Section 1: About You

What project are you applying for?

Celebrating learners' accomplishments.

Why are you interested in working with Oppia, and on your chosen project?

When I was first introduced to the field of web development a few months ago, I soon discovered that one of the best ways to learn new things is by contributing to open source projects. While searching for projects for the same, I came across Oppia (which is so far the first and only open-source project I've contributed to).

Due to me being a newbie in terms of experience in working with the technologies used in Oppia, the sheer number of resources and documentation available for new contributors to make use of was certainly something that compelled me to contribute to Oppia's development. Not to mention Oppia's mission of providing quality education to those who lack access to it, which I find amazing and would love to be a part of.

All of this combined with the fact that I'd get a chance to be productive throughout my winter break was what first got me into the Oppia project, and I've stuck around since thanks to the awesome community that is trying its best to provide meaningful education to – and consequently, have an impact on the lives of – the people who need it the most.

The reason I've picked this project is because I found it to be slightly on the creative side, and believe that this is a great step in making the experience of playing through Oppia's explorations much more enjoyable for learners, which could potentially motivate them to invest more time into learning in general (and particularly through Oppia). Additionally, I've always been rather interested in the frontend side of web development, particularly playing around with the UI. I possess a fair amount of experience in coding in Angular, and I believe I also have the ability to come up with practical designs to use for this project.

Prior experience

I've been contributing to Oppia for the past few months and am a part of the LaCE quality team. Some of my contributions while part of the team were centered around working on the UI.

I've listed some of my PRs in chronological order of when they were opened, to hopefully indicate my increasing understanding of the codebase and the practices adhered to in Oppia, as time goes on.

#14342: Replaced the sidebar on the contributor's dashboard with a dropdown to make the dashboard mobile friendly. Not the most impressive PR, partly due to it being an earlier one when I was still a beginner, but helped me learn about creating customizable dropdowns, and the basics of frontend testing.
#14452: Fixed the score circle on the practice session results page. Helped me understand the functioning of SVGs, and creating mock HTML elements to be used in frontend testing.

#14486: Implemented a new design for the search bar to make it more mobile friendly for small screen devices.
#14527: Implemented a new design for the activity cards on the community library to make them mobile friendly on small screen devices.
#14747: Added checks to validate explorations before they are added as chapters to a story.

Project size

This project is a medium sized one.

Project timeframe

I would prefer adhering to the default project timeframe, i.e. June 13 - September 12.

Contact info and timezone(s)

Name: Ch Vishnu Nithin Reddy
Email/Google Chat: nitinreddy226@gmail.com
Mobile no.: (+91) 70755 71557
Country and timezone: India, IST

Time commitment

13 June - 16 July: I will be able to commit around 30-40 hours/week.
17 July onwards: I will be able to commit around 20-25 hours/week as my classes would've begun by then.
I am also willing to vary the time I put in based on the pace at which I'm making progress in the project.

Essential Prerequisites

Answer the following questions (for Oppia web GSoC contributors):

- *I am able to run a single backend test target on my machine.*

```
[datastore] Mar 17, 2022 11:42:02 AM io.gapi.emulators.grpc.GrpcServer$3 operationComplete
[datastore] INFO: Adding handler(s) to newly registered Channel.
[datastore] Mar 17, 2022 11:42:02 AM io.gapi.emulators.netty.HttpVersionRoutingHandler channelRead
[datastore] INFO: Detected HTTP/2 connection.
06:13:38 FINISHED core.controllers.editor_test: 103.0 secs
Stopping Redis Server(name="sh", pid=7973)...
Stopping Cloud Datastore Emulator(name="sh", pid=7902)...

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   core.controllers.editor_test: 92 tests (95.7 secs)

Ran 92 tests in 1 test class.
All tests passed.

Done!
```

- *I am able to run all the frontend tests at once on my machine.*

```
17 03 2022 10:41:07.052:INFO [karma-server]: Karma v6.3.16 server started at htt
p://localhost:9876/
17 03 2022 10:41:07.054:INFO [launcher]: Launching browsers CI_Chrome with concu
rrency unlimited
17 03 2022 10:41:07.072:INFO [launcher]: Starting browser ChromeHeadless
i 「wdm」: Compiling...
17 03 2022 10:41:51.791:INFO [Chrome Headless 98.0.4758.102 (Linux x86_64)]: Con
nected on socket YZVyf8lF6_U9a8tUAAAB with id 47247195
```

```
Chrome Headless 98.0.4758.102 (Linux x86_64): Executed 7338 of 7339 SUCCESS (3 m
ins 19.098 secs / 2 mins 35.994 secs)
TOTAL: 7338 SUCCESS
TOTAL: 7338 SUCCESS
17 03 2022 10:45:41.720:WARN [launcher]: ChromeHeadless was not killed in 2000 m
s, sending SIGKILL.
Done!
```

- *I am able to run one suite of e2e tests on my machine.*

```
.    ✔✔✔ should check presence of skillreview RTE element in exploration linked to story
.    ✔✔✔ should add one more chapter to the story
.    ✔✔✔ should fail to add one more chapter with existing exploration
.    ✔✔✔ should add one more chapter and change the chapters sequences
[12:09:08] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:08] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:08] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:08] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:09] W/element - more than one element found for locator By(css selector, .protractor-test-add-prerequisite-skill) - the first result will
be used
[12:09:09] W/element - more than one element found for locator By(css selector, .protractor-test-add-prerequisite-skill) - the first result will
be used
[12:09:09] W/element - more than one element found for locator By(css selector, .protractor-test-add-prerequisite-skill) - the first result will
be used
[12:09:09] W/element - more than one element found for locator By(css selector, .protractor-test-add-prerequisite-skill) - the first result will
be used
.    ✔✔✔ should add one prerequisite and acquired skill to chapter 1
[12:09:47] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:47] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:47] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
[12:09:47] W/element - more than one element found for locator By(css selector, .protractor-test-add-acquired-skill) - the first result will be u
sed
.    ✔✔✔ should fail to add one prerequisite skill which is already added as acquired skill
.    ✔✔✔ should delete prerequisite skill and acquired skill
.    ✔✔✔ should delete one chapter and save



14 specs, 0 failures
Finished in 1031.847 seconds

Executed 14 of 14 specs SUCCESS in 17 mins 12 secs.
```

Other summer obligations

I do not have any summer obligations other than my classes which will resume on the 17th of July.

Communication channels

I prefer communicating over email or google chat for general discussions or resolving small queries, and regular weekly google meets to discuss my progress on the project (and discuss and solve potential blockers, should I encounter any), at the email ID mentioned above.

Section 2: Proposal Details

Problem Statement

Link to PRD (or N/A if there isn't one)	N/A
Target Audience	Learners of Oppia.
Core User Need	• As a learner, I currently have no indication of how much progress I've made through a chapter or when I have completed a checkpoint. • As a learner, once I reach the end of a chapter, the further learning recommendation I receive is limited, and is also (visually) not very prominent on the page (represented as a single line of blue colored text). • As a logged-out learner, the sign-up section I am presented with is easy to miss due to it being separated from the rest of the actual card content. The same applies to the exploration recommendations provided to me when not in story mode.
What goals do we want the solution to achieve?	• Make learners aware of how much progress they've made whenever they complete a checkpoint in a curated lesson, and motivate them forward. Include a brief message that varies based on the learners' progress. • Congratulate learners when they make it to the end of a chapter by adding some flair to this event (mainly using animations). • Expand the "further recommendations" a learner receives at the end of a chapter to include the option of practicing their newly acquired skills. • Make the post chapter completion recommendations the learners receive *visually* more "prominent" on the page so that they draw their attention. • Reposition (and modify) the sign-up section and the exploration recommendations that appear at the end so that they are easy-to-spot for learners.

Section 2.1: WHAT

This section enumerates the requirements that the technical solution outlined in "Section 2: HOW" must satisfy.

Key User Stories and Tasks

#	Title	User Story Description (role, goal, motivation) *"As a …, I need …, so that …."*	Priority [1]	List of tasks needed to achieve the goal (this is the "User Journey")	Links to mocks / prototypes, and/or PRD sections that spec out additional requirements.
1	Checkpoint message	As a learner, I need to see a brief summary of my progress whenever I complete a checkpointed state.	Must have	The learner completes a checkpointed state in a curated lesson	
				Desktop: A checkpoint message slides up from underneath the page footer and to the side, so as to not obstruct the card content. The message will be auto-dismissed after 10 seconds; it may also be dismissed via the CLOSE button.	Checkpoint message mocks Link to codepen demo (now visually outdated, but similar) (same link as the one in the mocks)
				Mobile: A small tooltip-like message will pop-up onto the screen, pointing to the lesson-info-modal button on the footer, that would persist on screen for 4 seconds (since the div only consists of a fraction indicating their progress, this short duration should be enough for the learner to understand the message, and the quick dismissal will ensure the message doesn't obstruct the card content for too long/distract the learner). This message would display their progress and a small checkpoint flag. The learner can tap on the lesson-info to view their progress bar, and they'll be greeted with a brief congratulatory message if they just completed a checkpointed state (should	

[1] Use the **M**o**SC**ow system ("Must have", "Should have", "Could have"). You can read more here.

				have). *The checkpoint message text should vary based on the learner's progress.**	
				Desktop: The learner may click on "SEE MORE"/"SAVE PROGRESS" to open up the lesson info modal (and save their progress if they're logged out).	
2	Lesson-info modal checkpoint highlighting	As a learner, I should be able to tell if I've just completed a checkpointed state when I open the lesson info modal (especially in mobile view).	Should have	The learner opens the lesson info modal right after completing a checkpointed state in a curated lesson.	
				The progress bar plays out a subtle animation of its most recently completed segment filling out.	
				They are also shown a checkpoint message, right under the progress bar.	Lesson-info-modal message mock
3	End chapter recommendations and milestone message for chapter completion	As a learner, I would appreciate being congratulated for successfully making it to the end of a chapter, followed by being provided with further learning recommendations that I could quickly get started with.	Must have	The learner clicks "Continue" on the penultimate state of a chapter.	
				The next state loads in with a blank card. A "check mark" animates in, right at the end of which confetti shoots out from underneath the top-nav bar (accompanied by a celebratory audio clip). The confetti then fades out, and the check mark animates out. The rest of the content (such as the sign-up section in-case the user is logged out) will be hidden during this period. *The check mark animating in may be skipped by tapping/clicking the screen while the animation is in progress..*	Lesson completion event mocks Link to rough codepen demo (slightly outdated, but similar) (same link as the one in the mocks)
				The Content of the end	

				exploration card slides up and fades into place, with the following appended to the end of it: • A section that contains a brief message congratulating the learner, *if* this was their nth-ever chapter to be completed (n = 1, 5, 10, 25, 50), • A better integrated, dismissable sign-up section that shows up for logged out users which allows them to log-in/sign-up and save their progress, • A section containing clickable cards that could lead the learner onto the next chapter/to a practice session (shows up regardless of whether the learner is logged in or not), and a button to take them back to the story page.	
4	Recommendation choice responses	As a learner, I should be led to a practice session/to the next chapter upon clicking the corresponding cards on the end exploration screen of chapter.	Must have	The learner clicks the next chapter card (if it exists) and the next chapter of the story is loaded in. Alternatively, the learner clicks the 'practice your skills' card, and they are led to the corresponding practice tab for the topic.	**Desktop**: Post chapter completion recommendation cards **Mobile**: Post chapter completion recommendation cards
5	Revamped sign-up/login section (for logged out users) and exploration	As a logged-out learner, I should be able to easily access the sign-up/login section after completing a	Must have	The learner reaches the end of a chapter. **Sign up section**: They are presented with a sign-up/login section that is better integrated into the	 Mock

	recommendation section	chapter, and should also be able to dismiss it. I should also be able to view and choose between exploration recommendations I receive at the end which should be easy to spot and concise.		conversation-skin. This section is now also made dismissable by clicking “Don’t show me again”.	
				Exploration recommendations: The exploration recommendations will now appear inside the card-container, and each recommendation will be represented as a compact card just to keep the section as a whole concise. A small button will be present at the bottom to take the learner back to the community-library.	Mock
				The learner can use the log-in/sign up buttons in the sign-up section to perform the required action. The learner can use the exploration recommendation cards to navigate to the exploration of their choosing.	

