Tutorial: Learn how to figure out the reproduction steps for a server error

Introduction

This tutorial will guide you through debugging a server error that is challenging to reproduce locally. Specifically, we will investigate and fix a `TypeError` related to certificate generation for contributors.

Skills Covered:

- Codebase Navigation
- Identifying and Analyzing Error Logs
- Debugging Techniques
- Reproducing Server Errors Locally

Scenario:

One of the server admins has reported the following error logs. Your task is to investigate the issue and determine how and why it is occurring.

Note: The primary goal of this tutorial is not to find a solution, but to guide you through the process of investigating and understanding the workflow of debugging server errors. In this tutorial, you will follow the steps a developer might take to investigate this server error.

```
TypeError: expected string or bytes-like object

Exception raised: expected string or bytes-like object
Traceback (most recent call last):
  File
"/layers/google.python.pip/pip/lib/python3.8/site-packages/webapp2.py",
line 604, in dispatch
    return method(*args, **kwargs)
  File "/workspace/core/controllers/acl_decorators.py", line 4788, in
test_can_fetch_all_contributor_dashboard_stats
    return handler(self, username, **kwargs)
  File "/workspace/core/controllers/contributor_dashboard.py", line 1062,
in get
    response = suggestion_services.generate_contributor_certificate_data(
  File "/workspace/core/domain/suggestion_services.py", line 3870, in
```

```
generate_contributor_certificate_data
    data = _generate_translation_contributor_certificate_data(
  File "/workspace/core/domain/suggestion_services.py", line 3945, in
_generate_translation_contributor_certificate_data
    plain_text = _get_plain_text_from_html_content_string(
  File "/workspace/core/domain/suggestion_services.py", line 1408, in
_get_plain_text_from_html_content_string
    html_content_string_with_rte_tags_replaced = re.sub(
  File "/layers/google.python.runtime/python/lib/python3.8/re.py", line
210, in sub
    return _compile(pattern, flags).sub(repl, string, count)
TypeError: expected string or bytes-like object
```

Procedure:

The following steps illustrate how a developer might tackle this issue. Try following this tutorial step-by-step on your local machine! This will give you a better sense of how to tackle other similar issues in the codebase. If you get stuck with a step in this tutorial, raise an issue in GitHub Discussions to get help.

***Important**: When you see a "practice question box", stop and try to figure out the answer on your own before reading ahead. You will learn more if you try to figure out your own answer to the question first!*

Setup:

1. **Install Oppia on Your Local Machine:** To begin, you'll need to have Oppia installed on your local machine. If you haven't done so already, please follow the installation steps provided in this wiki page.
2. **Check Out the Specific Commit:** To ensure that you are working with the same version of the code as this tutorial, navigate to your local Oppia directory and check out the specific commit:

   ```
   git checkout 192f0a9a4866debac160015bc949130aaae6a7fe
   ```

   This ensures consistency between your environment and the code referenced in this tutorial.

3. **Verify the Commit:** You can verify that you are on the correct commit by running:

   ```
   git log -1
   ```

The output should display the commit ID `192f0a9a4866debac160015bc949130aaae6a7fe`.

Debugging Process:

When faced with a server error, developers at Oppia typically follow these steps:

1. **Analyze the Error Logs to Locate the Affected Code:** Start by reviewing the error logs to find a stack trace that indicates where the error occurred. Pinpoint the relevant file, function, or line number where the problem originated.
2. **Examine the Affected Code:** Open the identified file(s) and examine the specific code blocks related to the error. Understand the intended functionality of the code and check for any immediate errors or inconsistencies.
3. **Investigate Potential Causes by Exploring the Code in Depth:** Consider possible reasons for the error, such as incorrect data types, missing conditions, edge cases, or unexpected inputs. Dive deeper into the code, focusing on sections that are most likely causing the issue. Look for logic errors, unhandled cases, or data processing problems that align with your initial suspicions.
4. **Reproduce the Error:** Set up an environment to recreate the conditions that led to the error. Use test data or modify unit tests to replicate the issue and confirm your hypotheses about its cause.
5. **Document Your Findings:** Once you've identified and confirmed the cause of the error, document your findings in detail on the original GitHub issue. Provide a summary of the error, clear steps to reproduce it, and any relevant observations or logs to support your conclusions.

