Test Plan

Epic Preface

Here stands your valiant Lead QA and before him lies a blank text file. Upon its surface, tales of unimaginable tests could be etched, epics that would echo through the major tech conferences! Yet, a shadow falls upon our hero's brow. For within my breast churns a tempest of doubt, a chilling fear of self-inflicted folly! A thousand ingenious ways to fail dance in my mind, a morbid ballet that threatens to cripple my creativity! Can they overcome this internal strife and weave a tapestry worthy of the ages, or will they be my own undoing?

Introduction

This test plan outlines the functionalities to be tested for an email marketing editor that offers both a visual editor and a code-based editor for creating email templates.

Test Objectives

- Ensure the visual editor allows users to create professional-looking email templates with a user-friendly interface.
- Verify the code-based editor provides full control over email structure and design for advanced users.
- Test compatibility of designed templates across different email clients and devices.
- Test accuracy of editor preview when compared to test tools/targets
- Test usability of both editors.
- Test Batch scheduling options
- Test throttling options

Test Scope

- Message HTML Editor
- Message Batch scheduling and throttling

Out of scope

- Audience
- Message Header
- Goals and Tracking
- Alerting
- Tags
- Classification
- Sculpt Editor

General assumptions

Since this is a mock test plan I'm taking the liberty of assuming that we're running a Laravel based monolithic backend with some add-on services like the email sending part and a React front-end. I'm also not extensively listing the test cases, but a mere few just to relay the overall testing approach. Non-functional requirements are considered "loose" at this point since unless said otherwise a startup should be focused on delivering more and better features than nitpicking over optimizations.

HTML Editor

Content

Questions

- What character encoding is supported? What implications could supporting other encodings bring?
- What constraints do we have for the key attributes: name, key, description and tags. What length each attribute can have, special characters (e.g. emojis)? And what do we have in terms of input validation and sanitization for those fields both at the back-end and front-end. Are all the attributes mandatory?
- Where do the tags come from?
- Once created can the user edit those attributes?

Test approach

Ideally at the start of every test execution, whether it's manual or automated we would benefit a lot from having a "clean slate" data set, which we can get by building the database from the ground up with migrations and populating the entities with database seeds written for our specific test cases. This approach may sound over the top but allows us to conduct more deterministic tests and spares us the trouble of handling the cleanup data after the test execution. If getting a clean DB for test runs is not achievable we still can leverage the database seeds to set up some more complex cases, specially when we're talking about the more complex smarty tags and their html representations, assuring that we can easily verify a case and repeat such test as much as we need. And if all that poses a challenge we can just keep the test data at the test cases, usually stored in a test case management system like TestRail, which is pretty common and actually helpful when communicating with other teammates like developers and even product people.

On the matter of what and how to test, we would exercise the following:

- Start with the Content creation, at first with the "happy path"
- Negative testing the required attributes
- Boundary testing each attribute given the constraints we have
- Simple input sanitization test with the most common attacks like SQL Injection and XSS

Once the product is stable we could then start investing some time in more non-functional tests like performance testing, unless there are more strict requirements these tests are more of a benchmark to compare the results we have after each iteration. Despite technically security and usability also being non-functional requirements due to the nature of the project, those should be treated as first class requirements given the risks and value they respectively represent to our users.

Content Library

Questions

- What options are available for the advanced search?
- From the documentation the impression I get is that tags are categorizing attributes like "footer" or "call-to-action" but at the content library table it seems that the tag refers to the way the element is added to the email content.
- When we add a content where it is placed within the message content? Last place where the text cursor was left?
- What if the user has a text block selected? Will the content replace the selected text?
- Does the success toast displayed at the top right of the screen bring real value to the end user? Won't they just see the content added?
- If the user has the textarea scrolled past the point where the text cursor rests (cursor is not in view) will the textarea scroll and focus where the content was added?

