1. QA Strategy & Test Types

For this kind of customer-facing application, I'd rely on a mix of manual exploration to catch UX issues and strict automation for the business logic. Here is how I would approach it:

- **Exploratory Testing:** This is always my starting point (as I did with this task). I spent time just using the app without a script to get a feel for the domain, see how the forms behave, and spot any obvious usability frictions or logical gaps.
- **Functional UI Automation (E2E):** We need to cover the "Happy Paths" for all four verticals (Comprehensive, Civil-Servant, Clinic, Dental). The goal here is to make sure the calculator works after every deployment and the user can actually reach the result page.
- **API-Level Testing:** While inspecting the network tab, I noticed the calculations are handled by the `/api/tariff/otto/quotes` endpoint. I'd definitely shift a lot of the heavy lifting here. Testing calculation logic, parameter validation, and error handling at the API level is much faster and more reliable than doing it all through the UI. It also helps catch backend errors that the frontend might be swallowing.
- **Boundary Value Analysis:** Since insurance logic relies heavily on strict limits (age, salary thresholds), checking the exact boundaries (min/max/limits) is critical.
- **Usability & Cross-Browser:** Since this is a public web app, we need to ensure it looks good on mobile and that the questions (like "Employee Count") are clear to a non-technical user.

2. Test Cases & Automation Scope

For Automation (Regression Suite): I wouldn't automate everything. I'd focus on the high-value, repetitive flows:

- **End-to-End Happy Paths:** Entering valid data for each insurance type and verifying that the Result Page loads with the 3 Call-to-Action options (Appointment, Email, Signup).
- **Calculation Checks:** I'd verify that the premiums shown on the frontend actually match the JSON response from the backend API.
- **Blocking Logic:** Ensuring users can't proceed if they don't meet the criteria (e.g., income too low).
- **Dynamic Forms:** Checking if the form fields update correctly when I switch status (e.g., from *Selbstständig* to *Beamter*).

Edge Cases & Manual Findings (From my exploratory session): I dug around a bit and found a few specific things that we should probably clarify with the Product Owner:

- **The "Hidden" Salary Boundary:** The text says the limit is 77.400€, but I noticed a specific warning about the "2025 voluntary switch" appears at **73.800€**. Weirdly, if I enter **73.799€**, that warning disappears. This hidden threshold isn't explained to the user.
- **Civil Servant Date Bug (Frontend vs. Backend):** This was the biggest issue I found. The frontend lets me enter *any* start date. But if I select "Civil Servant" and

enter a date that isn't the **1st of the month** (e.g., Jan 30th), the API throws a `400 Bad Request`. The user just sees a generic "Calculation Failed" error at the very end and has no idea why.
- **Employee Count Ambiguity:** For self-employed users, the field accepts "0". It's unclear if this means "0 employees (just me)" or if it's an invalid input, which might mess up the sick pay calculation.
- **Status Clarity:** The difference between "Student" and "In Ausbildung" regarding insurance coverage isn't very clear on the UI side.

Technical Implementation & Architecture Highlights

To ensure the test suite is robust, maintainable, and scalable, I implemented the following architectural strategies:

1. Dynamic Date Calculation (Handling Data Aging) Instead of using hardcoded years for age validation, I implemented a dynamic calculation logic (`Current Year - Target Age`). This prevents "Data Aging," ensuring that tests for boundaries (e.g., <18 or >100 years old) remain valid and maintenance-free in the future.

2. Data-Driven Testing (DDT) for Business Logic I utilized a Data-Driven Testing approach to handle complex income scenarios. By separating test data (JSON) from test logic, I could efficiently cover multiple salary thresholds and corresponding warning messages within a single test block, ensuring comprehensive coverage of business rules.

3. Page Object Model (POM) Architecture I strictly adhered to the Page Object Model design pattern. By decoupling UI locators and interaction methods from the test scripts, I improved code readability and reduced code duplication. This structure makes future updates (e.g., UI changes) significantly easier to manage.

4. Robust Flakiness Handling To prevent flaky tests caused by the unpredictable appearance of the Cookie Consent Banner, I wrapped the cookie handling logic in a `try-catch` block (or conditional check). This ensures the test flow continues smoothly whether the banner appears or not.

5. DRY Principle & Test Hooks I leveraged Playwright's `test.beforeEach` hooks to handle repetitive setup steps, such as navigating to the URL and initializing page objects. This adheres to the DRY (Don't Repeat Yourself) principle and keeps the individual test blocks clean and focused on assertions.

6. Exploratory Testing & UX Feedback Beyond automation, I conducted exploratory testing to identify logical ambiguities and UX frictions (e.g., unclear input fields, hidden boundaries). I have documented these findings to facilitate a discussion with the Product Owner for further improvements.

7. Modular Utility Classes Common functions and reusable logic were centralized into dedicated utility classes. Instantiating these classes within tests simplified the management of shared operations and kept the codebase modular.