Migration Checklist (Upgrading Rock)

- ☐ Read the Release Notes for the new version you are upgrading to
- ☐ Make a note of any major breaking changes or deprecated features you need to test or make changes to your environment for (i.e. Lava Engine Changes)
- ☐ Export Copy of Production Database
- ☐ Restore Copy of Production Data on Dev Database Server
- ☐ Sanitize Database (Very important, don't want to send duplicate emails out, or let people make contributions)
- ☐ Connect Dev Environment to the Sanitized Database
- ☐ Upgrade Dev Environment to Desired Version
- ☐ Upgrade Plugins
- ☐ Testing Round One (Rock Team, Advanced Users)
    - ☐ Critical functions (Whatever they are for you, make sure the way you most often use them is working, i.e. Send an Email, Make a Donation, Watch the Livestream, etc. some of them may require extra steps like updating *your* email address from the sanitized version, configuring communications, or refunding donations after they've been made)
    - ☐ Custom Blocks
    - ☐ Modified Core Blocks (depending on how you upgrade you might need to reapply changes)
    - ☐ Find things that aren't custom code but are custom to your instance and make sure they don't need additional configuration after the migration (e.g. custom sub pages in person profile may need to be created again and updated after moving to v15+)
    - ☐ Check for places in your Rock instance you've written SQL (dynamic data blocks) or Lava. If those places are public facing or have reports critical to daily operations, make sure you test them now.
    - ☐ Create a list of all these things you needed to test (Page, Block, Description of Test, Notes)
- ☐ Development team should now fix any bugs/breaking changes that have been discovered. For custom blocks or modified core blocks this could involve development time to make them functional in the new version.
- ☐ Testing Round Two (Invite more or all of your staff)
    - ☐ Critical functions (What does each person need to do their jobs? This could be complex workflows you've built for them, or custom dashboards. Make them verify everything works the way they expect it to!)
- ☐ Development team should now fix any bugs/breaking changes that have been discovered in this round of testing.
- ☐ Create a list of post migration configuration you will need to do (Page, Block, Description)
- ☐ Create a list of post migration custom plugin updates you will need to do
- ☐ After verifying everything in the Dev Environment works as expected select the date you will upgrade production
- ☐ Upgrade Production Environment
- ☐ Run through post-migration check lists of additional configuration and custom plugin upgrades or core block modifications you need to apply
- ☐ Keep the lists you made this migration and add to them as new projects come up, now you have a comprehensive list of custom things to test for your next migration

Project Testing

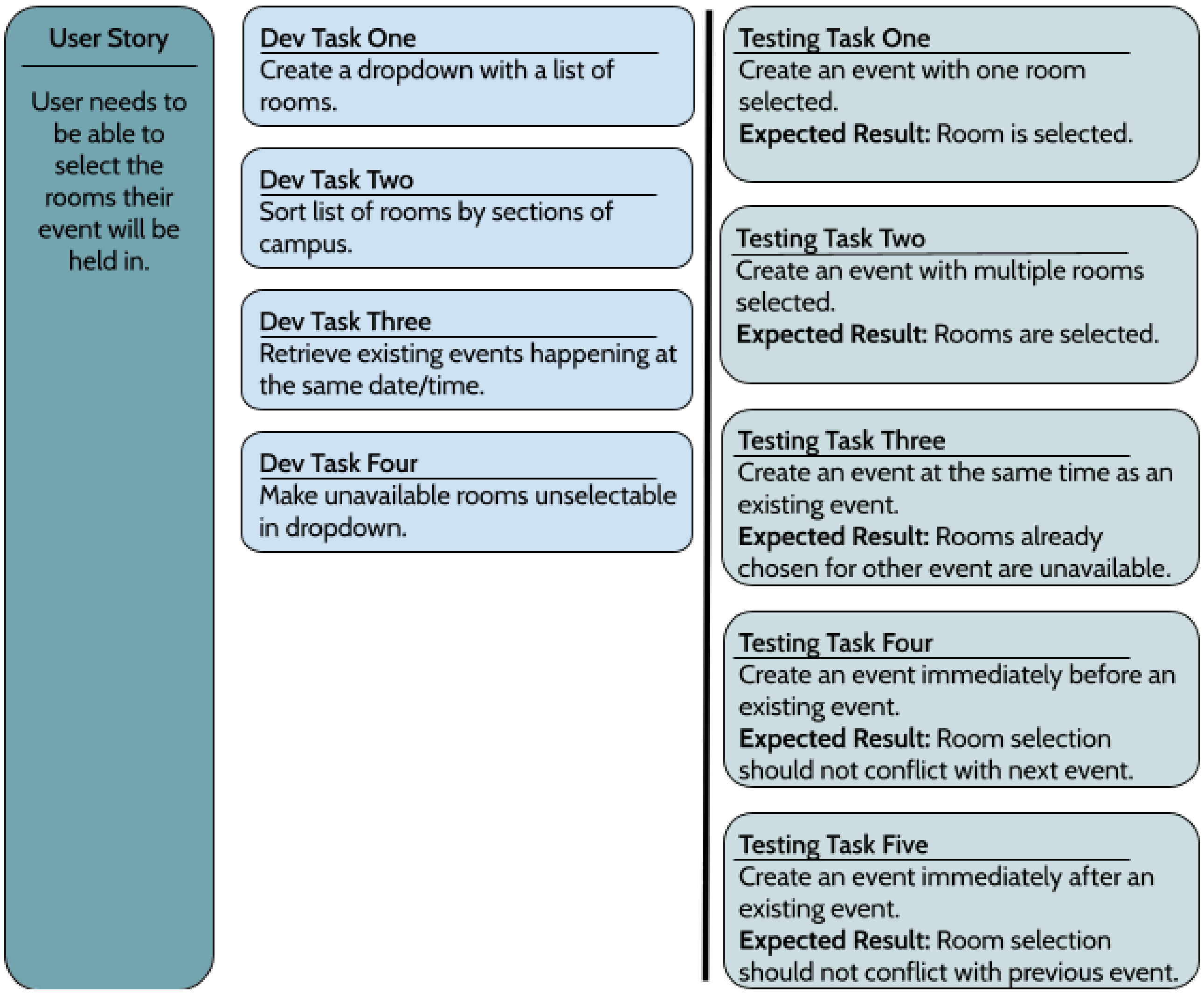
When defining a project with the product owner you should talk through all the requirements and write out a list of user stories. This is the list of everything the product needs to do, from the end user's perspective. The development team will create the necessary technical tasks to complete each user story but it can be helpful to keep each user story at a high level for communicating with the product owner and non-technical stakeholders.