*Making the checkpoint message vary based on the learner’s progress is aimed at making the checkpoint messages dynamic and keeping them from getting stale. The message will be selected from a predefined set of messages. The benefits of this approach are as follows:

- ➔ The most obvious benefit is that the checkpoint messages won’t get stale for the learner.
- ➔ This will work seamlessly with both existing and future chapters, and the messages will be dynamic for all chapters right from the moment the checkpoint celebration mechanism is put in place.
- ➔ These messages will be stored as i18n keys, meaning translations for these messages can be easily obtained via translate-wiki (using translate-wiki for i18n is something Oppia already does, so this will integrate perfectly fine).

Technical Requirements

Additions/Changes to Web Server Endpoint Contracts

#	Endpoint URL	Request type (GET, POST, etc.)	New / Existing	Description of the request/response contract (and, if applicable, how it's different from the previous one)
1.	**learnercompletedchapterscountthandler/**	GET	New	Returns the number of chapters completed by the learner.

Calls to Web Server Endpoints

#	Endpoint URL	Request type (GET, POST, etc.)	Description of why the new call is needed, or why the changes to an existing call is needed
1.	**explorehandler/init/**<exploration_id>	GET	This call will fetch the exploration data, which includes a dictionary of states which will be used for the following: ● Determining checkpointed states (and storing them as an ordered list) We need an ordered list of checkpointed states for determining: 1. when to display the checkpoint message 2. the total number of checkpointed states 3. position of each checkpoint in the exploration (its index in a list of all checkpoints) the latter two will be required in the progress bar.
2.	**story_data_handler/**<classroom_url_fragment>/<topic_url_fragment>/<story_url_fragment>	GET	This call will fetch the story summary, which will be used to determine the next chapter in line. This next chapter will then be displayed to the learner as a recommendation when they make it to the end of the current chapter.

UI Screens/Components

#	ID	Description of new UI component	i18n required?	Mock/ spec links	A11y requirements
1.	Checkpoint message component	The component that is meant to display progress info to the learner whenever they complete a checkpoint.	Yes		Screen Reader support (alt text for svg elements, etc.)

2.	End-chapter check-mark and confetti animation	An animation involving a check-mark “filling in” and confetti shooting out from the top of the screen.	No		Display toned-down animations with reduced motion to users with vestibular motion disorders who may have their settings set to disable excess motion/animations on their screens
3.	Recommendations section at the end of the chapter	The section that is meant to contain the further learning recommendations in the form of cards.	Yes		Screen Reader support (alt text for svg elements, etc.)
4.	Milestone message section	A small section displayed on the end exploration screen of a chapter if this was the nth-ever chapter completed by the learner.	Yes		Screen Reader support (alt text for star icon, etc.)

Data Handling and Privacy

No new user data will be collected.

Other Requirements

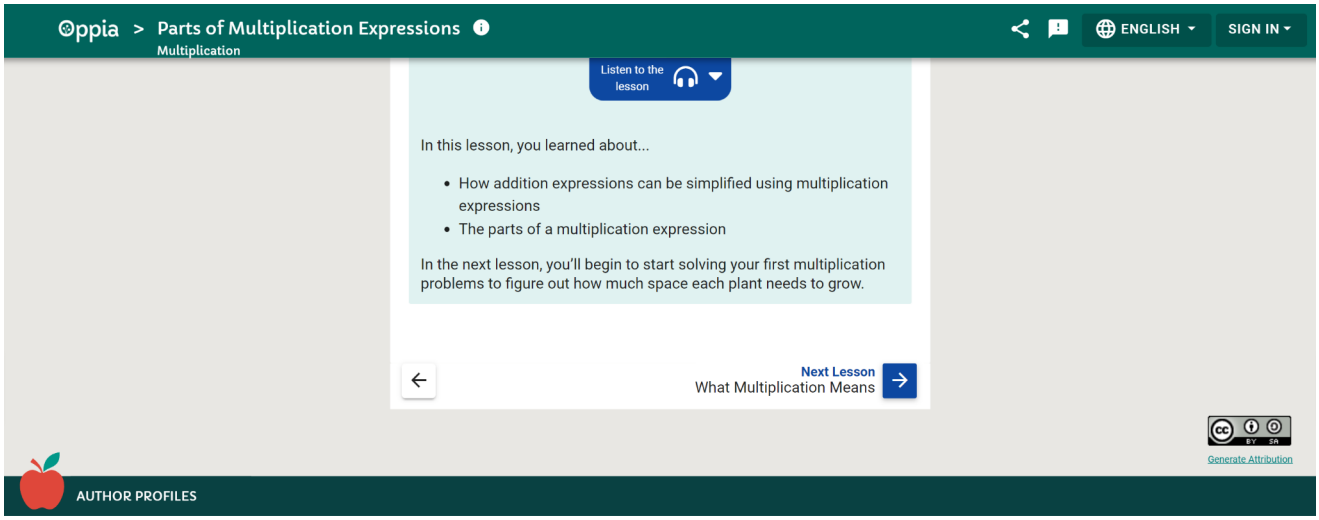
There are no other requirements.

Section 2.2: HOW

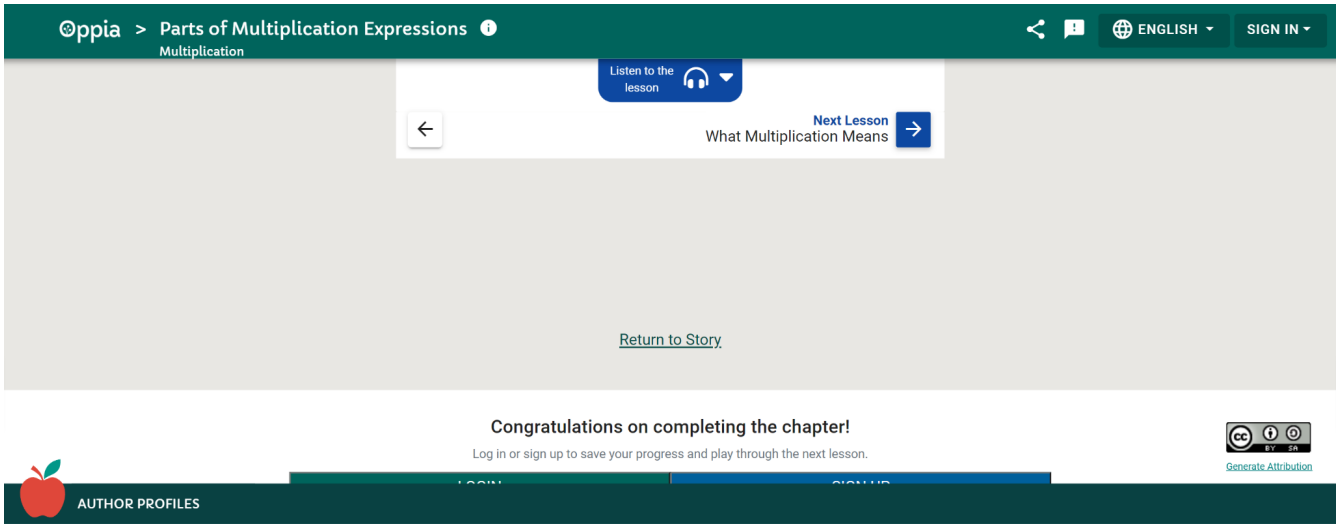
Existing Status Quo

- The learner currently has no knowledge of their progress through an exploration, nor do they have any indication of having completed a checkpoint, whenever they complete one.

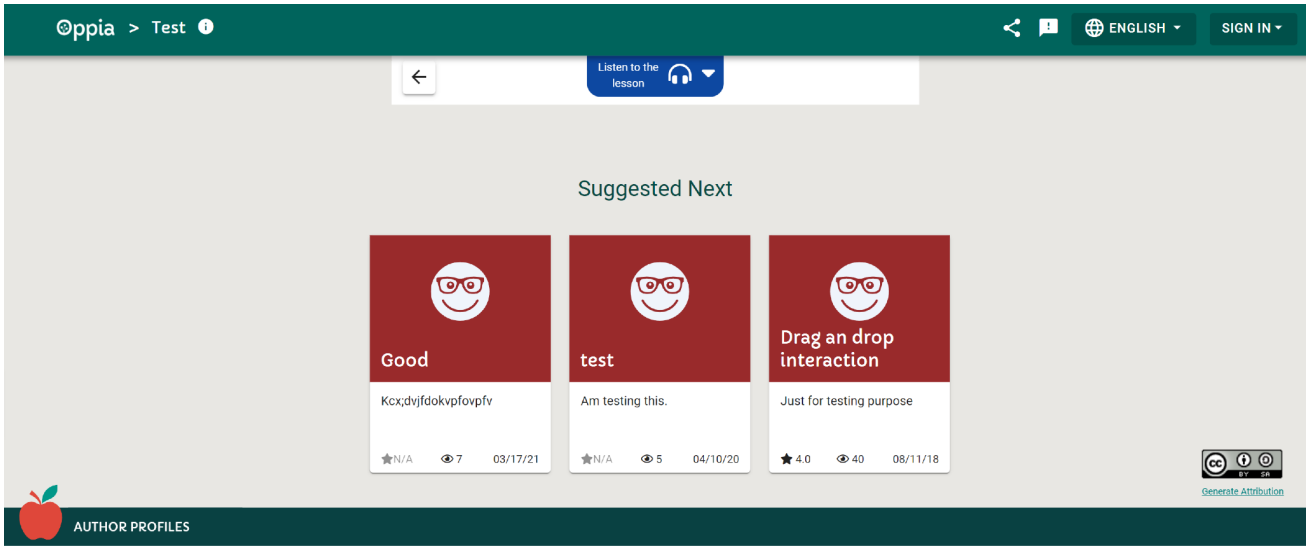
- Additionally, making it to the end of a lesson simply presents the learner with a *single* link to the next lesson, which doesn’t do a good job of drawing the learner’s attention.



- The sign-up section is also isolated from the actual card content, which may make it easier to miss for a learner.



- Similar to the sign-up section, the exploration recommendations too are not well-integrated into the rest of the conversation-skin.



Solution Overview

Checkpoint celebration event

This section is concerned with the congratulatory checkpoint event. This will comprise two parts - *checkpoint-celebration.component.ts* and *checkpoint-celebration-utility.service.ts*.

Component overview - Checkpoint celebration component

- ➔ This will be implemented as a new component - *checkpoint-celebration.component.ts*.
- ➔ This component's template will be placed inside the conversation-skin component.
- ➔ It will receive three properties as input from its parent (i.e. the conversation-skin component):
  - ◆ The *isLoggedIn* property (already present in the conversation-skin component).
  - ◆ The *initStateName* property of the current chapter (of the *ReadOnlyExplorationBackendDict* object, see the next question for how it will be obtained).
  - ◆ The *states* property of the current chapter (of the *ReadOnlyExplorationBackendDict* object, see the next question for how it will be obtained).
    Note: The states *property is a dictionary of all the states in the current exploration, along with all their individual properties (Eg: whether they are checkpointed or not, etc).*

- ➔ The *isLoggedIn* property will be used to pick the text to be displayed on one of the buttons in the checkpoint message div (this will be explained in detail when discussing the visual components of the checkpoint message div).

- ➔ The *states* dictionary will be used to obtain an array of all the checkpointed states (their names, to be specific), in the correct order (say **stateListForCheckpointMessages**). This list will be used to display to the learner their progress in the checkpoint message.
- ➔ The *init_state_name* property indicates the first state of the chapter.
- ➔ Additionally, the checkpoint messages will only be displayed within curated lessons, i.e. chapters, and not within community lessons (whether or not to display the messages will be evaluated using the *inStoryMode* boolean within the conversation skin).

How will the conversation-skin component obtain the *states* dictionary and the *init_state_name* property in the first place?