Stage 1: Analyze the Error Logs to Locate the Affected Code

Objective: *Identify where the error occurred in the code.*

Note: *This tutorial focuses on server errors, which are typically reported by server admins and come with error logs. For other issues, such as user-reported bugs, error logs might not always be available. In these cases, it is essential to ask users for "steps to reproduce the bug." If this information is not provided, we can contact the user to request additional details that will help us understand the issue better.*

Here is the error log we need to investigate:

```
TypeError: expected string or bytes-like object

Exception raised: expected string or bytes-like object
Traceback (most recent call last):
  File
"/layers/google.python.pip/pip/lib/python3.8/site-packages/webapp2.py",
line 604, in dispatch
```

```
    return method(*args, **kwargs)
  File "/workspace/core/controllers/acl_decorators.py", line 4788, in
test_can_fetch_all_contributor_dashboard_stats
    return handler(self, username, **kwargs)
  File "/workspace/core/controllers/contributor_dashboard.py", line 1062,
in get
    response = suggestion_services.generate_contributor_certificate_data(
  File "/workspace/core/domain/suggestion_services.py", line 3870, in
generate_contributor_certificate_data
    data = _generate_translation_contributor_certificate_data(
  File "/workspace/core/domain/suggestion_services.py", line 3945, in
_generate_translation_contributor_certificate_data
    plain_text = _get_plain_text_from_html_content_string(
  File "/workspace/core/domain/suggestion_services.py", line 1408, in
_get_plain_text_from_html_content_string
    html_content_string_with_rte_tags_replaced = re.sub(
  File "/layers/google.python.runtime/python/lib/python3.8/re.py", line
210, in sub
    return _compile(pattern, flags).sub(repl, string, count)
TypeError: expected string or bytes-like object
```

Understanding the Stack Trace:

- A stack trace is a report of the active stack frames at a certain point in time during the execution of a program. It shows the call sequence leading to the point where the error occurred. Each line provides the file name, line number, and function where the error occurred.
- The stack trace is read from the bottom up to understand the sequence of function calls leading to the error.
- The bottom-most lines generally point to standard library or third-party code (in this case, `re.py` from Python's standard library). Moving upward, you start seeing calls within the project codebase, which is where we should focus. The key is to find the first instance where **Oppia's code** interacts with the point of failure.

Practice 1: Locate the specific line in the stack trace where Oppia's code is directly involved in causing the failure. This is the point where the error is most likely to originate within the Oppia codebase.

Hint: Look for the first occurrence (from the bottom of the stack trace upward) where the stack trace shows a line of code from the Oppia codebase. This line represents the entry point in the Oppia code that led to the failure. Identifying this line will help you trace the error back to its origin in the code.

In this stack trace, the relevant line is:

```
File "/workspace/core/domain/suggestion_services.py", line 1408, in
_get_plain_text_from_html_content_string
html_content_string_with_rte_tags_replaced = re.sub(
```

This line indicates that the error originated in the `_get_plain_text_from_html_content_string` function within the `suggestion_services.py` file.

You have identified the error location in the codebase and gathered initial clues about what might have gone wrong.

Next Step is to pinpoint the Specific Part of the Code Causing the Error.

Practice 2: Locate the `_get_plain_text_from_html_content_string` function in the `suggestion_services.py` file.

Take a moment to locate it before moving forward.

Once you've located the function, review the surrounding code to understand what the function does and how the error might have been triggered.

Hint: To locate the function, you can make use of search functionalities of your code editor. (https://github.com/oppia/oppia/wiki/Tips-for-common-IDEs)

```
File "/workspace/core/domain/suggestion_services.py", line 1408, in
_get_plain_text_from_html_content_string
    html_content_string_with_rte_tags_replaced = re.sub(
File "/layers/google.python.runtime/python/lib/python3.8/re.py", line 210,
in sub
    return _compile(pattern, flags).sub(repl, string, count)
TypeError: expected string or bytes-like object
```

Based on the error logs, the issue arises because the argument passed to the `re.sub()` function in the `_get_plain_text_from_html_content_string` method is not of the expected data type (a string or bytes-like object).

You have now pinpointed the exact file, function, and line where the issue originated and are ready to dive deeper into the problem area.