It is also important to define exactly what good testing looks like for each user story or task so there is no confusion with your product owner on what you are asking them to do.

Once all the tasks to complete a user story have been developed then testing for that user story should begin. There will be some scenarios that the development team comes up with and specifically outlines for the product owner to test. This doesn't mean those are the only things that should be tested. The dev team should provide guidelines for the product owner to make sure testing is comprehensive, but the product owner and testers should also use the product organically and report any additional findings.

After bug fixes and other feature requests have been accommodated the product owner should mark the tasks or the user story complete.

Here is an example of a user story, the associated development tasks, and some possible testing tasks for the product owner. The product in this example is a form where ministry staff can submit requests to have events on campus.



User Story
User needs to be able to select the rooms their event will be held in.

Dev Task One
Create a dropdown with a list of rooms.

Dev Task Two
Sort list of rooms by sections of campus.

Dev Task Three
Retrieve existing events happening at the same date/time.

Dev Task Four
Make unavailable rooms unselectable in dropdown.

Testing Task One
Create an event with one room selected.
Expected Result: Room is selected.

Testing Task Two
Create an event with multiple rooms selected.
Expected Result: Rooms are selected.

Testing Task Three
Create an event at the same time as an existing event.
Expected Result: Rooms already chosen for other event are unavailable.

Testing Task Four
Create an event immediately before an existing event.
Expected Result: Room selection should not conflict with next event.

Testing Task Five
Create an event immediately after an existing event.
Expected Result: Room selection should not conflict with previous event.

Some examples on why testing this way is important...

That user story comes from an actual project at our church. At the time we created the product, we were not in a place where the dev team would explicitly help the product owner with their testing tasks. Our product owner and key stakeholder instinctively tested the first three tasks on their own. However, when testing an event on the same day as an existing event that did not conflict they did not test events immediately following or preceding the existing event.

The result, as you may have guessed, was that after the product was in production a staff member found they couldn't reserve a room for a seemingly available time. The event before it ended at 2pm but requesting her event to start at 2pm caused a conflict.

Those scenarios should have been laid out for the product owner from the start to avoid ministry staff finding the bug in production.

On the other side of testing, asking the product owner and testers to just use the product organically and see what they find, we've also discovered some scenarios we never anticipated when designing the product.

In addition to requesting a space for an event, our product allows ministry staff to request things like publicity, specific room set up, and amenities like coffee and sodas. One tester discovered that she could request custom set up and amenities for an off-site event. That is not a service we will provide for ministries so we realized we needed to make a change and prevent that from being an option.