**Lead Test Engineer – Technical Assessment Submission**

# ## Execution summary

This submission demonstrates my hands-on approach to test automation task focusing on high-impact, maintainable, and future-proof solutions. I focused on business-critical workflows, modern tooling, and clear defect reporting, reflecting both technical depth and leadership experience.

1. **Introduction**

I have designed a test automation solution for https://automationintesting.online, reflecting Company House’s current migration goals. The solution balances thoroughness with efficiency, focusing on critical user journeys, technical robustness, and future-proofing. My experience covers both modern and legacy automation tools, ensuring smooth delivery.

**2. Understanding the What, When, Why and How**

Before jumping into code, I reviewed the site https://automationintesting.online/.

# I understand key business flows (hotel booking, admin management), what needs automation, and how automation brings value. I prioritized areas most likely to disrupt user or admin experience, ensuring my solution targets high-impact risks and user-facing features.

# 3 Technical stack, Rationale, Tool choice and communication

# I chose to work with both Java/Selenium and Playwright/Typescript. Because the migration projects are heading in both directions and showcasing both aligns with your tech landscape. But also, I wanted to demonstrate that I’m equally comfortable with legacy refinement and modern re-platforming

# Technological stack and Rational

|  |  |
| --- | --- |
| Java 21 + Selenium 4 +Testing | This hits your internal system's future state — showing usage of native Selenium 4 features (relative locators, improved waits). |
| Playwright + Typescript | Playwright is the future-facing tech for front-end services. Fast, reliable, built-in waits, robust automation and supports parallel test execution  Typescript helps keep the codebase clean and maintainable, with strong typing and easy integration into CI/CD pipelines. |
| Allure Reporting | I wanted my reports to be human-readable — not just pass/fail. |
| JMeter | I simulated just enough traffic to show booking endpoint scalability and response times. |

That said, I’m also well-versed in Selenium/Java and can build equivalent test suites if your tech stack requires.

**Automation test coverage**  
Critical flows are completely tested with positive and negative tests. I’ve also included lightweight non-functional checks—such as accessibility and performance

**Test Selection - Priority:**  
My priority is to test the features that most impact your users and business—booking and contact forms, navigation, and room listings. Negative and edge-case testing are informed by common issues I’ve seen in production.

**Defect communication:**  
Defects are recorded through reproduce steps, description, priority severity, environment, test details and test evidence that will allow developers to locate and fix issues in the early stages. I arrange defect triage calls, including relevant stakeholders into a single room and discuss on the open issues, plan for the fix, capturing ETA (estimated turnaround time), root cause and seeking steps to prevent which is occurring in future. Post defects meeting, I would send minutes of meeting.

# 4. Test Design & Strategy

I focused and prioritized what impacts actual business value, critical customer facing functionalities - what stops a booking or impairs admin operations. I structured these using a modular Page Object Model in Selenium and component-based selectors in Playwright. Kept tests data-driven, used separate config files to switch environments, and added retry logic for flaky cases.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Scenario** | | |  | | --- | | **Reasoning & Test Objective** | |
| Booking Form Validation | Critical user-facing entry point. Validations, dropdown behavior, input types. |
| Room Availability Listing | Ensure dynamic updates post booking |
| Admin Panel Login | I checked for incorrect credentials and basic brute-force protection logic. |
| Booking API Integrity | Validated POST request payload structure and response schema. |
|  |  |

**Non-Functional Test Coverage (lightweight**):  
- Page load responsiveness

- Input validation (missing/invalid data)

- Accessibility (WCAG compliance)

- Performance (response time under load)

**Risk-Based Selection:**

I focus first on flows that would block bookings, break navigation, or compromise admin/security. Edge cases and negative tests mirror issues reflect real world production issues.

**5. Test Plan**

**Context & Scope:**  
Validate the essential workflows of [https://automationintesting.online](https://automationintesting.online/), focusing on features that impact the customer experience and business value.

**Key Objectives:**

- Ensure major user journeys (booking/contact) are robust, with proper error handling.

- Confirm navigation and room listings are always accessible and accurate.

- Evaluate non-functional requirements: performance and accessibility.

**Approach:**

* Use Playwright/Typescript for modern, robust web automation.
* Focus on functional test coverage for high-priority paths, using risk-based selection.
* Include negative and edge-case scenarios (invalid input, empty fields, long strings).
* Add non-functional checks to ensure the site is fast and usable for all.
* Document encountered defects with clear reproduction steps and expected results.