Stage 2: Examine the Affected Code

Examine the `_get_plain_text_from_html_content_string` function to understand its purpose and the specific operation where the error occurs. Here's the function:

```
`def _get_plain_text_from_html_content_string(html_content_string: str) ->
str:
    """Retrieves the plain text from the given html content string. RTE
element
    occurrences in the html are replaced by their corresponding rte
component
    name, capitalized in square brackets.
    eg: <p>Sample1 <oppia-noninteractive-math></oppia-noninteractive-math>
        Sample2 </p> will give as output: Sample1 [Math] Sample2.
    Note: similar logic exists in the frontend in
format-rte-preview.filter.ts.

    Args:
        html_content_string: str. The content html string to convert to
plain
            text.

    Returns:
        str. The plain text string from the given html content string.
    """

    def _replace_rte_tag(rte_tag: Match[str]) -> str:
        """Replaces all of the <oppia-noninteractive-**> tags with their
        corresponding rte component name in square brackets.

        Args:
            rte_tag: MatchObject. A matched object that contains the
                oppia-noninteractive rte tags.

        Returns:
            str. The string to replace the rte tags with.
        """
        # Retrieve the matched string from the MatchObject.
        rte_tag_string = rte_tag.group(0)
        # Get the name of the rte tag. The hyphen is there as an optional
        # matching character to cover the case where the name of the rte
        # component is more than one word.
```

```
1387        rte_tag_name = re.search(
                r'oppia-noninteractive-(\w|-)+', rte_tag_string)
1388        # Here, rte_tag_name is always going to exists because the string
            # that was passed in this function is always going to contain
1389        # `<oppia-noninteractive>` substring. So, to just rule out the
1390        # possibility of None for mypy type checking. we used assertion
    here.
1391        assert rte_tag_name is not None
            # Retrieve the matched string from the MatchObject.
1392        rte_tag_name_string = rte_tag_name.group(0)
1393        # Get the name of the rte component.
            rte_component_name_string_list = rte_tag_name_string.split('-')[2:]
1394        # If the component name is more than word, connect the words with
    spaces
1395        # to create a single string.
1396        rte_component_name_string = '
    '.join(rte_component_name_string_list)
1397        # Captialize each word in the string.
1398        capitalized_rte_component_name_string = (
                rte_component_name_string.title())
1399        formatted_rte_component_name_string = ' [%s] ' % (
                capitalized_rte_component_name_string)
1400        return formatted_rte_component_name_string
1401
          # Replace all the <oppia-noninteractive-**> tags with their rte
1402    component
1403       # names capitalized in square brackets.
          html_content_string_with_rte_tags_replaced = re.sub(
1404          r'<oppia-noninteractive-[^>]+>(.*?)</oppia-noninteractive-[^>]+>',
              _replace_rte_tag, html_content_string)
1405       # Get rid of all of the other html tags.
          plain_text = html_cleaner.strip_html_tags(
1406          html_content_string_with_rte_tags_replaced)
          # Remove trailing and leading whitespace and ensure that all words are
1407       # separated by a single space.
1408       plain_text_without_contiguous_whitespace = ' '.join(plain_text.split())
          return plain_text_without_contiguous_whitespace
1409


1410


1411
```

```
1412
1413
1414
1415
```

Practice 3: Locate the lines which are causing the error in the above mentioned function.

Hint: Review the error logs. In a real debugging scenario, the stack trace would help you pinpoint this specific line as the source of the error.

In this stack trace, the relevant line is:

```
File "/workspace/core/domain/suggestion_services.py", line 1408, in
_get_plain_text_from_html_content_string
html_content_string_with_rte_tags_replaced = re.sub(
```

This line indicates that the error originated in the `_get_plain_text_from_html_content_string` function within the `suggestion_services.py` file at line number 1408. Let's take a look at that line:

```
html_content_string_with_rte_tags_replaced = re.sub(
r'<oppia-noninteractive-[^>]+>(.*?)</oppia-noninteractive-[^>]+>',
      _replace_rte_tag, html_content_string)
```

Note that the third parameter, `html_content_string`, is being passed to the `re.sub` function and that this parameter is the argument of the function `_get_plain_text_from_html_content_string`.

The `re.sub` function is used to replace certain HTML tags. According to the Python documentation for re.sub, it expects the first argument to be a pattern, the second argument to be a replacement string, and the third argument to be the string to be searched and replaced.

Even though the `re.sub` documentation doesn't explicitly mention the `TypeError`, we can infer the following:

1. **TypeError Convention**: In Python, `TypeError` is conventionally raised to indicate that an operation or function received an argument of an inappropriate type. This means that if `re.sub` expects a string or bytes-like object but receives a different type, it raises a `TypeError`.

2. **Error Message**: The error message "expected string or bytes-like object" indicates that `re.sub` was given an argument that wasn't a string or bytes-like object, which is why it raised a `TypeError`.

Practice 4: Determine where the `html_content_string` is coming from.

Hint: Review the error logs carefully. Pay close attention to the lines mentioned in the stack trace, especially the one that is calling the _get_plain_text_from_html_content_string function. *This will give you a clue about where the* `html_content_string` *originates.*

According to the error log, this function is called in `suggestion_services.py` within the `_generate_translation_contributor_certificate_data` function.

```
# Retrieve the html content that is emphasized on the
# Contributor Dashboard pages. This content is what stands
# out for each suggestion when a user views a list of
# suggestions.
get_html_representing_suggestion = (
    SUGGESTION_EMPHASIZED_TEXT_GETTER_FUNCTIONS[
        Suggestion.suggestion_type]
    )
plain_text = _get_plain_text_from_html_content_string(
get_html_representing_suggestion(suggestion))
```

As we can see, we are passing the output of the function, get_html_representing_suggestion(suggestion) as the input to the function _get_plain_text_from_html_content_string.

