CRESTASTREAM

Test Automation Project

Comprehensive Technical Documentation

Playwright + TypeScript + Mock Server
Page Object Model Pattern
E2E + API + Hybrid Testing

Date: February 2026
Version: 1.0

1. What Is Cresta

1.1 About the Company

Cresta is a technology company founded in 2017 in Silicon Valley that develops AI-powered customer communication solutions. The company offers an AI-driven platform that enables contact center agents (customer representatives) to communicate more effectively in real time.

Cresta's core philosophy is this: AI does not replace humans; it assists them. While a customer representative is speaking with a customer on the phone or via chat, Cresta's AI engine analyzes the conversation in the background and provides suggestions to the representative.

1.2 Cresta's Product Suite

The Cresta platform consists of several core products:

Cresta Agent Assist

An AI-powered assistant that provides real-time suggestions to agents. When a customer voices a complaint, Agent Assist immediately displays the most appropriate response template, knowledge base link, or escalation recommendation on screen.

Cresta Director

An analytics dashboard that shows managers their entire team's performance. Real-time metrics, conversation analytics, sentiment distribution, AI score averages, and trend charts are all displayed on this dashboard.

Cresta Insights

A reporting tool that analyzes patterns and trends learned from all conversations. It provides information such as which topics receive the most complaints and which agent has the highest customer satisfaction rating.

Cresta Virtual Agent

An AI chatbot that can communicate directly with customers and resolve simple queries without human agent intervention.

1.3 What Does a QA Engineer Do at Cresta

A QA (Quality Assurance) engineer working at an AI company like Cresta carries responsibilities that are much broader and different from testing a standard web application:

- Real-time data flow testing: WebSocket connections, streaming data, live updates
- AI model output validation: Is the sentiment analysis correct Is the AI score reasonable
- High-load testing: How does the system behave with thousands of simultaneous conversations
- End-to-End (E2E) tests: Complete flows from login screen to dashboard, filters to export
- API tests: Verifying that backend endpoints function correctly
- Hybrid tests: Creating data via API and verifying its display in the UI
- Cross-browser tests: Does it work the same in Chrome, Firefox, and Safari
- Performance tests: Does the dashboard load within 3 seconds
- Security tests: Are XSS, SQL injection, and unauthorized access protected against

2. What Is the CrestaStream Test Project

CrestaStream is a test automation project that simulates Cresta's real product. The project consists of two main components:

Component	Description
CrestaStream MockServer	A fake backend server written with Express.js. It mimics real APIs.
CrestaStream Automation	A test framework written with Playwright + TypeScript. Uses the POM pattern.

This project covers all the test scenarios a QA engineer would encounter in the real world. Its purpose is to end-to-end test a Cresta-like product.

2.1 Project Scope

Test Type	File	What Does It Test
E2E - Login	`login.spec.ts`	Login screen, validation, error messages, security
E2E - Dashboard	`dashboard.spec.ts`	Metric cards, charts, table, filters, pagination
API	`conversations.api.spec.ts`	CRUD operations, health check, authentication
Hybrid	`hybrid-api-ui.spec.ts`	Create via API → Verify in UI

3. What Does the Mock Server (server.js) Do

3.1 Why a Mock Server

When testing a product in the real world, the backend (server side) is typically developed by a different team. Being dependent on the backend as a QA engineer is a significant problem:

- You cannot write tests if the backend is not ready
- If the backend has bugs, your tests fail for the wrong reasons
- Using a real database slows down tests
- Test data becomes uncontrollable

A Mock Server solves exactly this problem. You create a fake (mock) copy of the real backend. This fake server provides the same endpoints as the real APIs but stores data in memory (in-memory) instead of a database.

3.2 What Does the server.js File Do

Our server.js file is a Node.js application written with the Express.js framework. It performs the following functions:

1) Serves Static HTML Pages

The Login page (index.html) and Dashboard page (dashboard.html) are displayed in the browser. These pages are a simplified copy of a real Cresta dashboard. Each HTML element has a data-testid attribute added, so Playwright tests can reliably locate these elements.