- ➔ These will be obtained by making a call to the ***explorehandler*** endpoint via the *read-only-exploration-backend-api* service, by passing in the exploration ID (already present in the conversation-skin component). One of the properties of the response object is '*exploration*', which contains the required properties.
- ➔ The return value of this call will be an object of the following type:

- ➔ The rightmost type definition displays the two properties we need (underlined)- *init_state_name* and *states*.

The function you mentioned above which retrieves the exploration data is an async function. How will you keep the checkpoint celebration component from being initialized until the response object is received?

- ➔ The component's selector will be placed inside a div with the following ngIf directive:

```
<div *ngIf="states && initStateName">
```

```
  <checkpoint-celebration [states] = 'states'
                          [initStateName] = 'initStateName'
                          [isLoggedIn] = 'isLoggedIn'>
  </checkpoint-celebration>
</div>
```

Why do you need to obtain a list of checkpoints *in order* (i.e. stateListForCheckpointMessages mentioned above)?

- ➔ The learner's progress will be displayed as number_of_checkpoints_completed / total_number_of_checkpoints.
- ➔ We need **stateListForCheckpointMessages** for both values:
  - ◆ number_of_checkpoints_completed will be determined via the index of the current checkpoint in the array**.**
  - ◆ total_number_of_checkpoints will be set equal to the length of the array.

How will you obtain a list of checkpoints (in order) from the states dictionary?

- ➔ As mentioned above, the *states* property is a dictionary where each key is the state name (unique throughout an exploration), and each corresponding value is the state's properties as a dictionary (the property we are concerned with being 'card_is_checkpoint', which is a boolean value).
- ➔ The individual state dictionaries present in this *states* property are *not* guaranteed to be present in the correct order, i.e. iterating through the *states* property from the beginning of the dictionary to the end and selecting states where 'card_is_checkpoint' is true, is not guaranteed to result in a list that is correctly ordered.
- ➔ One way to obtain the checkpoints in the correct order is via **a simple Breadth first search**. This makes use of the fact that checkpoints are not bypassable.

Why will a BFS work?

- ➔ The checkpoints of an exploration are required to be non-bypassable. This means that all the checkpoints must be visited, and that too *in only one single order* – the latter can be proved by contradiction.
- ➔ If there is more than one possible order in which checkpoints may be visited in an exploration, it means that there are at least two checkpoints which have:
  - ◆ Their own incoming paths, at least one for each (otherwise, if they don't have at least one incoming path for **each** of them, it means that the only way to the second checkpoint would be through the first, which means there is only one possible order anyway),
  - ◆ More than one path between the two of them (because all paths are unidirectional, and for a possibility of different orders to exist, there must be at least two different paths between the checkpoints, in the opposite directions),

- ◆ *At least* one outgoing path from *at least* one of them (otherwise the only possibility would be an endless loop that goes back and forth between the two checkpoints, which is not allowed).

➔ However, the above three conditions being true would mean that one of the two checkpoints would be bypassable, which is not possible.

➔ This means that, when starting from the initial state, there is only one possible order in which the checkpoints will be encountered (which includes all the checkpoints). *The only problem is that there are multiple paths that can be taken to reach the terminal state*.

➔ This is where BFS (or any graph traversal method, really) comes in. It will traverse through the entire states dictionary (covering all possible paths), making sure not to revisit any states (thus avoiding loops), and will push a checkpointed state into the **stateListForCheckpointMessages** array whenever it encounters one.

How will you perform a BFS over the states dictionary?

➔ This will be discussed in detail in the Service overview for the *checkpoint-celebration-utility* service further below (more specifically, one of the three functions in the service will be designed for this very purpose):

```
ngOnInit(): void {
    // ...
    this.stateListForCheckpointMessages =
this.checkpointCelebrationUtilityService.getStateListForCheckpointMessages(
      this.states, this.initStateName);
    // ...
}
```

Why not store the list of checkpoints as a property of the exploration in the backend?

➔ The following table summarizes the comparison between the two approaches, and why I've decided to compute the list of checkpoints as needed, instead of storing a precomputed list:

	Adding a new property to the exploration backend dict to store a list of all checkpoints	**Computing the list of checkpoints as needed in the frontend via a BFS**
Simplicity of implementation	Currently the only source for whether a state is checkpointed or not is the states object, which contains the properties of each of the states of an exploration. Adding a list of checkpoints as a property	No new property is added, all we need to do is perform a simple BFS after retrieving the exploration data from the backend, as and when needed.

	of an exploration will introduce another source for the same piece of information. This means we would have to do the following: - Decide which of the two will act as the single source of truth for this piece of information (and ensure the other isn't editable by any means other than below). - Figure out a way to detect when the checkpoint property of the states of an exploration have been changed and recalculate and store the checkpoint list whenever a change is detected.	
Performance when in production (for every GET request)	- The response object for every GET request is slightly increased in size. - The returned object has a precomputed list that can be readily used.	- The response object is the same size as it was before. - A BFS needs to be performed on the 'states' property of the returned object before use.
Performance when in production (for every POST/PUT request)	- A check needs to be performed to determine any changes in the states dictionary that may have caused the checkpoint list to be outdated. - If yes, A BFS needs to be performed to recompute the checkpoint list and	- Same as before.
**The BFS mentioned in the above cells has a negligible computation cost, considering the size of the explorations in Oppia.*		
Additional/redundant code required	New code needs to be added for the following: - Checking for changes within the states dictionary after every post request. - Performing a BFS and recomputing the checkpoints list if the above is found to be true.	Code needs to be added only to call the function performing the BFS, and to filter out the checkpointed states. The actual BFS will be performed using the *computeBfsTraversalOfStates()* function from the

		compute-graph service, which allows code to be reused.

How will you decide when to display the checkpoint message div?

➔ This will be achieved as follows:

- ◆ subscribing to the onPlayerStateChange event emitter from the exploration-player-state service (this emitter fires whenever the current state is completed, and also provides the newStateName).
- ◆ calling *checkIfCheckpointAnimationToBeTriggered()* whenever the emitter is fired.

```
this.explorationPlayerStateService.onPlayerStateChange.subscribe(
      (newStateName) => {
        this.checkIfCheckpointAnimationToBeTriggered(newStateName);
      }
)
```

- ◆ The function *triggerCheckpointCelebration* will look as follows:

```
  checkIfCheckpointAnimationToBeTriggered(newStateName: string): void {
    if (!this.oldStateName) {
      this.oldStateName = newStateName; // Setting oldStateName if the
component has just been initialized.
      return;
    }
    if (newStateName === this.oldStateName) { // Checking if the state has
changed, terminating if not.
      return;
    }
    let checkpointPosition =
this.stateListForCheckpointMessages.indexOf(this.oldStateName); // Obtaining
the position of the checkpoint in the exploration.
    if (checkpointPosition === -1) { // Checking if the state is a checkpoint
(i.e. present in the checkpoint list), terminating if not.
      this.oldStateName = newStateName; // Updating oldStateName before
terminating.
      return;
    }
```

```
    this.currentCheckpointMessage =
this.checkpointCelebrationUtilityService.getCheckpointMessage(
      checkpointPosition, this.stateListForCheckpointMessages.length); //
Picking a checkpoint message.
    this.currentCheckpointTitle =
this.checkpointCelebrationUtilityService.getCheckpointTitle(); // Picking a
checkpoint title.
    this.triggerAnimation(); // Triggering the animation with the required
message and title.
    this.oldStateName = newStateName; // Updating the oldStateName to the new
one.
  }
```

In the above function, how will you pick a checkpoint title and message based on the checkpoint's position?

- ➔ This too will be discussed in detail in the overview for the *checkpoint-celebration-utility* service, further below.

How will you decide which kind of message div to trigger the animation for - the full-scale one or the minified mobile-friendly one?

- ➔ This will be decided (within the *triggerAnimation()* function) based on the screen width (which can be determined via the *windowDimensionsService* – by subscribing to the *resizeEvent* and re-evaluating whether the screen is too small for the full-scale message or not whenever the width changes).
- ➔ The minified message will be displayed when the screen width falls below 960px (I may switch this out for a more suitable value during actual implementation).

How will you trigger the animation of either style of the checkpoint message?

- ➔ The message div will originally have a *bottom* property value of -100vh. This ensures the message is hidden off-screen.
- ➔ The *transition-property* of the div will be set to *bottom*, and the *position* will be set to *fixed*.
- ➔ Animating the message will be achieved via a function setting the value of a property associated with the *ngClass* directive of the message div (for example, ***showFullScaleMessage***) to true – thus applying a class to the message div whose *bottom* value will be equal to 50px, causing the div to move upward onto the screen.
- ➔ The minified message's animation will be triggered in a similar fashion.

Service overview - Checkpoint celebration utility service

This service will contain three functions - **one to obtain an ordered checkpoint list** (by performing a breadth-first search), **one to pick a checkpoint message** based on the learner's current position, relative to the total number of checkpoints in the chapter, and **one to pick a checkpoint title**.
It will import the compute-graph service for performing the BFS.

FUNCTION FOR RETRIEVING AN ORDERED CHECKPOINT LIST

- ➔ The checkpoint list will be obtained using **a breadth-first search**. It will take two values as input: the *states* dictionary and the *initStateName* string.
- ➔ Components of the search:
  - ◆ A State object will be generated using the backend dict before beginning the search.
  - ◆ The actual search will be performed by the *computeBfstraversalOfStates()* function, which will return a list of all states, in BFS order.
  - ◆ The list will then be filtered through to obtain all the checkpointed states, which will be added to a separate list.

The code may look something like this:

```
getStateListForCheckpointMessages(statesbackendDict: StateObjectsBackendDict,
initStateName: string): string[] {
    let states =
this.statesObjectFactory.createFromBackendDict(statesbackendDict); // Creating
State object from the backend dict, for use with the compute graph service.
    let BfsStateList =
this.computeGraphService.computeBfsTraversalOfStates(initStateName, states,
initStateName); // Obtaining a BFS ordered list of all states.
    let stateListForCheckpointMessages = []; // List to store states for
checkpoint messages.
    BfsStateList.forEach((state) => {
      if (statesbackendDict[state].card_is_checkpoint) { // Checking whether a
state is checkpointed, and adding it to the list if so.
        stateListForCheckpointMessages.push(state);
      }
    })
    stateListForCheckpointMessages.shift(); // Removing the initial state from
the checkpoint list (see below for why).
    return stateListForCheckpointMessages;
```

```
  }
```

Why are you not keeping the first checkpoint in the checkpoint list?

- ➔ The initial state of an exploration is required to be a checkpoint.
- ➔ However, congratulating the learner for completing the very first state of an exploration (which more than likely will be a “welcome” card, explaining what they are going to learn) is meaningless.
- ➔ Thus, the first checkpoint of an exploration, i.e. the very first state of the exploration, will not be treated as a checkpoint (even though it is one).

Based on the above code, the terminal state doesn’t seem to be included in the checkpoint list. This means if the learner completes the last checkpoint of an exploration but is yet to make it to the end exploration state, their progress bar will still show 100% completion, which is misleading. How will you handle this?

- ➔ The number of total checkpoints will be slightly tweaked to be calculated as follows: (Length of the checkpoint list + 1). The (+1) will account for the end exploration state, and the progress bar won’t show 100% completion after completing the final checkpoint, and won’t be misleading to the learner.