**Test Coverage:**

* **Booking Form:** Valid booking, invalid email, missing required fields, long/special input handling.
* **Contact Form:** Successful submission, missing required fields.
* **Navigation/UI:** Homepage, contact page, room listings, basic link integrity.
* **Performance:** Booking form response time < 3 seconds.
* **Accessibility:** Homepage and key forms scanned for WCAG violations.

**Exit Criteria:**

* All core functional and non-functional tests pass.
* No blocking defects in critical workflows.
* Defects are reported and triaged. Ensure fix plan in place , defects retested and test closure report issued

**6. Project Structure**

/playwright-tests  
├── tests/  
│ ├── booking.spec.ts  
│ └── contact.spec.ts  
├── pages/  
│ ├── BookingPage.ts  
│ └── ContactPage.ts  
├── utils/  
│ └── testData.json  
├── playwright.config.ts  
├── README.md  
└── testplan.md

**Setup**

Prerequisites:

* Java setup: Ensure java 21, Mvn install, run with mvn test
* Playwright setup: **Node.js v16+, npm install**
* Playwright - run all tests: **npx playwright test**
* Playwright - run a single test suite: **npx playwright test tests/booking.spec.ts**
* View Playwrightreport **– npx playwright show**

**Extending tests:\*\* Add new `.spec.ts` files to `tests/`.**

* JMeter: Open `booking\_load\_test.jmx` in Apache JMeter and run.
* Bug Reporting: Record defects in `docs/bug-report.md` with clear steps and severity.
  + **- Test Plan & Rationale:** See `docs/test-plan.md` for logic and coverage**.** Add new .spec.ts files to tests/
  + See Playwright docs for advanced usage
* Reports: HTML files generated locally, or view Allure dashboard at /target/allure-results
  + **Bug reporting:** Record defects in docs/bug-report.md with clear steps and severity.
* **Test plan and rationale:** 
  + See docs/test-plan.md for test selection and coverage logic.
  + See docs/bug-report.md for example defect documentation.

6.1 Page Object Model (POM) with Playwright : I use the Page object model pattern with playwright for better reusability, maintainability, scalability, and code cleanliness in the automation suite This design abstracts page interactions into dedicated classes, so tests focus on business logic rather than low-level selectors or commands.

**Uses of POM:**

- Centralizes selectors and page actions, Reduces code duplication

- Simplifies maintenance when the UI changes and Improves test readability and reusability

Example: Booking Page

typescript

import { Page } from '@playwright/test';

export class Booking Page {

constructor(private page: Page) {}

async bookRoom(firstname: string, lastname: string, email: string, phone: string) {

await this.page.fill('#firstname', firstname);

await this.page.fill('#lastname', lastname);

await this.page.fill('#email', email);

await this.page.fill('#phone', phone);

await this.page.click('text=Book This Room');

}

async getSuccessMessage() {

return this.page.locator('.alert-success');

}

}

**Example: Using the POM in a Test**

import { test, expect } from '@playwright/test';

import { BookingPage } from '../pages/BookingPage';

test('should submit booking with valid data using POM', async ({ page }) => {

const bookingPage = new BookingPage(page);

await page.goto('https://automationintesting.online/');

await bookingPage.bookRoom('Steve', 'Mark', 'steve.mark@example.com', '1234567890');

await expect(await bookingPage.getSuccessMessage()).toContainText('Booking Successful');

})

This structure makes the automation suite easier to extend and maintain as the application evolves.

**7. Test Scripts (Playwright/Typescript)**

**7.1 Booking Form – Success, Error, Edge Cases -** validates that form works with valid user data & invalid date

**Positive flow – validates that form works with valid user data**

import { test, expect } from '@playwright/test';

test.describe('Booking Form', () => {

test('should submit booking with valid data', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await page.fill('#firstname', 'Steve');

await page.fill('#lastname', 'Mark');

await page.fill('#email', 'steve.mark@example.com');

await page.fill('#phone', '1234567890');

await page.click('text=Book This Room');

await expect(page.locator('.alert-success')).toContainText(**'Booking Successful'**);

});

**7.2 Negative test – ensures email validation is triggered correctly**

test('should show error for invalid email', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await page.fill('#firstname', 'Steve');

await page.fill('#lastname', 'Mark');

await page.fill('#email', 'notanemail');

await page.fill('#phone', '1234567890');

await page.click('text=Book This Room');

await expect(page.locator('.alert-danger')).toContainText(**'Invalid email'**);

});

**7.3 Empty form submission – checks required field validation**

test('should show error if required fields are empty', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await page.click('text=Book This Room');

await expect(page.locator('.alert-danger')).toContainText('Required');

});