Let's take a closer look at the method `get_html_representing_suggestion(suggestion)`. This method handles different types of suggestions and retrieves the corresponding HTML content based on the type of suggestion.

In the context of this method, two lambda functions are defined to extract HTML content based on the `suggestion_type`:

For translation suggestions:

```
SUGGESTION_TRANSLATE_CONTENT_HTML: Callable[
    [suggestion_registry.SuggestionTranslateContent], str
] = lambda suggestion: suggestion.change_cmd.translation_html
```

This lambda function is used when the suggestion type is related to translating content. It retrieves the `translation_html` property from the `change_cmd` attribute of the `suggestion` object.

For question suggestions:

```
SUGGESTION_ADD_QUESTION_HTML: Callable[
    [suggestion_registry.SuggestionAddQuestion], str
] = lambda suggestion: suggestion.change_cmd.question_dict[
    'question_state_data']['content']['html']
```

This lambda function is used when the suggestion type involves adding a question. It retrieves the `html` property from the `question_state_data` dictionary inside the `question_dict` of the `suggestion` object.

Practice 5: Based on the code provided and the context of the lambda functions, what could potentially go wrong with the data being accessed? Specifically, think about the structure of the `suggestion.change_cmd.question_dict['question_state_data']['content']['html']` and `suggestion.change_cmd.translation_html` fields. What potential data type or structure issues could arise here?

One possible issue is that the `html` property of the `question_dict` or the `translation_html` field might not always be a string as expected. It could be `None` or another data type entirely, which would cause issues when the `_get_plain_text_from_html_content_string` function tries to process it. If the function is expecting a string but receives a different type (such as `None` or a more complex object), it could result in a `TypeError`. This could explain why the error indicates that a string or bytes-like object was expected but something else was provided.

In this case, the issue is likely that either `suggestion.change_cmd.translation_html` or `suggestion.change_cmd.question_dict['question_state_data']['content']['html']` is not always a string, leading to the observed error when passed to the `re.sub()` function.

You now understand the purpose and expected behavior of the affected code and have identified areas where discrepancies may have occurred.

Stage 3: Investigate Potential Causes by Exploring the Code in Depth

Objective: *Formulate hypotheses about why the error is occurring based on initial findings, and verify them by examining the code more closely.*

At this point, brainstorm the possible data types that could be causing this issue. Consider the following questions:

- **What Data Types Are Expected?**
  What data types should `suggestion.change_cmd.question_dict['question_state_data']['content']['html']` and `suggestion.change_cmd.translation_html` be? Are they always strings, or can they also be other types?

- **Can Non-String Values Cause Issues?**
  Could these fields sometimes store non-string values, such as lists or None? If so, this could lead to issues when the `_get_plain_text_from_html_content_string` function attempts to process them.

You now have potential causes of the error. To confirm these, you need to dig deeper into the codebase.

Investigating the Source of the Problem

Our initial suspicion is that the problem might be due to the data types involved. To investigate further, we need to:

- **Check the Creation of Suggestions**
  Examine the code responsible for creating suggestions and questions to ensure that the expected data types are being used. This includes verifying that fields are populated correctly and checking for any anomalies that might lead to errors.

Practice 6: Locate the Code Responsible for Creating Suggestions

Hint: Analyze the network requests triggered when creating a suggestion, and then examine the handler attached to the endpoint. Use the resources such as Find the Right Code to Change and Analyzing the Codebase to guide your investigation.

Open the Developer Tools in your browser and go to the "Network" tab while creating a translation suggestion. You'll notice that the endpoint `/suggestionhandler` is triggered.

Next, check the `main.py` file in the Oppia codebase to find which handler is associated with this endpoint. You'll see that the endpoint is connected to:

```
get_redirect_route(
    r'%s/' % feconf.SUGGESTION_URL_PREFIX,
    suggestion.SuggestionHandler),
```

Practice 7: Analyze the `SuggestionHandler` in `oppia/core/controllers/suggestion.py`.
Can you find the code that is responsible for creating suggestions?

Within `oppia/core/controllers/suggestion.py`, you'll see that suggestion creation is managed by this method:

```
suggestion = suggestion_services.create_suggestion(
    suggestion_type,
    self.normalized_payload['target_type'],
    self.normalized_payload['target_id'],
    self.normalized_payload['target_version_at_submission'],
    self.user_id,
    self.normalized_payload['change_cmd'],
    self.normalized_payload['description']
)
```

This shows that the actual creation of the suggestion is handled by the `create_suggestion` method in the `suggestion_services.py` file.