FUNCTION FOR RETRIEVING A CHECKPOINT MESSAGE

- ➔ This function is meant to return a checkpoint message based on the learner’s position in the exploration, relative to the total number of checkpoints in the exploration. It will take two values as input – the learner’s current position (in terms of completed checkpoints) and the total number of checkpoints in the exploration.
- ➔ The function to pick a checkpoint message will look like this:

```
  getCheckpointMessage(checkpointPosition: number, totalCheckpointCount:
number): string {
    let messages = [
      [
        'You just completed the first checkpoint! Good start!',
        'Great work completing your first checkpoint! Keep it going!',
        'A perfect start! Keep it up!'
      ],
      [
        'You\'re making good progress! Keep going!',
        'Amazing! You just completed your second checkpoint!',
        'One more checkpoint completed, you\'re doing great!'
      ],
      [
```

```
        'You\'re halfway through, you\'ll be done in no time!',
        'You just made it halfway, nice work!',
        'Wow! You\'ve already made it halfway through the lesson! Amazing
work!'
      ],
      [
        'You\'re almost there! Keep it up!',
        'You\'ve almost made it to the end! Keep it going!',
        'Nice work! You\'re almost at the finish line!'
      ],
      [
        'Just one more to go, woohoo!',
        'Let\'s go! Just one more left!',
        'You\'re doing great, just one more to go!'
      ],
      [
        'You completed a checkpoint! Good job!',
        'Awesome, you completed a checkpoint! Keep going!',
        'Nice work! You just completed a checkpoint!'
      ]
    ]
    checkpointPosition++; // To account for zero-based indexing.
    if (checkpointPosition == 1) {
      return messages[0][Math.floor(Math.random() * 3)];
    } else if (checkpointPosition == 2) {
      return messages[1][Math.floor(Math.random() * 3)];
    } else if (checkpointPosition / totalCheckpointCount >= 0.5 &&
(checkpointPosition - 1) / totalCheckpointCount < 0.5) {
      return messages[2][Math.floor(Math.random() * 3)];
    } else if (totalCheckpointCount - checkpointPosition == 2) {
      return messages[3][Math.floor(Math.random() * 3)];
    } else if (totalCheckpointCount - checkpointPosition == 1) {
      return messages[4][Math.floor(Math.random() * 3)];
    } else {
      return messages[5][Math.floor(Math.random() * 3)];
    }
  }
```

FUNCTION FOR RETRIEVING THE CHECKPOINT TITLE

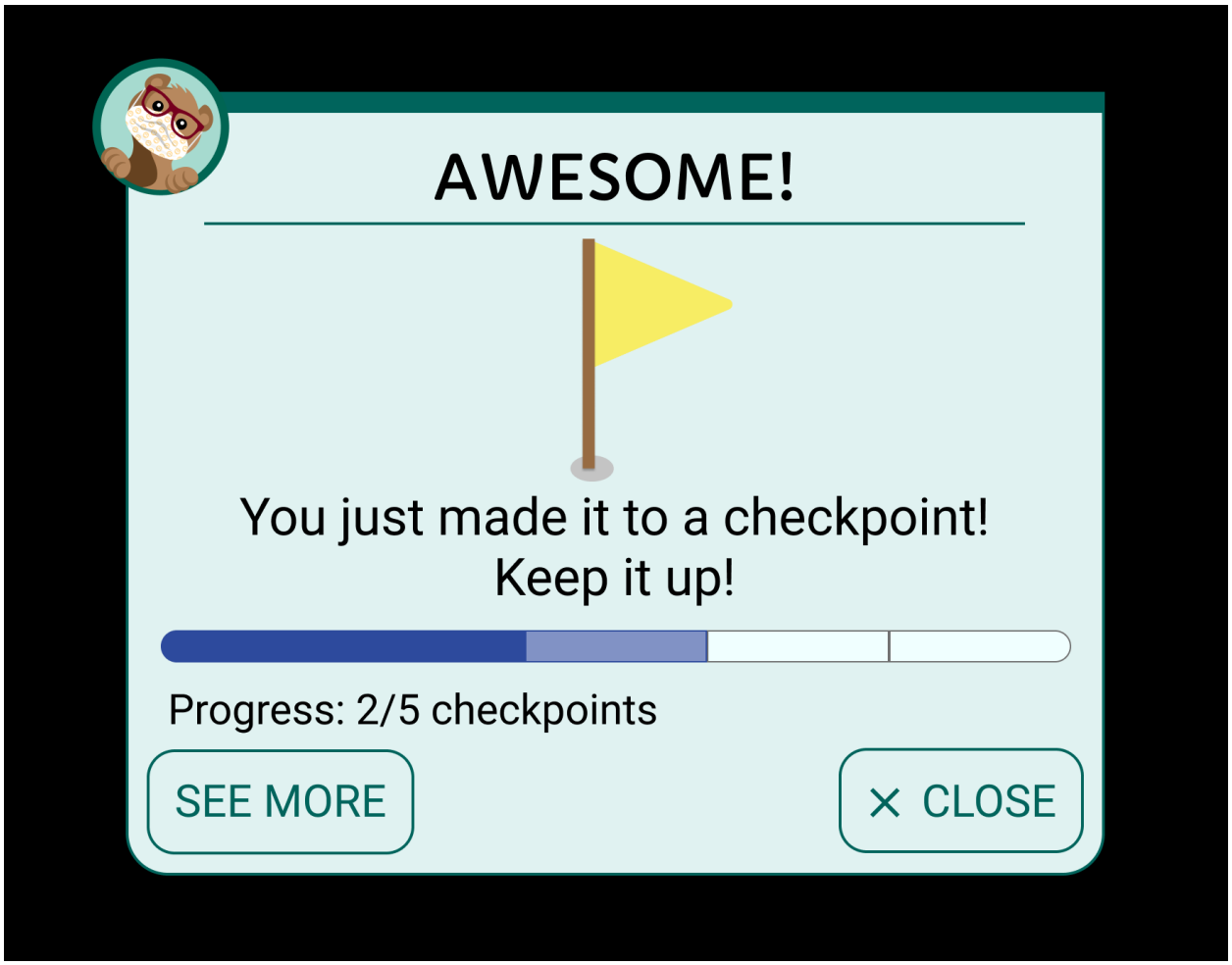
```
getCheckpointTitle(): string {
  const titles = [
    'Hurray!',
    'Awesome!',
    'Checkpoint!',
    'Good job!',
    'Great work!',
    'Well done!',
    'You Rock!'
  ]
  let r = Math.floor(Math.random() * titles.length);
  return titles[r];
}
```

Note: The strings above (for the checkpoint title and the checkpoint message) will be replaced with their corresponding I18N keys, which will be created during implementation. Actual human-readable messages have been used here just for ease-of-understanding. Please see the Internationalization sub-section of the Miscellaneous section further below, for how the keys will be used to obtain translations.

Component overview - Checkpoint celebration component (continued)

FULL-SCALE DESKTOP MESSAGE

Intended design



Consists of an **SVG flag** (which will animate into position), an **SVG timer** at the top (the message will be auto-dismissed when the timer fully depletes), an image of **the Oppia mascot**, **title text** and **message text**, the **progress-bar**, and **two buttons at the bottom** (discussed in a bit more detail down below).

How will you trigger the animation of the flag bouncing into position?

- ➔ The flag will be animated using a keyframes animation, by applying a class with the required CSS animation property using the *ngClass* directive (the directive will be "activated" by setting a property, say, ***animateFlag,*** to true).
- ➔ Additionally, the animation will be triggered inside a *setTimeout()* call, to ensure it only occurs after the message div fully appears on-screen.
- ➔ The keyframes rule for the flag's bounce animation will be as follows:

```
@keyframes flag-bounce {
  0%   { transform: scale(1, 1) translateY(0); opacity: 0; }
```

```
  10%  { transform: scale(1.05, .95) translateY(0); }
  30%  { transform: scale(0.90, 1.1)  translateY(-30px); opacity: 1; }
  50%  { transform: scale(01, 1) translateY(2px); }
  60%  { transform: scale(1.2, 0.90) translateY(5px); }
  70%  { transform: scale(1, 1) translateY(0); }
  100% { transform: scale(1, 1) translateY(0);}
}
```

How will you trigger the animation of the SVG timer?

- ➔ The timer SVG is a line element with start and end coordinates that lie on either edge of the message div.
- ➔ Two of an SVG's properties (among many) are its ***stroke-dashoffset*** and its ***stroke-dasharray***.
  - ◆ The ***stroke-dasharray*** property defines the patterns and gaps of an svg element. A dasharray value greater than or equal to the length of the SVG element renders the line as completely filled.
  - ◆ The ***stroke-dashoffset*** property defines the amount by which the line is offset from its "resting" position. A dashoffset value set to the length of the element completely offsets it by its entire length, thus rendering it invisible.
- ➔ Animating the SVG timer can be achieved by:
  - ◆ initially setting ***stroke-dasharray*** equal to the length of the line and ***stroke-dashoffset*** equal to 0 (i.e. the line is completely visible),
  - ◆ setting the *transition-property* to ***stroke-dashoffset*** with a suitable duration (10s),
  - ◆ and triggering the animation by raising the dashoffset value to be equal to the length of the line (this can be achieved by using the *querySelector()* method to select the timer element, and then setting its dashoffset value to the length of the timer (10 units)).

```
  // ...
  animateTimer(): void {
    let timer = document.querySelector('.timer-element') as
SVGPolylineElement;
    timer.style.strokeDashoffset = '10';
  }
  // ...
```

How will you place the Oppia mascot in the top right?

- ➔ It will be an image that will have its position set to absolute, and its top and left values set to a small negative value (-15px) to have the image pushed out of the message div.

How will you auto-dismiss the message div?

- ➔ Similar to how the div was animated in by setting the ***showFullScaleMessage*** property to true, a function will be used to flip the property's value back to false, causing the div's *bottom* property to be reverted back to its original value. Since the *transition-property* is set to *bottom* the div will animate downward, off-the-screen.
- ➔ This function will be called within a *setTimeout()* call a certain duration (10 seconds) after the message first appears.

How will you generate a progress bar to be displayed in the message?

- ➔ The progress bar will be enclosed in a fixed size div regardless of the total number of checkpoints. The number of segments in the bar will be equal to the total number of checkpoints – this means that the width of each segment will decrease as the total no. of checkpoints increases.
- ➔ Width of a segment = total width / number of checkpoints (*100 to obtain the %age of total width).
- ➔ The above percentage will be set to the *[style.width.%]* attribute of each segment to render the progress bar's outline.
- ➔ The number of checkpoints completed by the learner will be used to determine the number of filled (in blue) segments by calculating (width of each segment) * (number of checkpoints completed).
- ➔ One additional segment of light blue color will be generated if the number of completed checkpoints is less than the total number of checkpoints (to indicate to the learner which segment they are currently working towards completing).
- ➔ Colors that the progress bar will be composed of (please view the intended design image above for reference):
    - ◆ Progress bar outline - #707070
    - ◆ Yet to complete segments - #F0FFFF
    - ◆ Currently working towards completing - #2D4A9D99
    - ◆ Completed - #2D4A9D

The markup may look as follows (<u>this is almost the same as the progress bar present in the lesson-info-modal – one of the decisions made in the figma mocks' discussions was to ensure both the progress bars look identical</u>):

```
<div class="progress">
  <!-- border: 1px and border-radius: 10px -- the progress bar outline -->
  <div class="progress-bar-separator-container">
    <!-- position: absolute, width: 100% -- contains the lines that separate
the progress bar into different segments; the actual colored segments
```

```
indicating the progress will lie over this element since position is set to
absolute-->
    <div class="progress-bar-separator" [style.width.%]="100 /
total_number_of_checkpoints">
        <!-- border: 1px -- This will be n instances of this element (n =
number of checkpoints) separate the progress bar into the required number of
sections-->
    </div>
  </div>
  <div class="completed"
       [style.width.%]="(100 / total_number_of_checkpoints) *
number_of_checkpoints_completed">\
       <!-- represents the completed part of the progress bar -->
  </div>
  <div *ngIf="completedWidth != 100"
       class="working"
       [style.width.%]="100 / total_number_of_checkpoints">
       <!-- represents the part the learner is currently working towards
completing (meaning it will be one segment long), the ngIf ensures this will
remain invisible if the progress bar is 100% complete, so it doesn't appear
outside of the progress bar -->
  </div>
</div>
```

What will the SEE MORE button do?

- ➔ The button will read 'SEE MORE' for logged-in users, and 'SAVE PROGRESS' for logged-out users. The text to be displayed will be chosen based on the *isLoggedIn* property that is received as input.
- ➔ Clicking the button will open up the lesson info modal.

How will the SEE MORE button open the lesson-info modal?

- ➔ The component will possess an event emitter, which will be subscribed to in the exploration-footer component.
- ➔ Clicking 'SEE MORE' will emit a signal via the event emitter and *openInformationCardModal()* – the function responsible for creating a new instance of the lesson-info modal – will be called in the exploration-footer's subscription.