**7.4 Contact Form –negative tests Positive**

// Positive flow – validate successful contact form submission

import { test, expect } from '@playwright/test';

test('should submit contact form successfully', async ({ page }) => {

await page.goto('https://automationintesting.online/#/contact');

await page.fill('#name', 'Chris Tester');

await page.fill('#email', 'chris.tester@example.com');

await page.fill('#phone', '9876543210');

await page.fill('#subject', 'Enquiry');

await page.fill('#description', 'Just testing');

await page.click('#submitContact');

await expect(page.locator('.alert-success')).toContainText('Thank you for your message');

});

7.5 Empty submission – test for required field validation

test('should show error if required fields missing', async ({ page }) => {

await page.goto('https://automationintesting.online/#/contact');

await page.click('#submitContact');

await expect(page.locator('.alert-danger')).toBeVisible();

});

**7.6 UI Navigation & Room Listings -** To verify that the homepage, contact page, and room listings are accessible, correctly displayed, and that navigation between key pages works as expected.

import { test, expect } from '@playwright/test';

// Validates homepage content and smooth navigation flow between main pages

test('homepage navigation and content', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await expect(page).toHaveTitle(/Automation in Testing/);

await expect(page.locator('h1')).toContainText('Rooms');

await page.click('text=Contact');

await expect(page).toHaveURL(/.\*contact/);

await expect(page.locator('h2')).toContainText('Contact');

});

test('room listing visible', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await expect(page.locator('.room-info')).toBeVisible();

});

**7.7 Create a utility which would be reusable**

This utility helps generate **unique Mailinator email addresses** dynamically during test execution. It’s especially useful when testing workflows that send confirmation or notification emails (like booking confirmations) to a user.

**7.8 Example of creating the utility**

// utils/mailinatorEmailGenerator.ts

export function generateMailinatorEmail(prefix: string = "testuser"): string {

const randomNum = Math.floor(Math.random() \* 10000); // generates 0-9999

return `${prefix}${randomNum}@mailinator.com`;

}

**7.9 Using the above utility (importing) and use in booking test**

import { generateMailinatorEmail } from '../utils/mailinatorEmailGenerator';

test('booking form with Mailinator email', async ({ page }) => {

const testEmail = generateMailinatorEmail('qauser');

console.log(`Using test email: ${testEmail}`); // You can check it on Mailinator UI

await page.goto('https://automationintesting.online/');

await page.fill('#firstname', 'Jane');

await page.fill('#lastname', 'Tester');

await page.fill('#email', testEmail);

await page.fill('#phone', '0123456789');

await page.click('text=Book This Room');

await expect(page.locator('.alert-success')).toBeVisible();

});

8 **Admin tests validation**

The system should reject default credentials like 'admin'/'admin', which are commonly attempted in brute-force attacks.

import {test, expect} from '@playwright/test';

// Test to ensure 'admin'/'admin' credentials do NOT allow access to admin panel (security check)

test ('Security: Block login with username "admin" and password "admin"', async ({ page }) => {

await page.goto('https://automationintesting.online/#/admin');

await page.fill('#username', 'admin');

await page.fill('#password', 'admin');

await page.click('#doLogin');

// Assert that access is denied and no admin content is loaded

await expect(page.locator('.alert-danger')).toContainText(/not allowed|forbidden|invalid|blocked|unauthorized/i);

// Assert admin panel/dashboard is NOT visible

await expect(page.locator('.admin-panel')).not.toBeVisible();

// Optionally, check URL hasn't changed to admin dashboard

await expect(page).not.toHaveURL(/admin-dashboard|admin-panel/);

});

**9 Performance Test -** To verify that the booking form responds successfully within 3 seconds, ensuring acceptable performance for users.

import { test, expect } from '@playwright/test';

//Ensures booking flow completes within an acceptable user experience threshold (< 3s)

test('booking form response time < 3s', async ({ page }) => {

await page.goto('https://automationintesting.online/');

await page.fill('#firstname', 'Steve');

await page.fill('#lastname', 'Mark');

await page.fill('#email', 'steve.mark@example.com');

await page.fill('#phone', '1234567890');

const start = Date.now();

await page.click('text=Book This Room');

await expect(page.locator('.alert-success')).toBeVisible();

const duration = Date.now() - start;

expect(duration).toBeLessThan(3000);

});

**9.1 Accessibility Test -** To confirm that the homepage is accessible and does not have any violations according to the axe-core accessibility checks.

axe-core accessibility checks automatically validate your web pages against industry accessibility standards, helping you deliver inclusive, compliant user experiences.