2) Provides REST API Endpoints

Method	Endpoint	Description
POST	`/api/auth/login`	Performs user login, returns JWT token
POST	`/api/auth/logout`	Closes the session
GET	`/api/conversations`	Returns conversation list (supports filtering)
POST	`/api/conversations`	Creates a new conversation
GET	`/api/conversations/:id`	Single conversation detail
PUT	`/api/conversations/:id`	Updates a conversation
DELETE	`/api/conversations/:id`	Deletes a conversation
GET	`/api/metrics`	Returns dashboard metrics
GET	`/api/agents`	Returns agent list
GET	`/health`	Server health check

3) In-Memory Database

The Mock Server stores data in RAM. Every time the server restarts, the data resets. This ensures tests run independently from each other. For example, 3 test users (admin, agent, manager), 5 sample conversations, and 3 agents come pre-loaded.

4) CSP Header

Adds a Content Security Policy header. It grants the unsafe-eval permission so that Playwright can run JavaScript evaluate() on the page. This would not exist in a real production environment, but is necessary for the test environment.

4. Test Types — Detailed Explanation

4.1 E2E (End-to-End) Tests

E2E tests simulate the operations a user would perform in the real world from start to finish. They open the browser, navigate to the page, click buttons, fill forms, and verify the results.

Login Tests (login.spec.ts)

The login test file covers the following scenarios:

- Successful Login: Correct login with admin, agent, and manager accounts
- Remember Me: Login with the remember me option
- API Validation: Verifying the API response during login
- Invalid Login: Error message verification with wrong email/password
- Empty Form: HTML5 validation when an empty form is submitted
- Invalid Email Format: Warning message when the email format is incorrect
- Security: Password field masking, XSS attack protection
- Accessibility: Keyboard navigation, labels, ARIA attributes
- UI Interactions: Checkbox toggle, input clearing, form submission with Enter key

Dashboard Tests (dashboard.spec.ts)

The dashboard test file is much more comprehensive and covers the following areas:

- Dashboard Loading: Correct loading of all components (metric cards, charts, table)
- Performance: Dashboard loading in under 5 seconds
- Metric Cards: Display of 4 different metrics with correct values
- Table Filtering: Search, sentiment filter, status filter, multiple filters
- Date Selection: Today, last 7 days, last 30 days filters
- Pagination: Next page navigation, changing rows per page
- Export: CSV and Excel export buttons

- Charts: Trend chart and sentiment pie chart display
- Live Indicator: Live indicator status
- Refresh: Data updating with manual refresh

4.2 API Tests

API tests directly test backend endpoints with HTTP requests. They do not open a browser. Using Playwright's built-in APIRequestContext, they send JSON requests and verify the responses.

- Authentication: Login, logout, invalid token testing
- CRUD: Conversation create, read, update, delete
- Filtering: Filtering by sentiment, status, and agent
- Metrics: Correct retrieval of dashboard metrics from the API
- Health Check: Server health verification
- Error Handling: Error states such as 404, 401

4.3 Hybrid (API + UI) Tests

Hybrid tests are the test type that best reflects real-world scenarios. The fundamental logic is:

Create data via API → Verify display in UI → Modify via API → Check reflection in UI

- Create a conversation via API, check if it appears on the dashboard
- Delete a conversation via API, verify it is removed from the table
- Compare metrics from API with metrics displayed in the UI for consistency
- Bulk conversation creation and verification in the UI
- Error cases: Non-existent conversation ID, invalid login

5. Playwright vs Selenium

Playwright was chosen for this project. Understanding the differences between the two tools is very important:

Feature	Playwright	Selenium
Developer	Microsoft	Selenium HQ (open source)
Language Support	TypeScript, JavaScript, Python, Java, C#	Java, Python, C#, Ruby, JavaScript
Browsers	Chromium, Firefox, WebKit (built-in)	Chrome, Firefox, Safari, Edge (WebDriver)

Installation	Everything comes with npm install	Requires separate WebDriver download
Speed	Very fast (direct browser protocol)	Slower (HTTP WebDriver protocol)
Auto-Wait	Built-in (waits until elements are ready)	Manual waits required
API Testing	Built-in APIRequestContext	None (separate library needed)
Parallel Execution	Built-in, very easy	Requires Selenium Grid
Network Intercept	Built-in route/mock support	Limited (came with Selenium 4)
Trace/Debug	Trace Viewer (video, screenshot, DOM)	Limited logging
Documentation	Excellent, modern, with examples	Good but scattered