What will the CLOSE button do?

➔ It will dismiss the message (by reverting the *bottom* property of the message div to its default value).

Note: The auto-dismissal of the message mentioned above will, of course, take place by calling the necessary function inside a setTimeout() call. The return value of the call will be stored in a variable. We need to store this in a variable because we may later need it to clear the timeout if the close button is clicked i.e. ***we need to prevent the auto-dismissal function from being called if the message is manually dismissed by clicking CLOSE****.*

```
// ...
this.dummyTimeoutVariable = setTimeout(() => {
  this.autoDismissFunction();
}, 8000);

// ...
closeButtonClicked() {
  clearTimeout(this.dummyTimeoutVariable);
  // Rest of the code...
}
```

MINIFIED MOBILE MESSAGE

Intended design

The tool-tip will appear on-screen and will remain in-place for a constant amount of time (4 seconds), before vanishing.
It will point to the lesson-info-modal button in the exploration-footer.
The tooltip will display the user's progress, and a checkpoint flag to indicate what the displayed value is about.

What will be different in the lesson-info modal when the user opens it after completing a checkpointed state?

- ➔ The modal will additionally display a checkpoint message right below the progress bar.
- ➔ The progress bar itself will also exhibit a subtle animation (filling up to the current progress) to indicate to the learner that this is what they just achieved.

When will you display a checkpoint message in the lesson-info-modal?**

- ➔ The approach to determine when the checkpoint message is to be displayed will be similar to the solution for when to display the checkpoint message div, which was outlined above (i.e. using the *checkpoint-celebration-utility* service to obtain a list of checkpointed events, subscribing to the *onPlayerStateChange* event emitter of the exploration-player-state service to get the newStateName whenever the state changes, and so on).

How will you display a checkpoint message in the lesson-info-modal?**

- ➔ The utility service will be used for obtaining the checkpoint message.
- ➔ It will be displayed as simple bold text underneath the progress bar in the *lesson-information-modal*.

**The solutions outlined for the above two questions are very barebones because they are based on the current state of the code in the *lesson-information-modal* component – the code is incomplete at the moment as the user checkpoints project is still underway. Once the project is complete and I have the complete code to work with, I may/may not be able to come up with better, cleaner solutions (note that the solutions described above *will* work – it's just that I may (or may not) be able to come up with a better solution once I have the completed code to work with).

Ideally the user checkpoints project should be completed before the CBP (which is when I plan on further analyzing the code to come up with more suitable solutions), but I can't say anything for certain right now as the "Implementation plan" section of the project's design doc is not complete at the moment, meaning I can't estimate when the project will be completed.

How will you add the "subtle animation" that you mentioned above, to the progress bar?

- ➔ The progress-bar's segments, as stated previously, are divs with a particular width (which change as the learner makes more progress).
- ➔ Animating the progress bar in the lesson info modal will be achieved like so:
  - ◆ Determining when the animation needs to take place, i.e. when the learner has just completed a checkpointed state.
  - ◆ Calculating the previously completed width = ***completedWidth*** - ***widthOfOneSegment*** (both the properties are present in the lesson-information-modal component's class).
  - ◆ Temporarily setting ***completedWidth*** to the above calculated value, instead of its actual value.
  - ◆ Setting ***applyTransitionProperty*** to true, which applies the following class to the progress bars' segments (via the *ngClass* directive):

```
.set-transition-property {
  transition-property: width;
  transition-duration: 4s;
}
```

- Reverting ***completedWidth*** back to its actual value, thus triggering the animation – the progress bar will show the most recently completed segment animating in (filling in).
- Reverting ***applyTransitionProperty*** back to false (inside a *setTimeout()* call – 4.5s – to ensure the value is only reverted back AFTER the animation fully completes).
- The last step is to ensure the *.set-transition-property* class is removed, so that this animation won't take place anymore for regular states (until another checkpointed state is encountered).

End chapter celebration event

The section is concerned with the end chapter celebration event, consisting of the **SVG check mark** and the **confetti**.

Component overview - Check mark

- ➔ This will be implemented as a new component - *end-chapter-check-mark.component.ts.*
- ➔ This component's template will be placed inside the conversation-skin component.
- ➔ This component's intended purpose is purely visual. There is no need for any sort of input from its parent component.

Component overview - Confetti

- ➔ This will be implemented as a new component - *end-chapter-confetti.component.ts.*
- ➔ This component's template will be placed inside the conversation-skin component.
- ➔ This component's intended purpose is purely visual. There is no need for any sort of input from its parent component.

When will you trigger the check mark and confetti animations?

- ➔ The animation will be triggered when the learner hits continue on the penultimate state, i.e. the state whose immediate successor is the end exploration state.
- ➔ The animation will be triggered within setTimeout() calls to ensure proper timing. The details of **how the animation will be triggered** (for both the check mark and the confetti), how **the skipping of the animation** will function, and how **the rest of the content on the page** will be presented (or hidden, rather) during this animation are discussed below.

How will you trigger the animations?

- ➔ The animations will be triggered by calling functions responsible for triggering the animations (present in their respective components) from their parent component, i.e. the *conversation-skin* component.
- ➔ This will be achieved using the *ViewChild* decorator. This decorator allows the parent component to access and call properties and methods in the child components:
  - ◆ Import the *ViewChild* decorator into the parent component.
  - ◆ Import the component class whose methods you wish to gain access to into the parent component.

```
import { ViewChild } from '@angular/core';
import { ChildComponent } from '...';
```

  - ◆ Create an instance of the class using the *ViewChild* decorator.

```
@ViewChild(ChildComponent) childComponent: ChildComponent;
```

  - ◆ Use this instance of the class to access methods pertaining to the child component whenever required.

How will you decide when to call the functions which will trigger the animation?

- ➔ As stated above, the animations are meant to be triggered when the learner hits continue on a penultimate state, i.e. as soon as the terminal state of a chapter is loaded.
- ➔ The above happens when the learner clicks continue on a state whose immediate successor is a terminal state (currently the only interaction which is terminal is the end exploration interaction).
- ➔ This means the animations for the check mark and the confetti need to be triggered when
  - ◆ The learner hits continue on a state and
  - ◆ The immediate successor state is an end exploration state.
- ➔ Clicking ‘Continue’ on a state may call one of two functions in the *conversation-skin* component:
  - ◆ *showUpcomingCard()* if the current interaction type is not ‘Continue’ meaning the learner just submitted an answer and then clicked continue.
  - ◆ *submitAnswerFromProgressNav()* if the current interaction type is ‘Continue’ meaning the learner clicked the only button they were presented with.
- ➔ Given the information above, the easiest way to trigger the animation is as follows:
  - ◆ Call a helper function that performs some checks and then calls functions in the child components which are responsible for triggering the animations (for

example, in case of *submitAnswerFromProgressNav()* the helper function call will look like so):

```
 submitAnswerFromProgressNav(): void {
   this.currentInteractionService.submitAnswer();
   if (this.displayedCard.getInteractionId() === 'Continue') { //
submitAnswerFromProgressNav() is also called whenever the submit button is
clicked (for interactions of type other than Continue), therefore this check is
necessary. Such a check isn't required in case of showUpcomingCard().
     this.triggerEndAnimationIfTerminal(); // Helper function that performs
checks before calling functions from the child components (the check mark and
confetti components) that will trigger the animations.
   }
 }
```

➔ The helper function will then check whether the successor state is terminal or not (triggering the animation if it is), like so:

```
 triggerEndAnimationIfTerminal(): void {
   if (this.nextCard && this.nextCard.isTerminal()) {
     console.log('This log statement will be replaced with functions from the
check mark and confetti components which will trigger the animations. As
stated above, the child components will be made accessible here using the
ViewChild decorator.');
   }
 }
```

How is the check mark animation meant to play out?

➔ Similar to the SVG timer in the checkpoint celebration component, the check mark is completely made up of SVG elements:
- ◆ The outer enclosing element is a circle SVG element,
- ◆ The tick within the circle is a polyline SVG element that goes through three distinct points (vertices).

➔ The mark-up of the check mark will be as follows:

```
<svg class="check-mark-svg" viewBox="0 0 40 40">
  <circle class="circle"
          stroke="#00645c"
```

```
          stroke-width="3"
          fill="none"
          r="15"
          cx="20"
          cy="20">
  </circle>
  <!-- The stroke-dashoffset and stroke-dasharray of the circle element will be
set to 94.26 i.e. its circumference -- the circle will initially be rendered
completely invisible. The transition-property will be set to stroke-dashoffset.
-->


  <polyline class="tick"
            fill="none"
            stroke="#00645c"
            stroke-width="3"
            points="11,20 16,25 28,15" />
  <!-- The stroke-dasharray and stroke-dashoffset of the tick will be set to 23
i.e. its length -- the tick will initially be rendered completely invisible.
The transition-property will be set to stroke-dashoffset. -->

</svg>
```

➔ The end result (after fully animating in) looks as follows:

- ➔ The animation is intended to play out like so:
  - ◆ *Animation begins*: The circle starts animating in (filling up),
  - ◆ *250ms after beginning*: The tick starts animating in.
- ➔ It will be animated out in a similar fashion.

How will you trigger the check mark animation?

- ➔ As described in the previous section, an SVG element has two important properties responsible for how the element ends up being rendered – its stroke-dasharray, and its stroke-dashoffset. Both will initially be set to the lengths/circumference of the elements for both the elements (the circle and the tick) i.e. both will initially be invisible.
- ➔ The function to trigger the animation may look like this:

```
checkMarkAnimateIn(): void {
  this.circleIsShown = true;
  setTimeout(() => {
    this.tickIsShown = true;
  }, 250);
```

```
}
```

➔ Each of the properties (**circleIsShown** and **tickIsShown**) will be associated with the *ngClass* directive on the corresponding SVG elements. Setting these values to true will apply a class to the elements, the only property of which will be "stroke-dashoffset: 0;".

```
.check-mark-transition-class {
  stroke-dashoffset: 0;
}
```

➔ SInce the transition-property is set to stroke-dashoffset, the SVGs will animate in upon calling the above function. The function for the SVGs to animate out will look similar.

How is the confetti animation meant to play out?

➔ There are three kinds of confetti pieces (of various colors, a total of 17):
- ◆ a rectangle block piece (a simple HTML div),
- ◆ a circle piece (a HTML div with border-radius set to 50%),
- ◆ a squiggly line-like piece (an SVG element made using inkscape).

➔ They will be placed underneath the top-nav bar at different "resting" positions before the animation is triggered, *with various colors and slightly different animation-delays just to add some randomness*:

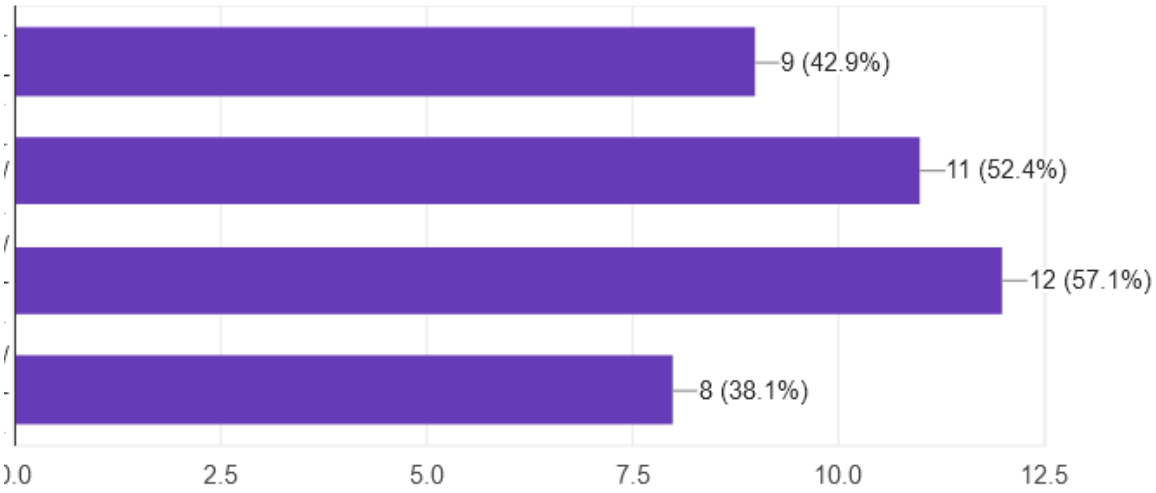
```
.confetti {
  position: fixed;
  width: 10px;
  height: 34px;
  top: 40px;
  left: 53%;
  opacity: 0;
  background: #0000FF;
}
.confetti:nth-child(2n) {
  left: 47%;
  background: #00AA00;
  width: 14px;
  height: 40px;
}
.confetti:nth-child(3n) {
```

```
  background: ##FF0000;
  width: 8px;
  height: 30px;
  animation-delay: 200ms;
}
.confetti:nth-child(4n) {
  width: 20px;
  height: 20px;
  border-radius: 50%;
  animation-delay: 200ms;
}
.confetti:nth-child(7n) {
  background: #FFFF00;
  width: 10px;
  height: 30px;
  animation-delay: 400ms;
}
```

- ➔ Each individual piece of confetti is meant to shoot out from underneath the top-nav bar, spread over the top-fourth of the screen and hang for a moment before fading out.
- ➔ Additionally, this audio clip will accompany the confetti shooting out, and will play alongside it.
- ➔ The clip will be stored within the assets folder, within the (new) 'audio' subdirectory, and will be loaded into the confetti component upon its initialization. It will then be played along with the confetti animation.