// Validates that homepage meets accessibility standards for inclusivity and WCAG compliance

import { test } from '@playwright/test';

import AxeBuilder from '@axe-core/playwright';

test('homepage accessibility compliance', async ({ page }) => {

await page.goto('https://automationintesting.online/');

const results = await new AxeBuilder({ page }).analyze();

expect(results.violations).toEqual([]);

});

9. 2 Sample jmetre test –

I would like to add lightweight JMeter test simulates concurrent user bookings to verify the system's responsiveness under modest load. The goal is to observe how the booking endpoint handles parallel requests, identify potential latency issues, and ensure stability of the service — particularly under realistic user conditions. This reflects the kind of baseline performance check I’d routinely incorporate as part of sprint-end quality gates or release validation.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Scenario** | | |  | | --- | | **Reasoning & Test Objective** | |
| **Thread Group** | 10 users, 5s ramp-up, 1 loop |
| **HTTP Method** | POST |
| **Endpoint** | /booking |
| **Payload** | JSON body with booking details |
| **Headers** | Content-Type: application/json |

### JMeter Component Breakdown

1. **Thread Group**
   * Users: 10
   * Ramp-up: 5 seconds
   * Loop count: 1
2. **HTTP Request Sampler**
   * Method: POST
   * URL: https://automationintesting.online/booking
   * Body Data: Use the JSON above
   * Add **HTTP Header Manager**:
     + Content-Type: application/json
3. **Assertions**
   * Add **Response Assertion**:
     + Field to Test: Response Code
     + Pattern: 200, 201
4. **Listeners**
   * View Results Tree
   * Summary Report
   * Aggregate Report

Sample Json payload

{

"bookingdates": {

"checkin": "2025-07-20",

"checkout": "2025-07-22"

},

"depositpaid": true,

"email": "testuser@example.com",

"firstname": "chris",

"lastname": "Tester",

"phone": "0123456789",

"roomid": 1

}

**9.3 Broken links -** The objective of the broken links test is to ensure that all hyperlinks on the homepage lead to valid web pages and do not result in errors.

import { test, expect } from '@playwright/test';

test('No broken links on homepage', async ({ page }) => {

await page.goto('https://automationintesting.online/');

const links = await page.$$eval('a', as => as.map(a => a.href));

for (const link of links) {

if (link.startsWith('http')) {

const response = await page.goto(link);

expect(response?.status()).toBeLessThan(400);

await page.goBack();

}

}

});

**10 Few bugs identified:**

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | | Type(functional or non-functional | Bug description |
| 1 | | Functional | Booking form accepts invalid email format (&##\*\*@gmail.com) |
| 2 | | Functional | Booking form accepts past dates (e.g., 2023) without error |
| 3 | | Functional | Booking API accepts malformed payloads and returns inconsistent status codes |
| 4 | | Functional | Room listings do not update dynamically after booking |
| 5 | | Functional | "Send Us a Message" form allows single-character input in the name field |
| 6 | | Functional | Double-clicking submit causes duplicate submissions |
| 7 | | Functional | Navigation buttons/tabs lead to broken or incorrect pages |
| 8 | | Functional | Submitting forms with long or special characters is not properly validated |
| 9 | | Functional | Forms allow empty required fields to be submitted |
| 10 | | Functional | Broken links found on the homepage and key pages |
| 11 | | Functional | DELETE /booking/:id returns 201 Created instead of 200 OK / 204 No Content |
| 12 | | Functional | Booking can proceed even without selecting a room |
| 13 | | Non-Functional | Slow page load observed on the homepage and booking pages |
| 14 | | Non-Functional | Booking form response time sometimes exceeds 3 seconds |
| 15 | | Non-Functional | Room availability is not refreshed dynamically after actions |
| 16 | | Non-Functional | API allows high-volume concurrent requests (>200) with no rate limiting |
| 17 | | Non-Functional | Large or abnormal input lengths degrade performance or layout |
| 18 | | Security | Forms are vulnerable to XSS/injection (no input sanitization) |
| 19 | | Security | Sensitive information is shown in raw error messages |
| 20 | | Security | Booking IDs are sequential and easily guessable |
| 21 | | Security | No brute-force protection on admin login page |
| 22 | | Security | Malformed JSON payloads are accepted by API without schema checks |
| 23 | | Accessibility | Missing alt text on images |
| 24 | | Accessibility | Missing ARIA labels on form fields |
| 25 | | Accessibility | Tab order is broken — keyboard navigation is inconsistent |
| 26 | | Accessibility | Poor color contrast affects readability |
| 27 | | Accessibility | Forms and elements (e.g., Suite Room) not accessible via keyboard navigation |
|  |  |

**11 Bugs report examples**

**1. Bug ID:** BR-001  
**Title:** Booking form accepts invalid email format e.g.: &##\*\*@gmail.com

**Description of the defect:**  
The booking form allows submission with an invalid email ("&##\*\*@gmail.com ") and incorrectly shows success message what instead of an error.