To further investigate, we need to and review the Suggestion Creation Services method to understand how fields like `content_html` are populated.

Practice 8: Analyze the `create_suggestion` method of the `suggestion_services.py` file. Can you find the code where content_html field is being populated/used ?

On reviewing the method, we can see at this line that for suggestions of type 'Translations,' we are getting content_html using this get_content_html and are performing a equality check against change_cmd dict's content_html property. This might be what we are looking for. Let's take a look at this method

```
def get_content_html(
      self, state_name: str, content_id: str
  ) -> Union[str, List[str]]:
      """Return the content for a given content id of a state.

      Args:
          state_name: str. The name of the state.
          content_id: str. The id of the content.
```

```
        Returns:
            str. The html content corresponding to the given content id of a
state.

        Raises:
            ValueError. The given state_name does not exist.
        """
        if state_name not in self.states:
            raise ValueError('State %s does not exist' % state_name)

        return self.states[state_name].get_content_html(content_id)
```

Here, we notice a discrepancy between the type annotation (`-> Union[str, List[str]]`) and the docstring. The docstring states that the method returns a `str`, but the type annotation indicates that it could return either a `str` or a `List[str]`. This is a good example of why you shouldn't rely solely on docstrings to understand a function's behavior. Docstrings can become outdated or inaccurate over time, especially in large codebases like Oppia, where many contributors make changes.

Verify the Return Type by Analyzing the Code

To accurately determine the return type, you need to look at the actual implementation of the `get_content_html` method. Here, the method calls `get_content_html(content_id)` on `self.states[state_name]`. To understand what is being returned, you need to trace this call further into the `self.states[state_name]` object and its `get_content_html` method.

If we look further into this code:

```
return self.states[state_name].get_content_html(content_id)
```

This function eventually returns the value:

```
return content_id_to_translatable_content[content_id].content_value
```

Here, the `content_value` property is populated via:

```
self.content_value = content_value
```

The type of `content_value` is defined as:

```
content_value: feconf.ContentValueType
```

Upon examining `ContentValueType`, we find:

```
# The data type for the translated or translatable content in any
# BaseTranslatableObject.
ContentValueType = Union[str, List[str]]
```

By investigating the return types of each function and following the flow of data through the helper methods, we confirm that the `get_content_html` method can indeed return either a `str` or a `List[str]`. However, from examining the backend functions alone, it's not entirely clear whether the `content_html` variable actually ends up being a `List[str]` or just a `str`.

The evidence we have gathered so far is suggestive, but not definitive. To gather more solid proof, we need to trace where the `content_html` property in the `change_cmd` dictionary is actually populated.

Looking at the `SuggestionHandler`:

```
suggestion = suggestion_services.create_suggestion(
            suggestion_type,
            self.normalized_payload['target_type'],
            self.normalized_payload['target_id'],
            self.normalized_payload['target_version_at_submission'],
            self.user_id,
            self.normalized_payload['change_cmd'],
            self.normalized_payload['description']
        )
```

This snippet indicates that the `content_html` is part of the `change_cmd` dictionary, which is passed from the frontend to the backend. This means the backend isn't strictly defining what the type of `content_html` should be; rather, it's receiving it from the frontend.

To gain more clarity, we need to examine the network calls in the frontend that send data to the `SuggestionHandler`. This can help us trace the source of the `content_html` value.

In the frontend, we find the function `suggestTranslatedTextAsync`, which is responsible for making a call to the handler:

```
async suggestTranslatedTextAsync(
   expId: string,
```

```
    expVersion: string,
    contentId: string,
    stateName: string,
    languageCode: string,
    contentHtml: string | string[],
    translationHtml: string | string[],
    imagesData: ImagesData[],
    dataFormat: string
): Promise<void> {
    const postData: Data = {
      suggestion_type: 'translate_content',
      target_type: 'exploration',
      description: 'Adds translation',
      target_id: expId,
      target_version_at_submission: expVersion,
      change_cmd: {
        cmd: 'add_written_translation',
        content_id: contentId,
        state_name: stateName,
        language_code: languageCode,
        content_html: contentHtml,
        translation_html: translationHtml,
        data_format: dataFormat,
      },
      files: await
this.imageLocalStorageService.getFilenameToBase64MappingAsync(imagesData),
    };
    const body = new FormData();
    body.append('payload', JSON.stringify(postData));
    return this.http.post<void>('/suggestionhandler/', body).toPromise();
}
```

Here, we can see that `content_html` can be either a `string` or a `List[string]` (array of strings). This confirms that the frontend can indeed send a `List[string]` as the value for `content_html`. Thus, the `content_html` property for translation suggestions could indeed be either a single string or a list of strings.