5.1 Why Did We Choose Playwright

1. Auto-Wait Mechanism: In Selenium, you need Thread.sleep() or WebDriverWait when an element is not found. Playwright automatically waits for the element to be visible, clickable, and stable.
2. Built-in API Test Support: We used Playwright's APIRequestContext in our hybrid tests. In Selenium, this would require a separate library like RestAssured or Axios.
3. Faster: Playwright communicates directly with the browser through the Chrome DevTools Protocol (CDP). Selenium uses the HTTP-based WebDriver protocol, which introduces extra latency.
4. Easy Setup: The command npm install @playwright/test automatically installs everything (including browsers). In Selenium, you need to separately download drivers like ChromeDriver and GeckoDriver.
5. Modern Architecture: Playwright was written from scratch by Microsoft in 2020. Selenium has existed since 2004 and is built on legacy architecture.

6. Why Did We Use TypeScript

We chose TypeScript over JavaScript for this project. There are very important technical reasons for this:

6.1 Type Safety

TypeScript assigns types to variables. This catches errors at compile time (before the code runs).

```
// JavaScript - Error only appears at runtime
const result = getMetrics();
```

```
console.log(result.totalConversation); // TYPO! Goes unnoticed

// TypeScript - Warns you immediately
const result: DashboardMetrics = getMetrics();
console.log(result.totalConversation); // ERROR! Property does not exist
```

6.2 IDE Support and Autocomplete

When you use TypeScript, editors like VS Code provide autocomplete (IntelliSense). For example, when you type analyticsPage. all available methods are listed. This is a huge advantage, especially with Page Objects.

6.3 Refactoring Ease

When you rename a method, TypeScript automatically updates all usage locations. In JavaScript, you would need to make this change manually everywhere.

6.4 Better Documentation

Interface and type definitions document the code itself. When another developer looks at the code, they immediately understand what each function takes and what it returns.

6.5 Why Not Another Language

Language	Advantage	Why We Didn't Choose It
Java	Common with Selenium, enterprise standard	Heavy, verbose, no natural integration with Playwright
Python	Easy to learn, short syntax	Weak type safety, difficult for large projects
C#	Strong type system, .NET ecosystem	Incompatible with Node.js ecosystem, extra config
JavaScript	Playwright's native language, easy start	No type safety, errors slip through in large projects

Conclusion: TypeScript = JavaScript's power + Java's safety. The ideal language for test automation.

7. Page Object Model (POM) Pattern

POM is the most fundamental and important design pattern in test automation. A separate class is created for each web page. This class contains the elements on the page and the operations that can be performed on that page.

7.1 Test Comparison: Without POM vs With POM

Without POM (Bad Practice)

```
// Same locators repeat in every test
await page.locator('[data-testid="email-input"]').fill('admin@test.com');
await page.locator('[data-testid="password-input"]').fill('pass123');
await page.locator('[data-testid="login-button"]').click();
```

With POM (Good Practice)

```
// Clean, readable, easy to maintain
await loginPage.login('admin@test.com', 'pass123');
```

7.2 POM Structure in Our Project

Class	Extends	Responsibility
BasePage	-	Common methods: safeFill, safeClick, waitForApi, getByTestId
LoginPage	BasePage	Login form: sign in, error checking, social login, navigation
AnalyticsPage	BasePage	Dashboard: metrics, charts, table, filters, export, pagination

7.3 Locator Strategy

The priority order of locators used to find elements:

6. data-testid (Best): Immutable, specifically designed for testing. Example: getByTestId('email-input')
7. getByRole (Good): Also useful for accessibility. Example: getByRole('button', { name: 'Login' })
8. getByLabel (Fair): For form elements. Example: getByLabel('Email')
9. CSS Selector (Last Resort): Fragile, subject to change. Example: page.locator('.btn-primary')

> *We used the data-testid strategy in our project. Since we control the HTML elements, we gave each element a unique test ID.*

8. Project Architecture and File Structure

8.1 Mock Server Structure

```
CrestaStream-MockServer/
├── server.js           # Main server file (Express.js)
├── package.json        # Dependencies
└── public/
    ├── index.html      # Login page
    └── dashboard.html  # Dashboard page
```