**Steps to Reproduce:**

1. Go to homepage.
2. Fill booking form with " &##\*\*@gmail.com 3. Click "Book This Room".

**Expected Result:** No error message and proceeding to book

**Actual Result:** Validation error should be thrown – invalid email format

**Impact** – Booking is done with invalid email

**Severity:** Medium ;**Priority** : Major

**Screenshot:** *Attach screenshot if available*

***Time stamp****: < give date and time>*

***Environment****:*

***Test Phase:*** *<system testing.*

***Subtype:*** *Automation testing*

***Tested by****: <resource name>*

***Assigned to****: <development lead>*

***Status:*** *New*

***2. Bug ID: BR-002***

*D****escription of the defect****: Booking form past dates*

**Steps to Reproduce:**

1. Navigate to the website : Check Availability & Book Your Stay

2. Give past dates in check e.g.: 11/7/2023 and checkout 11/8/2023, click on check availability

3. Submit the form

**Expected Result:** Error message should through saying past dates not allowed for booking

**Actual Result:** proceeding further for booking

**Impact** – Booking system may create entries for the past dates

**Severity: Critical** ;**Priority** : Showstopper

**Screenshot:** *Attach screenshot if available*

***Time stamp****: < date and time>*

***Environment****: <environment name>*

***Test Phase:*** *<system testing.*

***Subtype:*** *Automation testing*

***Tested by****: <resource name>*

***Assigned to****: <development lead>*

***Status:*** *New*

# 12 Leadership approach

# I can mentor junior testers, lead knowledge sharing, and drive quality culture across teams. I have extensive experience of doing so in such a project type. As part of continuous learning, keep sharing what we learn, help each other out, and do regular reviews of our test cases to keep improving together.

# In real-world delivery, I’d integrate this suite with CI/CD (e.g., GitHub Actions, Jenkins), use environment-specific toggles. My past experience at Lloyds, RMG, and Maersk demonstrates my ability to lead such initiatives, guide teams, and drive quality-first culture across agile teams.

# 13 Future road map : From the future perspective, As part of making our test automation suite even better and more useful for the team, here are some practical steps I’d plan for the future for the effective test automation:

# Just few ideas are below:

# - Automate in CI/CD:\*

# Set up our tests to run automatically in our CI/CD pipelines (like GitHub Actions or Jenkins). This means every time someone makes a change, we’ll get instant feedback if something breaks.

**- Better Test Data Management:**

Make test data setup and cleanup more robust and automated, so tests don’t interfere with each other and we can run them in parallel

- **More Non-Functional Testing:**

Extend our coverage for things like performance (how fast the site is), how many users we can handle at once, and accessibility (making the site usable for everyone).

- **Add Security Checks:**

Start including some basic security tests, like checking for XSS or SQL injection, and use tools to scan for vulnerabilities.

**- Test on More Browsers and Devices:**

Use Playwright to make sure our site works well on different browsers and phones, not just one.

**- Tackle Flaky Tests and Visual Changes:**

Add tools to spot flaky (sometimes failing) tests and catch unexpected visual changes in the UI.

- **Improve Reporting:** Make our test results and reports even clearer, so anyone on the team can see what’s working and what’s not at a glance.

# - API Contract Testing:

# Add checks to make sure our APIs always follow the agreed rules (contracts). This helps catch breaking changes early and keeps integrations smooth.

# Booking journeys and admin were my priorities with load and API validations as the foundation. I automated UI with ease using Playwright and Selenium to demonstrate framework migration potential. I can also add contract tests with Pact or schema validators at the API boundary.

**E.g.:** Perform DELETE on an existing booking → verify expected HTTP code.

* Send malicious payloads (e.g., <script>…) during create/update.
* Issue PATCH requests (and check if the booking reflects the change).
* Blast more than 200 requests in 10 minutes to trigger rate limiting.

# I do keep an emphasis on achieving good foundation testing, with particular focus on the most important flows to users and establishing a basis for future easy maintainability.

# Note :

# I’ve made the solution simple, segmented, and functional—more about being useful than quantity-driven. I focused and prioritized key flows affecting actual users. This is how I'd develop on real projects

# If you so prefer, I'd be more than willing to clarify my line of thinking or even add the test cases to pick up on any specific cases that your interested in.