~~This method returns either a string or a list of strings~~.

We have identified the root cause of the error. The `_get_plain_text_from_html_content_string` function expects a `str`.When it receives a list of strings, it results in a `TypeError`.

Note on Why `mypy` Did Not Catch This Error:

`mypy` is a static type checker for Python, designed to enforce type safety by verifying that types declared in type annotations are respected throughout the code. However, `mypy` has limitations when dynamic typing is involved, especially with the use of functions like `setattr` that dynamically set attributes at runtime.

In the `create_suggestion` method, the `change_cmd` parameter is annotated with `Mapping[str, change_domain.AcceptableChangeDictTypes]`. The `AcceptableChangeDictTypes` union type includes a variety of types such as `str`, `bool`, `float`, `int`, `None`, `List[str]`, and several domain-specific dictionary types. This annotation indicates that `change_cmd` can contain any of these types.

Within the `BaseChange` class, the `setattr` function is used to dynamically set attributes based on the contents of the `change_dict`

```
for attribute_name in cmd_attribute_names:
    setattr(self, attribute_name, change_dict.get(attribute_name))
```

Normally, if a variable were declared with the type `AcceptableChangeDictTypes` and later passed to a function expecting a `str`, `mypy` would flag this as a potential error because `AcceptableChangeDictTypes` can also include types like `List[str]`, which are not compatible with a `str` type.

However, in this case, `setattr` is used to assign a value from `change_dict` which could be of any type included in `AcceptableChangeDictTypes` to an attribute that is declared to be of type `str`. Since `setattr` is a built-in function that dynamically assigns values to object attributes, `mypy` cannot enforce type safety at this point. It assumes that the attribute, which is declared as `str`, will always be of type `str`, even though it might actually hold a `List[str]` or another type from `AcceptableChangeDictTypes`.

This is why `mypy` does not catch the type mismatch when a function like `_get_plain_text_from_html_content_string`, which expects a `str`, encounters an attribute that is, in fact, a `List[str]`. The use of `setattr` bypasses `mypy`'s static type checking, leading to potential runtime type errors that `mypy` does not detect.

You have pinpointed the exact cause of the error in the code, understanding why the issue occurs under certain conditions.

Stage 4: Reproduce the Error

Objective: *Confirm the suspected cause of the error by replicating it in a controlled environment.*

Once you have a hypothesis about the root cause of the issue, it's time to verify it. There are several ways you can do this:

Option 1: Reproduce on a Local Server

Try to replicate the error on a local server by following the user journey that leads to the issue. This involves setting up a local environment, creating or modifying an exploration with rule inputs, and attempting to generate a certificate to see if the error occurs again. This method is practical and quick if you can accurately simulate user actions, but it may not always capture the exact conditions of the live environment.
Pros: Practical and quick if you can closely simulate user actions.
Cons: May not always replicate the exact conditions of the live environment.

Option 2: Use Unit Tests

Modify or create unit tests to check if they trigger the same error when using the rule inputs in question. This approach involves locating existing unit tests, such as those for the `generate_contributor_certificate_data` function, and adjusting them to use the problematic inputs. This method can be efficient if you already have a good understanding of the issue, but it relies on having relevant unit tests available or being able to adapt existing ones easily.

Pros: Efficient if you have a clear idea of the issue.
Cons: Requires existing or easily adaptable unit tests.

Option 3: Write a Validation Job

Develop a Beam job to fetch all translations and check their data types, reporting any that are not strings. This approach involves creating and testing the job and then running it on a live server with the help of server admins. It's a thorough and systematic method, but it can be time-consuming and requires server-side execution.

Pros: Comprehensive and systematic.
Cons: Time-consuming and involves server-side execution.

Option 4: Add Logging for Detailed Insight

Insert `logging.error()` statements into the codebase to capture more detailed information when the error happens. By placing these logs around suspected areas of the code, you can gather data that helps you understand the problem better. However, this method requires reviewing server logs and might depend on waiting for the error to reoccur in production.

Pros: Provides detailed context; useful for understanding complex issues when other methods fail.
Cons: Requires additional review of server logs and depends on the error happening again in production.

Practice 7: Which option do you think is better and why? Consider:

- **Option 1** is ideal if the issue does not require specific production data. It's often the quickest way to validate your hypothesis.
- **Option 2** is preferable if the issue can be simulated with unit tests, especially when you have a clear understanding of the root cause.
- **Option 3** is useful when you need to perform a broad investigation of production data that you don't have locally, and you lack a clear lead on the issue.
- **Option 4** is helpful when other methods do not provide enough detail or you need to generate logs to diagnose the problem more precisely.

