User Story Testing Checklist

To ensure a thorough peer review, check functionality **and** exploratory test, as outlined below.

❏ **Functionality Check**

❏ Review all code that was created/updated as indicated by the developer in the [design document](https://community.appian.com/w/the-appian-playbook/1598/appian-user-story-design-template) for the story or otherwise. Ensure that:

❏ The developer has included comments where code is difficult to understand or where important design choices are made

❏ The code follows team development standards (e.g. naming conventions, UI consistency guidelines, performance considerations)

❏ Expression rules include useful test cases

❏ Interfaces load with default test data

❏ Review applicable design documents to ensure that no functionality was missed in the implementation of the design

❏ Test ALL acceptance criteria and known edge cases

 ❏ Test from all applicable personas, not just as a system administrator

❏ For created/updated processes:

❏ Monitor the process and validate proper paths and process variables

❏ Check downstream data flows (e.g., audit fields are accurately updated)

 ❏ Review the Health Dashboard and ensure no new issues have surfaced and that there are no recommendations associated with the story package

❏ Review the unreferenced object section of the application to ensure no new, unnecessary objects have been created

 ❏ Review dependent systems to ensure the appropriate data is coming into and flowing out of Appian

❏ Ensure performance is within acceptable limits

❏ <Add additional, application-specific story testing criteria here>

❏  [**Exploratory Testing**](https://community.appian.com/w/the-appian-playbook/963/exploratory-testing)**:**

- **Note: How long should you test?**

- Testing is finished when the tester is confident that they have tested every identified risk. If the tester has not used up the time allotted for testing but feels that any new tests would be redundant, they should stop. If the tester has used up the time they originally allotted but still feels that there are areas of uncertainty in the functionality, they should continue until they feel comfortable that they have tested thoroughly.

❏ Read the design document (if applicable) for the story to do the following:

❏ Find areas of risk that should be tested

❏ Read the developer’s suggested tests to direct your testing

 ❏ How should you test?

❏ Do not test with System Admin credentials or with users that have access to multiple roles

❏ Leverage users that have one specific role/persona

❏ Happy path testing should include the user that you expect to be performing that action

❏ Negative testing should confirm users from other roles don’t have access to perform that action

❏ Use Exploratory Testing Heuristics to guide your testing. The most useful ones for our context are:

❏ **Never and Always**: Determine what the app should never do and always do, and see if these can be violated in any way by the functionality you are testing.

*Example:*

- *If there is a component or section that should always be hidden after an action is taken (a button is clicked, etc.), test it with different contexts or user groups to verify that it will always stay hidden.*

❏ **Beginning, Middle, End**: Vary the position of an element. You could vary where a given element appears in a sequence, or you could act on an element in a given position.

*Example:*

- *If you are using a paging grid and you delete the last item in the grid, did you reset paging to avoid a SAIL error?*

❏ **CRUD (Create, Read, Update, Delete)**: Using the other heuristics in this list, such as Beginning, Middle, and End or Zero, One, Many, change data elements by creating, reading, updating, and deleting them to ensure that the behavior is expected.

*Example:*

- *Use CRUD with Zero, One, Many in a scenario where the user can upload and delete documents by checking data elements in the database and in the user interface after uploading a document, deleting a document so that there are none uploaded, adding several documents, then deleting several documents, etc.*

❏ **Follow the Data:** In all actions which alter data, follow the changes in the data to find unexpected results.

*Example:*

- *If a process variable is passed throughout a process model, monitor the process and check the pv! each time it is modified by the user or the system*

❏ **Goldilocks:** Use inputs that are too big, too small, and just right. This may lead to errors or data truncation (see also **Too Few** and **Too Many**).

*Example:*

- *If the user has to enter in their age, try testing with negative values.*

❏ **Reverse:** Do things in reverse order. Undo everything. Skip to the end and work backward.

*Example:*

- *If you are using a breadcrumb-style approach to wizard navigation, try going backward through your breadcrumbs, or taking a path that users shouldn’t normally take.*

❏ **Too Few:** Applies when you have counts of things. Test with fewer things than the app expects.

❏ **Too Many:** Applies when you have counts of things. Test with more things than the app expects.

❏ **Violate Data Format Rules**: If the application expects values of a certain format, try violating those expectations to see how the system responds.

*Example:*

- *If there is a field for a user to type in their name, see what happens when they type in an integer or special characters instead.*

❏ **Zero, One, Many**: Test with inputs of zero, one, or many when appropriate to find:

1. Problems with plurals, such as “0 record found” or “1 records found”

2. Problems with count or percentage calculations, including divide-by-zero and off-by-one errors

*Example:*

- *If you are testing a grid, test with no records to show, one record, and a number of records much larger than the batch size.*