What other clips were considered for this purpose?

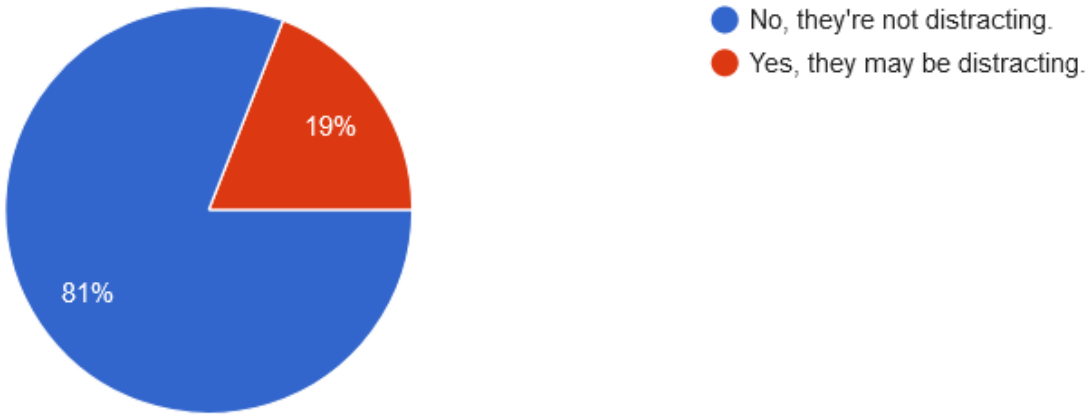
- ➔ The following clips were considered:
  - ◆ Option 1
  - ◆ Option 2 (the option which ended up being selected)
  - ◆ Option 3
  - ◆ Option 4
- ➔ A short survey was then carried out, the results of which are below:



➔ Despite not being the most popular option, the second clip was selected on account of it sounding more “grand” than the rest (which, although a subjective opinion, was shared by other reviewers as well), and it suits the event of the confetti shooting out far better than the rest.

Will the audio clip be skippable/can it be muted?

➔ No, the clip will not be skippable. The clip is played at the end of a lesson, meaning it will not be distracting to the learner in any way. In addition to this, according to a small survey (among people who are NOT active Oppia developers), the most popular opinion was that such audio clips would not be distracting to the learner:



➔ Finally, a dedicated button to mute the audio will then require the learner to go out of their way (for example, to the user preferences page) to re-enable the audio. In contrast, muting the device audio as and when required is a much more convenient way of handling it, should the learner find the audio distracting.

➔ Considering the above reasons, the audio clips will not have any dedicated option/button to have them muted.

The above two discussions are summarized below:

Which clip do you think is suitable for this purpose?	**Do you think the clips may be distracting?**	**What do you think is a good way to mute these clips if need be?**
Option 2 sounds the most suitable, despite not being the most popular, as it sounds more “grand” than the rest.	No, the audio plays upon successfully completing a lesson. At this stage there isn’t much to distract the learners from to begin with.	I think using the device’s built-in means (volume buttons, etc.) are the best way to handle this issue. The above option allows the user to change their choice as need be, without having to go out of their way from the exploration player (which would be the case if the option to toggle the audio is present in, say, the preferences page).

How will you trigger the confetti animation?

- ➔ Each piece of confetti will have to be triggered individually; there will be 14 different keyframes rules and animations, shared between the 17 pieces.
- ➔ The animation class will be of the following form:

```
.animation-class-n {
  animation: confetti-anim-n 3000ms cubic-bezier(0, 0.7, 0, 1);
}
```

- ➔ The keyframes rules for each animation will be similar to the following example, just with different arguments in the *translate* and *rotate* functions:

```
@keyframes confetti-anim-1 {
  0% { opacity: 1; }
  50% { opacity: 0.7; }
  100% { transform: translate(-350px, 40px) rotate(9deg); }
}
```

Please visit the link to the codepen demo of the end exploration celebration event (in the key user-stories table of the WHAT section) to view the complete set of CSS keyframes rules for the confetti pieces.

- ➔ A property will be associated with the *ngClass* directive on each individual piece of confetti, and when set to true, each piece will get its corresponding animation class applied to itself, thus triggering the required animation.
- ➔ The property will then be flipped back to false inside a setTimeout() call after the animation has finished, thus removing the animation class.

<u>Accessibility concern</u> – How will these animations be displayed to users who 'prefer-reduced-motion'?

- ➔ The *prefers-reduced-motion* media feature will be used to determine whether to display the full-fledged animations (the check-mark and the confetti animations both) to regular users, or a toned-down version (described underneath the next question) of them to users who have their settings set to reduce on-screen motion/animations.

How will the toned-down animations play out?

- ➔ The confetti will not be displayed to the user (Although this is subjective, my idea behind the confetti is that it adds a bit of 'SURPRISE!' effect to the celebration event due to its rapid and abrupt entry, and calm exit. Due to this very idea it is unsuitable to be displayed to users sensitive to animations. Thus, I believe such an animation should be hidden on the screens of such users).
- ➔ Each of the check-mark SVGs will fade-in instead of filling into their respective positions (this will be achieved by tweaking the *opacity* property instead of the *stroke-dashoffset* property when the *prefers-reduced-motion* media query is active).

Is there a "Don't show me again" button for turning off the animations? Why/why not?

- ➔ No, there is no such button. These animations will, however, be skippable on a per-animation basis by tapping/clicking on the screen when the animation is in progress (will be made possible using the hostlistener decorator).
- ➔ Turning off these animations is not a particularly useful option, as they are simply a part of the experience. Moreover, re-enabling them would require the learner to go out of their way (to say, the user preferences settings page) and look for the required option. In contrast, this issue is not existent for the sign-up section (which also has a "Don't show me again" button) as there is an ever-present sign-up button on the top-nav bar.

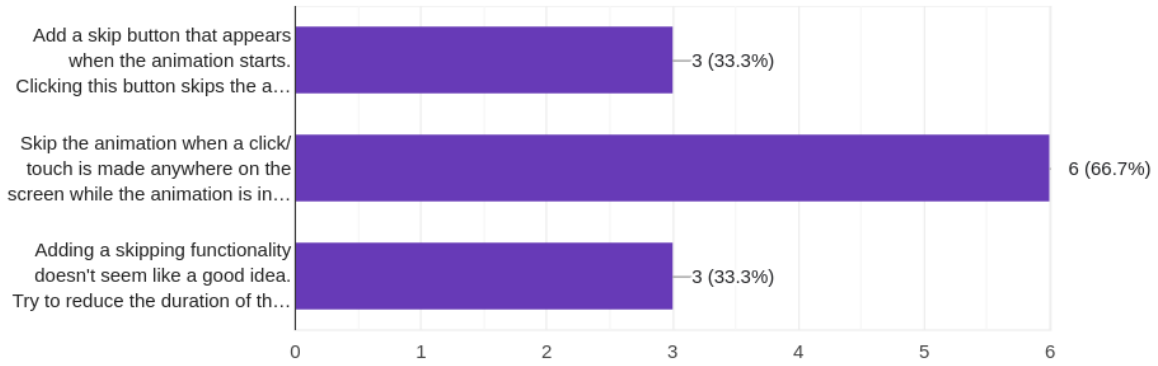
What other alternatives were considered to skip the check-mark animation?

- ➔ The table below summarizes the options:

	Adding a skip button which skips the	**Skipping the animation in response to a click/touch**	**Reducing the duration of the**

	animation	**anywhere on the page**	**animation**
User effort	Allows for the user to skip the animation when they wish to, but clicking a small dedicated button requires considerable user effort.	Allows for the user to skip the animation when they wish to, and also requires very little user effort compared to the previous option.	Requires no effort on the user's part as the animation disappears on its own.
How the animation plays out	Allows the animation to be played out in a leisurely manner, so it doesn't seem jarring for the user.	Allows the animation to be played out in a leisurely manner, so it doesn't seem jarring for the user.	The animation would seem too rushed/jarring to users.
Repetitiveness	The ability to skip the animation makes it less repetitive for the user, but the act of clicking a small button at the end of each chapter itself may get repetitive.	The ability to skip the animation makes it less repetitive for the user, and this is achieved with a simple tap/click on the screen.	The animation is always played out in full and may become repetitive soon.

In addition to the table, the option to skip on touch/click was the most popular among the people I presented this issue to as part of a short survey:



What happens to the rest of the page content during the animation?

- ➔ The following elements will be hidden during the animation period:
  - ◆ **The end exploration card content** - This section will have its
    - *transition-property* set to *transform*, *opacity*
    - *opacity* set to 0
    - *position* set to fixed (to keep it out of the normal flow of the page)
    - and *transform* set to translate(0px, 10px) (the last three will be applied via a single class, say, ***.hide-content,*** via the *ngClass* directive)

- Once the check mark animation is over ***.hide-content*** will be removed from the card content, thus the *opacity* of the card content will be reverted back to 1, the *transform* will be removed and the *position* will be reverted back to relative.
- This will cause the card content to slide up and fade into place.
- **The sign up/log-in section** - This section will have its
  - *display* property set to none to hide it during the animation period.
  - It will be reverted back to the default value once the check mark animation is over.

Milestone message

The section is concerned with the conditional milestone message which will be displayed to the learner on the end exploration screen if the learner completes their nth-ever chapter, where n could be one of the following: 1, 5, 10, 25, 50.

INTENDED DESIGN

Component overview - NONE

➔ This will NOT be implemented as a separate component. It will be a part of the *conversation-skin* component.

How will you determine when to display this message (i.e. how will you determine the value of ‘n’)?

➔ The primary piece of information required for this is the number of chapters completed by the user (which I have been referring to as ‘n’ throughout the document)..

➔ This can be achieved using ***learnercompletedchapterscounthandler/,*** which returns the number of chapters completed by the learner (like so):

```
class LearnerCompletedChaptersCountHandler(base.BaseHandler):
    """Provides the number of chapters completed by the user."""

    GET_HANDLER_ERROR_RETURN_TYPE = feconf.HANDLER_TYPE_JSON
    URL_PATH_ARGS_SCHEMAS = {}
    HANDLER_ARGS_SCHEMAS = {'GET': {}}

    @acl_decorators.can_access_learner_dashboard
```

```
   def get(self):
       """Handles GET requests."""
       (
           learner_progress_in_topics_and_stories,
           number_of_nonexistent_topics_and_stories) = (
               learner_progress_services.get_topics_and_stories_progress(
                   self.user_id))

       all_topic_summary_dicts = (
           learner_progress_services.get_displayable_topic_summary_dicts(
               self.user_id,

learner_progress_in_topics_and_stories.all_topic_summaries))

       completed_chapters_count = 0
       for topic in all_topic_summary_dicts:
           for story in topic['canonical_story_summary_dict']:
               completed_chapters_count +=
len(story['completed_node_titles'])

       self.render_json({
           'completed_chapters_count': completed_chapters_count,
       })
```

- ➔ The response object returned by the call contains a single property – *completed_chapters_count*.
- ➔ A call to this endpoint will be made in the onInit hook of the *conversation-skin* component, and a flag will be set to true if *completed_chapters_count* is equal to one of the following values: 0, 4, 9, 24, 49 (and (value + 1) will be stored in a variable for the next step).
- ➔ Once the end of the lesson is reached, the milestone message will be shown if the above flag is found to have been set to true, and the displayed value will be the same as what was stored above.