In our case, **Option 1** or **Option 2** is ideal since we have a clear root cause: the HTML content can be either a string or a list of strings. We want to verify that this discrepancy causes the error. **Option 3** can serve as a fallback if our direct testing methods do not fully replicate the error.

General Rule for Choosing an Verifying Option

As a general rule, start with the first item on the list above that makes sense for the issue you're tackling:

- **Do Option 1** if the issue doesn't require specific production data.
- **Do Option 2** if the issue requires specific production data, but you can simulate it with unit tests.
- **Do Option 3** if you need to know what's in the production data in the first place.
- **Do Option 4** if you have no idea what to do and need more detailed information.

Let’s go with Option 2 for this tutorial. Per the error log, the error is being generated due to this line:

```
plain_text = _get_plain_text_from_html_content_string(get_html_representing_suggestion(suggestion))
```

If we look carefully at the function:

```
def _get_plain_text_from_html_content_string(html_content_string: str) -> str:

 # Rest of the code.
   html_content_string_with_rte_tags_replaced = re.sub(
      r'<oppia-noninteractive-[^>]+>(.*?)</oppia-noninteractive-[^>]+>',
      _replace_rte_tag,
      html_content_string
   )
   # Rest of the code.
```

Within the function, the value `html_content_string` is passed to `re.sub`. The `re.sub` function, as defined in Python's standard library, expects its third argument to be a string (or a bytes-like object).

The error message `"expected string or bytes-like object"` suggests that `html_content_string` was not of the expected type when `re.sub` attempted to use it. This implies that the input passed to `_get_plain_text_from_html_content_string` was something other than a string or bytes-like object. And from the above investigation, we now suspect it to be a list of strings.

***Note**: Now that we understand why `mypy` did not catch this error, we can clarify that the type hint `html_content_string: str` in the function signature is actually accurate and not part of the issue. The type hint correctly indicates that the function expects a string argument. The problem arose because of how the attribute was dynamically assigned using `setattr` without a type check, leading to a case where a list of strings was mistakenly passed to this function.*

Let's take a look at the unit tests

Practice 8: Can you find the unit tests which could be used for our case? Check which unit tests are covering the behavior of generating certificates.

In the unit test class `ContributorCertificateTests`, we have the method `test_create_translation_contributor_certificate`, which creates a dummy translation that serves as the input for the certificate generation method. As you can see here:

```
change_cmd = {
            'cmd': 'add_translation',
```

```
            'content_id': 'content',
            'language_code': 'hi',
            'content_html': '',
            'state_name': 'Introduction',
            'translation_html': '<p>Translation for content.</p>'
        }
```

Translation_html holds the suggested value for an exploration card and here we are passing an HTML string. Let's try changing it from a string to a list of strings, as that's what we are suspecting:

```
 change_cmd = {
            'cmd': 'add_translation',
            'content_id': 'content',
            'language_code': 'hi',
            'content_html': '',
            'state_name': 'Introduction',
            'translation_html': ['<p>Translation for content.</p>'],
        }
```

Now let's run the backend unit tests by running `python -m scripts.run_backend_tests --test_target=core.domain.suggestion_services_test` (You can check out the wiki on how to run backend unit tests: https://github.com/oppia/oppia/wiki/Backend-tests)

Notice that the tests are failing with a similar error message to what we saw from the production logs.

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

======================================================================
ERROR: test_create_translation_contributor_certificate
(core.domain.suggestion_services_test.ContributorCertificateTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File
"/Users/ash/Desktop/openSource/oppia/core/domain/suggestion_services_test.p
y", line 7351, in test_create_translation_contributor_certificate
    suggestion_services.generate_contributor_certificate_data(
  File
"/Users/ash/Desktop/openSource/oppia/core/domain/suggestion_services.py",
line 3952, in generate_contributor_certificate_data
    data = _generate_translation_contributor_certificate_data(
```

```
  File
"/Users/ash/Desktop/openSource/oppia/core/domain/suggestion_services.py",
line 4026, in _generate_translation_contributor_certificate_data
    plain_text = _get_plain_text_from_html_content_string(
  File
"/Users/ash/Desktop/openSource/oppia/core/domain/suggestion_services.py",
line 1462, in _get_plain_text_from_html_content_string
    html_content_string_with_rte_tags_replaced = re.sub(
  File "/Users/ash/.pyenv/versions/3.8.15/lib/python3.8/re.py", line 210,
in sub
    return _compile(pattern, flags).sub(repl, string, count)
TypeError: expected string or bytes-like object

