.

 Check your work: Run `npm test CoveyTownConversationAPI` again, you should now see the test "Includes newly created conversations when a new player joins" pass.

Checkpoint, check your work on this entire task

We suggest that you submit your code at this point on GradeScope, which will run Avery's *extremely* thorough test suite on your code. If there are test errors, you will likely find it easier to debug them *now* rather than moving on to implement more code (and perhaps introduce more bugs).

Part 2: Completing the TownService responsibilities [30 points total]

Congratulations on making it this far! You are almost done, there are only two more tasks to complete the feature.


Task 2.1: Track conversation area participants [15 points]

In Avery's design, each of the users connected to a town track which conversation area (if any) they are in, and send this information to the `CoveyTownController`. The `CoveyTownController` needs to track which users are in each conversation. `CoveyTownController` has a method `updatePlayerLocation`, which is called each time that a player's location changes. You should use the `conversationLabel` property on the `UserLocation` that is passed to `updatePlayerLocation` to identify the user's current conversation area (as reported by that user).

Your objective for this task is to implement functionality so that at the end of the execution of this method, the `_conversationAreas` list tracked by the town controller reflects that player's transition between conversation areas, updating the `occupantsByID` property on any effected conversation areas. You must also update the `Player` instance, setting the property `activeConversationArea` to be the `ServerConversationArea` instance representing that conversation area that the player is now part of (or `undefined` if they are no longer within one).


If any conversation areas are updated, you must emit a `onConversationAreaUpdated` event, similarly to how you did in Task 1.2 when a conversation is created. Note that you might need to

send multiple `onConversationAreaUpdated` events: one for a user leaving an area and one for them entering another. It is important that no client ever believe that a user is in two conversation areas at the same time. Hence, be sure to send the “exit” update before the “enter” update.

 Check your work: Again, Avery has provided a single sanity test that checks some of the behavior defined above. You can run this test with the command `npm test -- -t 'CoveyTownController updatePlayerLocation'`, and you might also consider extending it.

Task 2.2: Remove participants from conversation area if they disconnect [10 points]


When a player disconnects from the server, there is no “movement” that happens, but any resources used by that player are cleaned up by the `CoveyTownController`’s `destroySession` method. Update `destroySession` to remove disconnected players from any conversation area that they had been a participant in and emit any `onConversationAreaUpdated` events as necessary.

 Check your work: Avery has not provided you with a sanity test for this task. Consider testing it manually, or enhance the sanity test that they provided to test this behavior.

Task 2.3: Automatically end a conversation area when it’s unoccupied [5 points]

Avery has implemented logic on the frontend to show a default greeting message when there is no conversation area defined for a space. The code that you are implementing, then, needs to notify the frontend when a conversation area becomes unoccupied. When the last player leaves a conversation area emit the

`onConversationAreaDestroyed(destroyedArea:ServerConversationArea)` event to each of the town controller’s listeners, and then remove that conversation area from the town controller’s list of conversation areas. When the last player leaves a conversation area, there is no need to emit a `onConversationAreaUpdated`.

 Check your work: Avery has not provided you with a sanity test for this task. Consider testing it manually, or enhance the sanity test that they provided to test this behavior.

Submission Instructions

Submit your assignment in GradeScope. The easiest way to get into GradeScope the first time is to first **sign into Canvas** and then click the link on our course for “GradeScope”. You should then also have the option to create an account on GradeScope (if you don’t already have one) so that you can log in to GradeScope directly. Please contact the instructors immediately if you have difficulty accessing the course on GradeScope.

To submit your assignment: run the command `npm run-script zip`. GradeScope will provide you with feedback on your submission, but note that it will *not* include any marks that will be assigned after we manually grade your submission for code style (it will show 0 for this until it is graded).

© 2022 Jonathan Bell, Adeel Bhutta, Ferdinand Vesely and Mitch Wand. Released under the [CC BY-SA](#) license