How will you display a star next to the message?

- ➔ The basic star from material-icons will be downloaded as an SVG and used for this purpose (an SVG will allow me a greater degree of customizability as opposed to an icon-code).

Recommendations section after chapter completion

The section is concerned with the recommendations section which would appear at the end of a chapter.

Component overview - End chapter recommendations component

- ➔ This section will be implemented as a new component - *end-chapter-recommendations*.
- ➔ It will be placed inside the *ratings-and-recommendations* component.
- ➔ It will receive the following properties from its parent:
  - ◆ The *getExplorationLink()* function's return value. This will be used to guide the learner on to the next chapter if the learner chooses to.
  - ◆ The *storyNodeIdToAdd* property. This will be used to identify the next story node from the list of storyNodes which will be obtained via the story-viewer-backend-api-service.

    Note: The above two properties are actually present in the *conversation-skin* component, NOT the *ratings-and-recommendations* component. These properties will first be passed down from the *conversation-skin* component to the *ratings-and-recommendations* component, which will then pass them down to the *end-chapter-recommendations* component where they are needed.

INTENDED DESIGN

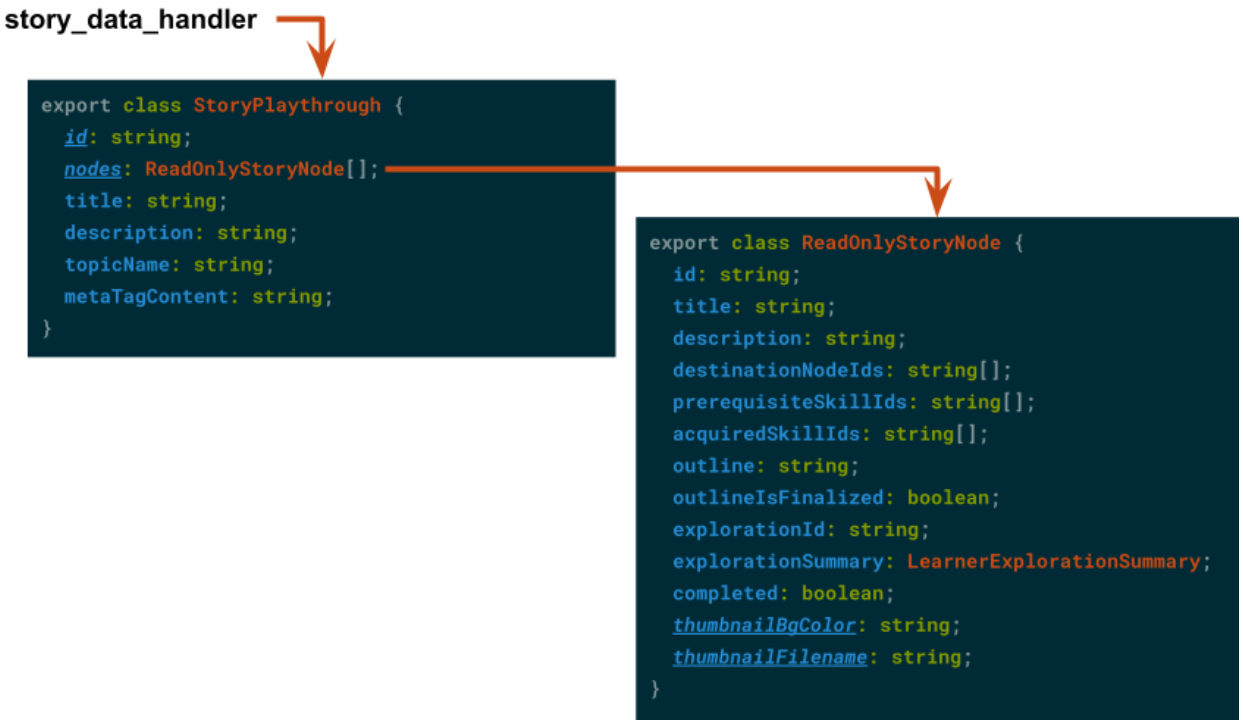
NEXT CHAPTER CARD

How will you obtain the details of the next chapter (in order to display the next chapter card) if it exists?

- ➔ **URL to the next chapter**: Can be obtained via the *getExplorationLink()* function, as stated above.
- ➔ The rest of the chapter details – the chapter image, the chapter bg color, the chapter title – will be obtained via the *story-viewer-backend-api* service's *fetchStoryDataAsync()* function, which makes use of the ***story_data_handler*** endpoint.
- ➔ The function accepts *topicUrlFragment*, *classroomUrlFragment*, and the *storyUrlFragment* strings as input.
- ➔ These three values can be obtained via the urlService's *getUrlParams()* method:

```
let topicUrlFragment = this.urlService.getUrlParams().topic_url_fragment;
let classroomUrlFragment =
  this.urlService.getUrlParams().classroom_url_fragment;
let storyUrlFragment = this.urlService.getUrlParams().story_url_fragment;
```

- ➔ The response object will be an object of the class *StoryPlaythrough*. The properties of this object that we are concerned with are *id* (a string) and *nodes* (an array) – both underlined.
- ➔ Within the nodes array itself, the three properties we are concerned with are *title, thumbnailBgColor* and *thumbnailFileName* (explanation following the diagram):



- ➔ The required storyNode can be obtained by comparing the *id* property of each *ReadOnlyStoryNode* object with the *storyNodeToAdd* property that the component received as input (and selecting the node where they both are equal).

- ➔ Once the storyNode is identified, we can proceed to obtain the following information:
  - ◆ **Chapter title:** The *title* property of the storynode.
  - ◆ **Chapter image background color**: This can be obtained by the *thumbnailBgColor* property, which itself is a color value in hexadecimal format.
  - ◆ **Chapter image**: The required image URL will be obtained using the *thumbnailFileName* property and the *getThumbnailUrlForPreview()* method of the *assetsBackendApiService*, like so:

```
iconUrl = this.assetsBackendApiService.getThumbnailUrlForPreview(
  AppConstants.ENTITY_TYPE.STORY, id, thumbnailFilename);
```

where *id* is the storyid (i.e. the *id* property of the *StoryPlaythrough* object).

PRACTICE-SESSION CARD

How will you obtain a link to the corresponding practice tab?

- ➔ The practice tab for a topic is located at the following URL: `/learn/{{classroom_url_fragment}}/{{topic_url_fragment}}/practice`
- ➔ *classroom_url_fragment* and *topic_url_fragment* will be obtained using the *urlParamsService()*, similar to the way they were obtained in the previous answer.
- ➔ Finally, the required URL will be obtained using the *urlInterpolationService*:

```
this.practiceTabUrl = this.urlInterpolationService.interpolateUrl(
  TOPIC_VIEWER_PAGE, {
    topic_url_fragment: topicUrlFragment,
    classroom_url_fragment: classroomUrlFragment,
  }) + '/practice';
```

- ➔ This URL will then be set equal to the href attribute on the practice session-card link.

How will you obtain the image to be displayed on the practice-session card?

- ➔ The image is already an existing asset in the Oppia codebase, and is used to represent the practice tab on the *topic-viewer-page*.
- ➔ The image will be obtained via the *getStaticImageUrl()* method of the *urlInterpolationService*:

```
<img alt="practice icon"
[src]="getStaticImageUrl('/icons/train_icon_24px.svg')">
```

- ➔ Finally, the cards will have their border-radius set to 20px, and the title section which lies over the chapter image will be set to 85% opacity.
- ➔ The cards' container will have its *display* property set to flex, and *justify-content* set to space-around.

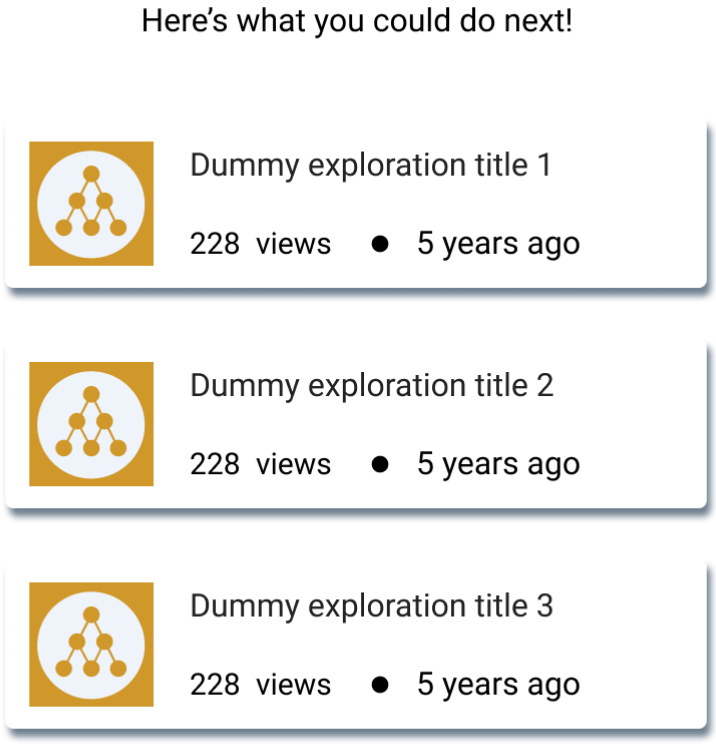
Revamped log-in/sign up section and exploration recommendations

The section is concerned with revamping the log-in/sign up section that currently appears at the end of a chapter for logged-out users and the exploration recommendations that appear for logged-in users at the end of an exploration.

Component overview - ratings-and-recommendations-component

- ➔ The log-in/sign up section and exploration recommendation section are both parts of a bigger component - the *ratings-and-recommendations* component. This component is currently present in the conversation-skin component, *outside* of the card container div.
- ➔ The *ratings-and-recommendations* component's template will be moved inside the tutor-card container div, and its elements will then be modified to better fit the card-container, i.e. modified to fit within lesser width – these parts of the component are discussed in detail below –
- ➔ **Exploration rating section:** This part does not need any adjustment – the reduced width will simply cause the single line of text to flow into the next line, the rating stars will fit perfectly fine in the card-container.
- ➔ **Exploration recommendation section:** The standard-sized exploration recommendation cards will be replaced with much more compact cards, so that they fit inside the card-container div.
    - ◆ The "compact cards" will be obtained by making use of the fact that if a *[mobileCutoffPx]* value is passed to the card component as input, the mobile version of the card (i.e. the "compact card") will be displayed when the screen width falls below that value (this mechanism is already in place – #14527).
    - ◆ The idea is to pass a very high value (5000) as *[mobileCutoffPx]*, so that the mobile version of the card is displayed at all times.
    - ◆ Finally, *flex-direction* will be set to column and *align-items* will be set to center to ensure the cards line-up perfectly.