----------------------------------------------------------------------
FAILED    core.domain.suggestion_services_test: 1 errors, 0 failures
```

Thus, our suspicion appears to be correct. However, it's important to note that multiple bugs could produce the same error message, so while this provides strong supporting evidence, it doesn't conclusively confirm that this is the cause of the initial error.

You have successfully reproduced the error locally, validating your hypothesis and preparing to implement a solution.

Stage 5: Document Your Findings

Note: *When tackling server errors at Oppia, it is essential to document your findings thoroughly on the issue thread. This practice not only ensures transparency in the debugging process but also enables other contributors to understand the progress, validate the issue, and collaborate effectively on finding a solution.*

How to Document Your Findings:

1. Start with a Summary of the Error:
    - Provide a brief description of the server error you encountered.
    - Include key details from the error logs and any initial observations.

    Example:
    I encountered a `TypeError: expected string or bytes-like object` in the `generate_contributor_certificate_data` method while accessing the Contributor Dashboard. The error occurs at line 1408 in the `suggestion_services.py` file, specifically within the `_get_plain_text_from_html_content_string` function, which calls `re.sub`.

2. Detail the Steps Taken to Reproduce the Error:
    - Outline the steps you followed to reproduce the error locally.
    - Mention the environment setup, data used, and any modifications made to the code.

    Example:
    To reproduce the error:

        - I set up a local environment with the latest version of Oppia.
        - I modified an existing backend test in `ContributorCertificateTests` to create a dummy translation suggestion. Specifically, I adjusted the `translation_html` field to be a list of strings instead of a single string, which matches the suspected cause of the error.
        - I ran the backend tests using `python -m scripts.run_backend_tests --test_target=core.domain.suggestion_services_test`.
        - The tests failed with a similar `TypeError` as observed in the production logs, confirming that the issue is reproducible locally.
3. Identify the Commit or PR Likely to Have Introduced the Error:
    - Find the commit or PR that might have caused the issue using the Oppia wiki guide.
    - Mention the PR in your comment as a starting point for further investigation.

    Example:
    After examining recent changes, it appears that the issue might have been introduced in a PR that modified the `get_content_html` method to return `Union[str, List[str]]` instead of just a `str`. Link to the PR - Here, we are changing the return type of the get_content_html method from string to either strings/list. (It's not auto navigating, if you want to look, please check exp_domain file's line number 2096).

4. Explain the Possible Root Causes:
    - Describe your analysis of the potential causes.
    - Explain why the changes in the identified PR might have led to the error.
    - Provide any supporting information, such as error logs or specific observations.

    Example:
    The `get_content_html` method, which now returns `Union[str, List[str]]`, is being used in the `_get_plain_text_from_html_content_string` function. This function expects a `str`, but it sometimes receives a `List[str]`, causing a `TypeError`. The discrepancy between the expected input type (`str`) and the actual input type (`List[str]`) seems to be the root cause of the error.

5. Suggest Next Steps:
    - Recommend further testing, confirming the bug with other contributors, or starting work on a fix.
    - Clearly outline what should happen next.

    Example:
    Next steps:

        - Review the `get_content_html` method and decide whether it should always return a `str` or update the `_get_plain_text_from_html_content_string` function to handle both `str` and `List[str]`.
        - Check with other team members to gather insights on whether this bug might affect other parts of the codebase.
        - If necessary, begin work on a fix by modifying the relevant functions to handle different data types appropriately.

By providing a clear and detailed comment on the issue thread, you have effectively communicated the problem and your findings to other contributors. This will help others understand the progress, reproduce the issue, and collaborate on finding a solution.

Conclusion

Congratulations! You've learned how to debug server errors that are typically challenging to reproduce due to limited information. You've also developed skills in reading stack traces and determining the steps needed to reproduce a bug.

For more insight, you can explore some debugging stories we've listed on our wiki. To further deepen your understanding of debugging at Oppia, refer to our Debugging Guide.

If you feel confident and want to apply the skills you've acquired from this tutorial, consider tackling one of the following issues: Good First Issues for Server Errors.

Tidy Up

Now that you've completed the tutorial, it's important to tidy up your local repository to avoid any confusion in future development work. Follow these steps:

1. **Switch Back to the Main Branch:** Return to the main branch of the repository:
   `git checkout main`
2. **Remove Detached State (if needed):** If you checked out a specific commit and ended up in a detached HEAD state, you can safely delete any temporary changes by running:
   `git checkout -D 192f0a9a4866debac160015bc949130aaae6a7fe`

3. **Clean Up Untracked Files:** If there are any untracked files or changes, you can remove them using:
   ```
   git clean -fd
   ```

By following these steps, you'll have a clean working environment ready for future contributions or tutorials.

We Value Your Feedback

Did you find this tutorial useful? Or, did you encounter any issues or find things hard to grasp? Let us know by opening a discussion on GitHub Discussions. We would be happy to help you and make improvements as needed!