Test approach

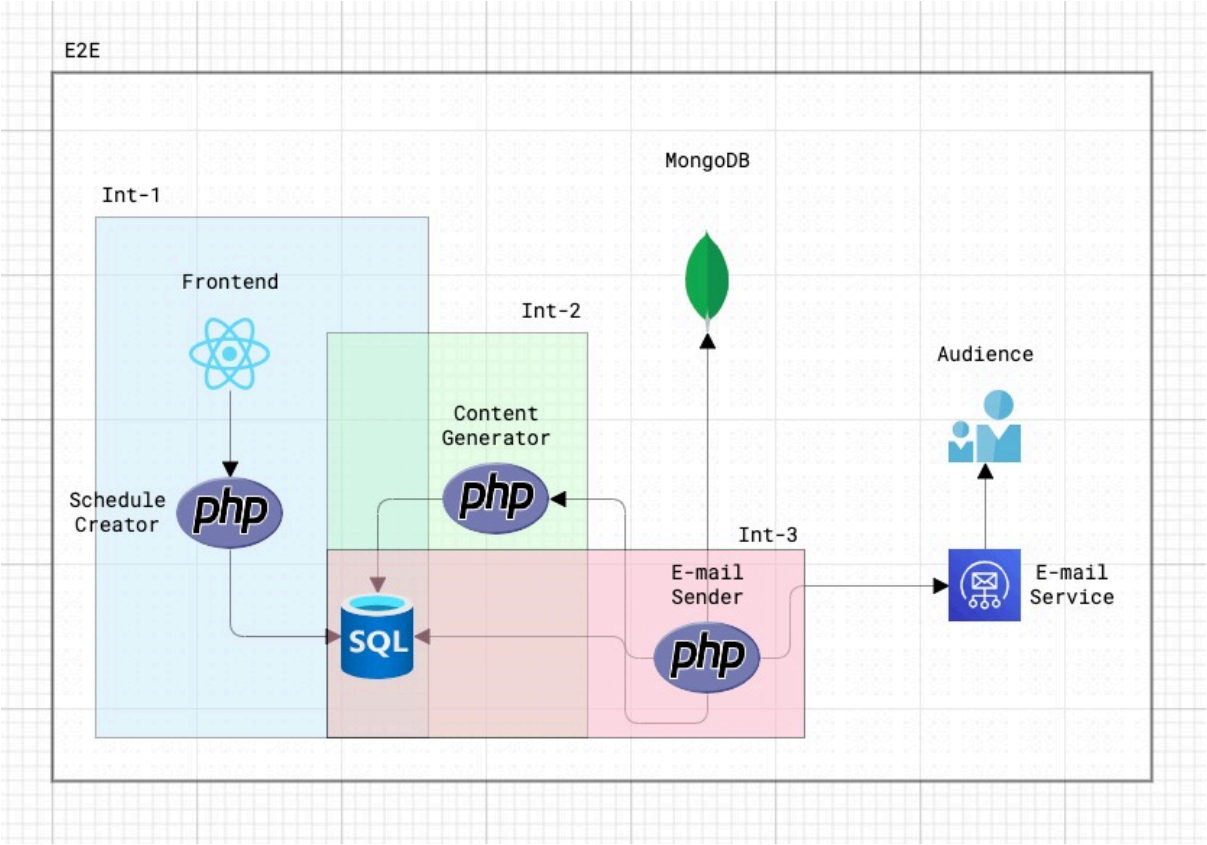
Similar to the setup for the Content, ideally we would have a set of "Contents" already available and the test cases would cover the usage of the tags in the email body, the correctness of the tags added and the user can save the template and make sure the tag is at its intended place.

- Cursor at the body of an empty template then add the Content
- Cursor at the body of a template with text then add the Content
- Cursor at the body of a template with a text block selected then add the Content
- Cursor at the first line of the body of a template with text then add the Content
- Cursor at the last line of the body of a template with text then add the Content
- Cursor at the body of a template with text, add the Content and check if the success toast/message was displayed

Batch scheduling and throttling

And we arrive at the tricky one, the automation nightmare, due to its nature my proposal would be to split the testing of this section in 3 parts so it can be more easily tested:

- Schedule creator, the front facing part where the user creates and configures the scheduling and batching configurations
- The email content generator
- The cron-like service that reads the configuration created at the "Schedule creator", invokes the "email content generator", and passes on all that information for the proper email sending service This approach doesn't mean we're not running a full end-to-end test, only that we're not relying so much in it, and makes our test results more accurate in the advent of a failure.



Questions

- How far in the future can the user schedule a Batch message?
- Can the user create a base Schedule Sending and leave it as a "template" for subsequent executions by copying it as draft?
- How does the user select the audience for the email?
- What happens when we schedule a Batch message at a given time that for the user is in the future but for the user that time is already in the past due to the timezone difference when they Schedule send according to contact's time zone?
- If the audience of the schedule has a mix of timezones when does the schedule

Test Approach

Since here we have a nice way of illustrating how to integrate different layers of testing for a common goal, let's start by discussing the lower layer, the unit tests highest priority is to speed up and make development more reliable, even when we're not refactoring, having clear unit tests points us to the expected behavior of that code unit, also working as documentation (a tricky kind of documentation that actually complains when we forget to update it).

The aim here is not to get a high code coverage, but to make sure that we're trying to mitigate some of the risks we have on this very sensitive part of the product. In the diagram you can see the colored squares delineating the boundaries of what we can call integration or service tests, at that level we will make sure that the services are behaving as expected but with an isolated focus, adhering to the expected contracts. Here lies also the textbook case where we can run a test case through a user interface and assert its effect directly at the database level.

Notice that the Schedule Creator service creates an entity that's stored at the DB. That Schedule is consumed by the Email sender service, but aside of sitting and waiting for the e-mails to be sent at the scheduled time, there's no other way to verify if the schedule stored at the DB is laid out as expected by Sender, so it makes sense to have a test case where we create a Schedule and check directly at the DB if everything is stored out as expected.

This may look like an overkill, but bear in mind that we're testing a very, if not the most, delicate part of the entire company, one defect here can lead to hundreds of thousand of emails to be sent to the wrong audience, at the wrong pace, with the wrong content, tarnishing not only our client's reputations, but also their domains and IPs.

Not only do we try our best to prevent any bugs, we also aim to use our tests to make it easier to debug and fix any issue that might go live, because time here is also of the essence, we have to recover as quickly as possible. And this approach may seem overwhelming for QAs to write all those tests, and it is, but as it was hinted when discussing about unit tests, here's where we can and must start bringing the entire team into the fray and have everyone contributing to writing and executing tests, from test case validation to actually writing automated tests and performing test runs.