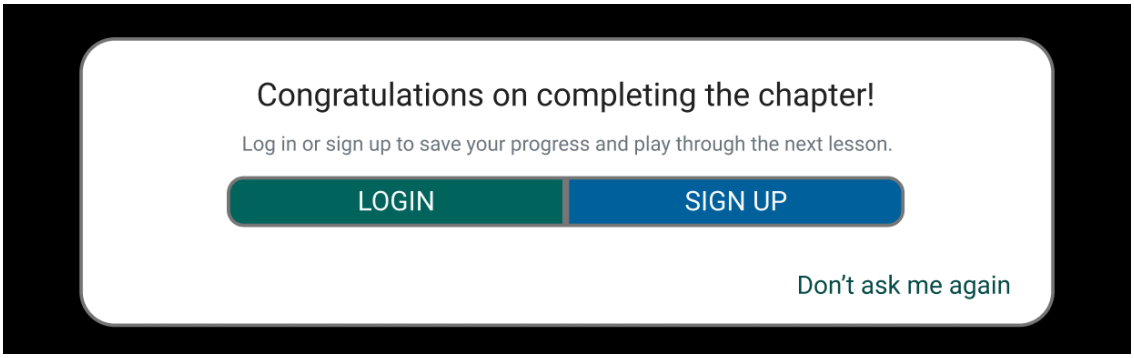
    INTENDED DESIGN



➔ **<u>Sign up/login section</u>:** Similar to the exploration rating section, this section will also adjust itself perfectly fine with no real changes needed as far as fitting inside the card-container is concerned. The following changes will, however, still be made:

- ◆ The login button's both left corners and the signup button's both right corners will be rounded out via the *border-radius* property.
- ◆ This section will be enclosed inside its own container, which will be given rounded edges and a box-shadow to indicate to the learner that the section is something that is embedded into the page, and not something that is a natural part of it.
- ◆ The above point is to ensure that the learner recognizes what will be dismissed when they hit the "Don't show me again" button, which will dismiss the sign-up section for the learner and ensure it doesn't show up again.

<u>INTENDED DESIGN</u>



- ➔ **Conditionally displayed navigation links – RETURN_TO_STORY, RETURN_TO_LIBRARY and RETURN_TO_COLLECTION:**
  - ◆ These links don't need any adjustment with regards to fitting inside the smaller space. They will be able to fit into the card-container correctly just the way they are.
  - ◆ Additionally, each of these links will be enclosed in a small rounded div, making them appear as buttons instead of stray pieces of text (as they currently do).

How will you store the learner's preference of hiding the sign up/login section, and ensure that the sign-up section won't have to be dismissed every time a logged-out learner makes it to the end of a chapter?

- ➔ Once the non-logged-in learner clicks "Don't show me again", their preference will be stored in their browser's localStorage.
- ➔ The function for setting and retrieving the learner's preference will look as follows:

```
disableSignUpSection(): void {
  this.windowRef.nativeWindow.localStorage.setItem(
    'hide_sign_up_section', 'true');
}

shouldSignUpSectionAppear(): boolean {
  let value = this.windowRef.nativeWindow.localStorage.getItem(
    'hide_sign_up_section');
  if (value) {
    return false;
  }
  return true;
}
```

Miscellaneous

Internationalization

Internationalization is carried out via unique I18N keys for each piece of text to be internationalized. Translations may then be obtained in real time via:

- ➔ The translate pipe (in the HTML)
- ➔ The translate service's *instant()* method.

Where are the i18n keys supposed to go?

- ➔ They are supposed to go into the *oppia/assets/i18n* folder. New keys will be created inside en.json and qqq.json for strings that require i18n.

How will you obtain the translation?

- ➔ In case of the HTML translate pipe, the translation can be obtained by appending the i18n key with the following string: '| translate'.
- ➔ In case of the translate service, the service first needs to be injected into the component, following which the key needs to be passed into the *instant()* method of the service. This will return the translated string (if it exists).
- ➔ Additionally, in case of the translate service, we may also need to subscribe to the onLangChange event emitter, so that translations are updated whenever the selected language changes.

Which strings will be internationalized?

- ➔ The list of strings for which i18n keys will be created (if they don't already exist) includes:
  - ◆ All the checkpoint messages and titles
  - ◆ Milestone messages (for all 5 values of 'n')
  - ◆ Progress-text and button-text in the checkpoint message div
  - ◆ Text in the recommendations section ("Here's what you could do next!", etc.)
  - ◆ Learner facing text in general

Third-Party Libraries

None required.

"Service" Dependencies

None.

Impact on Other Oppia Teams

The best way for Oppia's lesson creators to harness the benefits of this project would be to ensure checkpoints are placed in meaningful locations throughout a chapter.

Key High-Level and Architectural Decisions

Decision 1: Using standard CSS animations instead of Angular animations

An alternative is to use the latter instead.

I believe using CSS animations is a better approach because of the reasons listed in the table below:

	Alternative 1: CSS animations	**Alternative 2: Angular animations**
Maintainability and future development	(Newer) Contributors are *far* more likely to be familiar with animating via CSS, as compared to Angular animations.	Contributors are not as likely to be familiar with Angular animations and this may act as a hurdle for someone working on/around the code corresponding to this project, especially if they are relatively new to Angular.
Easier control (via the component class)	CSS animations cannot be manipulated via the component class (ignoring cases like manipulating the template via angular's structural directives, etc).	Angular animations offer more control over the animations in a template via its corresponding component class. *This is, however, irrelevant due to how many animations are actually SVG elements with their transition-property set to stroke-dashoffset, while the rest of the animations are incredibly simple, making them just as easy (if not more) to be implemented via standard CSS.*
Performance	Operations that operate via the compositor-thread, without interfering with the main thread (as opposed to paint and layout operations, which do), do not hinder the page's performance. Two properties which are handled by the compositor alone are transform and opacity (**Source**: The beginning of this Google developers article discussing performance hits caused	*"Angular's animation system is built on CSS functionality…"* **Source**: Angular documentation *"Angular's animation system lets you build animations that run with the same kind of native performance found in pure CSS animations."*

	by animations) – which are the primary means of animation I will be using throughout the project. The only instance where I will use a layout property (which may have a slightly greater performance hit than the above two properties) is my use of the display property which allows for the skipping of the check-mark animation. This is not concerning as it will be the sole layout property change that will occur at that moment, and will be instantaneous. Not to mention it is optional and will not always take place.	**Source**: Archived Angular 2 documentation Given how Angular animations are built on native CSS, there is bound to be very little difference in performance between the two.

Risks and mitigations

No potential risks.

Implementation Approach

[Web only] Storage Model Layer Changes

None.

Domain Objects

None.

User Flows (Controllers and Services)

Existing controllers:

➔ New handler: **LearnerCompletedChaptersCountHandler.** Will return the number of chapters completed by the learner.

[Web only] Web frontend changes

Checkpoint celebration:

➔ *checkpoint-celebration.component.ts*: New component for the checkpoint message (both the full-scale div and the minified tooltip-style message).

- ◆ It will receive *isLoggedIn* (boolean)*, states* (object), and *initStateName* (string) as input.
- ◆ On initialization, it will call upon the *checkpoint-celebration-utility* service to obtain a list of checkpointed states and subscribe to the *onPlayerStateChange* event emitter (from the exploration-player-state service) to determine when a checkpoint has been encountered.
- ◆ It will obtain checkpoint messages and titles from the *checkpoint-celebration-utility* service whenever a checkpoint is encountered, and trigger the animation.

➔ *checkpoint-celebration-utility.service.ts*: New helper service meant for use by the *checkpoint-celebration* component.

- ◆ It will contain a total of three functions:
  - One for obtaining an ordered checkpoint list (via a BFS) for which it will import the compute-graph service,
  - One for obtaining the I18N key of a checkpoint message, meant to loosely indicate the progress made by a learner,
  - One for obtaining the I18N key of a checkpoint title.

End chapter celebration:

➔ *end-chapter-check-mark.component.ts*: New component meant to carry out the check mark SVG animation.

➔ *end-chapter-confetti.component.ts*: New component meant to carry out the confetti spray animation (and play celebratory audio accompanying it).

Post chapter recommendations section:

➔ *end-chapter-recommendations.component.ts*: New component meant to display recommendations to the learner when they make it to the end of a chapter.

- ◆ It will receive the next chapter URL and the node ID of the next storyNode (chapter).
- ◆ It will call the **story_data_handler** endpoint to receive chapter information such as thumbnail, bg color, etc.
- ◆ It will use the urlInterpolationService to generate a link to the topic's corresponding practice page.

Existing components:

➔ *conversation-skin.component.ts* and components within it: Existing component, will be modified in the following ways:

- ◆ Will be modified to call the **explorehandler** endpoint to obtain the *states* dictionary and the *initStateName* property, to be passed down to the *checkpoint-celebration* component.
- ◆ Will be modified to call the **learnercompletedchapterscounthandler/** endpoint to obtain the number of completed chapters.

- Will be modified to include a new, small section to be shown on the terminal state of a chapter if the learner completed their 1st, 5th, 10th, 25th, or 50th chapter.
- Will be modified to include the ratings-and-recommendations component within the card-container region, when on the terminal state.
- Will be modified to be able to access two of its children components' – *end-chapter-check-mark* and *end-chapter-confetti* – methods via the use of the ViewChild decorator.

➔ *ratings-and-recommendations.component.ts*: Existing component, will be modified in the following ways:

- Will be modified to display recommended explorations in the form of compact cards instead.
- Will be modified to now include the *end-chapter-recommendations* component.
- Will be modified to now store the learner's sign-up section preference in the localStorage, and retrieve the same as and when needed.

Existing backend API service:

➔ *learner-dashboard-backend-api.service.ts*: A new function will be added to make HTTP calls to the **learnercompletedchapterscounthandler/** endpoint.

Documentation changes

None required as the development workflow will remain unaffected.

Testing Plan

E2e testing plan

#	Test name	Initial setup step	Step	Expectation
1.	End chapter recommendations test	A topic with a story is generated, which contains at least two chapters.	Head over to the story-viewer page, and start the first chapter.	The chapter begins and the first state is loaded into the conversation-skin.
			Complete the chapter.	The end-exploration screen is loaded, which contains a section displaying the next chapter and practice session as recommendation cards.
			Click on the recommendation cards.	The corresponding destination is loaded upon clicking on the card.
2.	Checkpoint celebration test	An chapter with two checkpoints is generated	Start the newly created chapter.	The chapter begins and the first state is loaded into the conversation-skin.
			Make it to the checkpoint.	The checkpoint message appears on-screen.

Feature testing

Does this feature include non-trivial user-facing changes? **YES**

Implementation Plan

The implementation plan consists of two milestones.

Milestone 1 objective: Celebrate lesson completion in a meaningful, actionable, and rewarding way for the learner.

Milestone 1 schedule:

No.	**Description of PR / action**	**Prereq PR numbers**	**Target date for PR creation**	**Target date for PR to be merged**
1	Add end-chapter completion check-mark and confetti (along with the celebratory audio meant to be played alongside the confetti animation)	-	15 June (Add a feature gating flag before getting started)	19 June
2	Revamp the end-exploration recommendations and login/sign-up section (the *ratings-and-recommendations* component)	1	21 June	25 June
3	Add post-chapter recommendations section and milestone message (alongwith the required endpoint)	1, 2	30 June	5 July
4	Add E2E tests (milestone 1)	1, 2, 3	10 July (Put milestone 1 up for feature review)	14 July
5	Bug fixes pertaining to milestone 1, if required	1, 2, 3, 4	18 July	21 July

Milestone 2 objective: Celebrate checkpoint completion in a meaningful, rewarding way for the learner that encourages them to continue progressing.

Milestone 2 schedule:

No.	Description of PR / action	Prereq PR numbers	Target date for PR creation	Target date for PR to be merged
6	Add the checkpoint celebration utility service	-	23 July (Add a feature gating flag before getting started)	25 July
7	Add checkpoint celebration messages	6	01 August	06 August
8	Integrate checkpoint messages with the lesson-information-modal component	6, 7	08 August	11 August
9	Add E2E tests (milestone 2)	6, 7, 8	16 August (Put milestone 2 up for feature review)	20 August
10	Bug fixes pertaining to milestone 2, if required	6, 7, 8, 9	24 August	28 August

Future Work

- ➔ Look into more ways of “gamifying” the experience of playing through an exploration.
- ➔ Monitor the feedback on the checkpoint messages feature. Determine what kind of impact they’re having on a learner’s experience. Determine whether they could be modified in any way (for example, by making the checkpoint messages configurable) to potentially improve the experience.
- ➔ Continue my work with the LaCE quality team.