8.2 Test Project Structure

```
CrestaStream-Automation/
├── playwright.config.ts    # Playwright configuration
├── package.json            # Dependencies
├── tsconfig.json           # TypeScript configuration
├── pages/                  # Page Object Model classes
│   ├── base.page.ts       # Base class
│   ├── login.page.ts      # Login page
│   └── analytics.page.ts  # Dashboard page
├── tests/                  # Test files
│   ├── e2e/
│   │   ├── login.spec.ts
│   │   ├── dashboard.spec.ts
│   │   └── hybrid-api-ui.spec.ts
│   └── api/
│       └── conversations.api.spec.ts
├── utils/                  # Utility tools
│   ├── logger.ts           # Colored log output
│   └── api-client.ts       # API client
├── fixtures/               # Test data
│   └── test-data.json      # Users, conversations, filters
└── .github/workflows/      # CI/CD pipeline
    ├── ci.yml              # Tests run on every push
    ├── pr-smoke.yml        # PR smoke tests
    └── nightly.yml         # Nightly regression tests
```

9. Test Flow: How It Works From Start to Finish

9.1 Execution Steps

10. Mock Server is started: cd CrestaStream-MockServer && npm start
11. Server starts running on localhost:3000
12. Test command is given: cd CrestaStream-Automation && npm test
13. Playwright reads the playwright.config.ts file
14. Browser (Chromium) is launched (in headless mode)
15. Each test file is executed sequentially
16. Login is performed in each test's beforeEach hook
17. Test steps are executed (click, fill, verify)
18. Results are written to logs and the HTML report
19. Screenshots and videos are captured for failed tests

9.2 A Typical Test Scenario

Let's say the negative sentiment filter test on the dashboard is running:

20. Playwright opens Chromium and navigates to localhost:3000
21. The login form appears, email and password are filled in
22. The login button is clicked, /api/auth/login API is called
23. Token is received, redirected to the /dashboard page
24. Dashboard loads, /api/metrics and /api/conversations APIs are called
25. "negative" is selected from the sentiment filter dropdown
26. The table updates, showing only negative conversations
27. The test verifies that the row count in the table is greater than or equal to 0
28. The test is marked as PASSED

10. Key Concepts Glossary

Concept	Description
E2E Test	A test that simulates a user's operation from start to finish. Browser opens, clicks are made, forms are filled.
API Test	Testing by sending direct HTTP requests to the backend. No browser opens, only JSON data exchange.
Hybrid Test	Combination of API and UI tests. Creates data via API and verifies it in the UI.
Mock Server	A fake copy of the real backend. Provides the same APIs but stores data in memory.
POM (Page Object)	A separate class for each web page. Centralizes locators and actions in one place.
Locator	A selector used to find an HTML element on the page. Example: data-testid, CSS selector
data-testid	A special attribute added to HTML elements. Designed for testing, immutable.
Assertion	The step where the test verifies the expected result. Example: expect(count).toBeGreaterThan(0)
Auto-Wait	Playwright's automatic waiting mechanism. Waits until the element is visible.
Headless Mode	The browser running in the background without displaying on screen. Ideal for CI/CD.
CI/CD Pipeline	Tests running automatically every time code is pushed. GitHub Actions is used.
Sentiment	Customer emotion analysis. Positive, neutral, or negative.
AI Score	The quality score the AI assigns to a conversation. Between 0–100.
CRUD	Create, Read, Update, Delete operations.
JWT Token	Authentication token. Received after login, sent with subsequent requests.
Fixtures	Test data. Stored in a JSON file. Users, conversations, filter values.

11. Conclusion and Evaluation

This project covers all the test automation skills a modern QA engineer would need:

Skill	Usage in Project
E2E Test Automation	Login and Dashboard tests (50+ scenarios)
API Test Automation	CRUD, auth, health check tests
Hybrid Test Design	API-UI data consistency tests
Page Object Model	BasePage, LoginPage, AnalyticsPage
TypeScript	Type-safe interfaces, enums
Mock Server Development	Fully functional fake API with Express.js
CI/CD Integration	GitHub Actions workflows
Performance Testing	Load time measurement and assertion
Security Testing	XSS, input validation, password masking
Accessibility Testing	Keyboard navigation, ARIA labels

This project is a professional-grade test automation solution that simulates end-to-end QA processes at an AI company like Cresta.