Rethinking Test Automation:
The Case for Moving Beyond the User Interface

Rapid development models are forcing quality teams to balance speed with coverage. To enable both effective and efficient testing in this environment, businesses need to replace conventional UI-based automation techniques with more holistic approaches.

Executive Summary
With digital disruption and a fiercely competitive business landscape, many organizations are forced to balance frequent software upgrades and new feature additions with the ability to support all possible platforms and accelerate time-to-market.1 As pressures mount, many businesses are adopting lean development methods that offer complete lifecycle automation, from environment setup through build and deployment. These efforts make systems more complex, spawning a higher incidence rate of product quality issues due to insufficient testing.2

Conventional user interface (UI)-based automation approaches cannot handle the needs of these rapid development models, as they are unable to test components or subsystems from the get-go that contain business logic beyond the UI. For speed and coverage, it is important to look beyond automating the UI layer. This whitepaper focuses on unconventional methods and techniques that enable more effective and efficient test automation, early on in the process.

Limitations of UI-based Automation
Focusing test automation on functional behavior at the application UI is an easier option for test strategists, given the abundance of commercial and open source tools and implementation frameworks. In our experience, however, limiting testing to the UI layer may be inadequate due to the following factors:

- **Dependence on UI readiness**: In most applications, the UI is delivered later in the lifecycle. Waiting for the UI to be ready would mean introducing automation at a much later point in the application lifecycle. Minor UI changes almost always prompt object definitions to be updated frequently, leading to limited reuse and defeating the very purpose of automation.

- **Incomplete strategy**: For complex applications, in which business logic resides in multiple tiers, database objects and other components, it can be a challenge to cover all application controls and subsequent code interactions.
Lack of focus on the business layer: Some components, or a subsystems, containing business logic may need to be tested independently from the UI-based tests.

The need for faster turnaround: In projects based on Agile practices and projects that follow weekly/biweekly release cycles, fixing and turning around automated tests is challenging due to shorter testing windows. It may be time-consuming to fix multiple test failures due to UI changes.

Automation beyond the UI is immune to UI changes, resulting in reduced script maintenance efforts. This approach also enables testing of complicated business rules or validating component-level behavior independent of the UI, thus ensuring a complete testing strategy. Usually, testing at the middle tier or data level is faster than UI tests and provides faster feedback on application behavior. This also acts as an enabler for early testing in upstream environments.

Exploring Alternative Test Automation Approaches

The conventional strategy is to test end-to-end (i.e., as an end-user would do when using the application UI manually) or, more efficiently, with UI automation. Applications with multi-tiered architectures communicate using APIs, Web services and messaging middleware via the enterprise messaging bus. The UI (Web browsers, client applications) takes care of end-user interactions and presentation.

However, most business rules and functional complexities typically reside in the middle tier (see Figure 1). This creates an opportunity to validate business rules and functionality, enhancing coverage at the middle tier. Since communication between the interacting components, or interfacing layers, happens through message transactions, this creates an opportunity to:

- Simulate component or application behavior: Input messages directly into a system, thereby simulating end-user action and reducing UI dependency.

- Launch processes: Make a call to trigger API processing data or rules or, in turn, calling another API.

- Validate business functionality: Make a call to a service or API, parse messages and verify message content and/or querying status in the database for additional confirmation.

We propose the following solution:

- Analyze the technical stack: Study the application platform, services and message types, protocols and message formats, API available for interactions and database available for conducting additional validations.
• **Identify tests for non-UI automation:** Analyze the testing flow to identify the functionalities, components or flows that can be validated using the API or services. Also, identify the inputs for the tests and output to be validated. Any dependencies, such as test data and other third-party services, should be marked for consideration in the design phase.

• **Evaluate design and tool fitment:** Commercial tools, such as CA Technologies’ DevTest, IBM’s Rational Integration Tester, SmartBear Software’s SoapUI NG Pro, etc., may be evaluated for fit from a technology and design perspective. This approach should be governed by the costs involved; while these tools are feature-rich, it can be more economical to develop a client in Java or native technology. The objective should be to build a scalable solution to support all service or API testing needs within the portfolio.

• **Develop solution constituents:** Using a framework with reusable components that has multi-protocol and communication support can help reduce redundancies, as well as enable socialization and know-how. Building an architectural prototype will help establish the concept and encourage stakeholder feedback. The solution design should provide for segregated test data to facilitate data changes during releases. The solution should run tests individually or in a batch, with single-command execution, potentially run in a continuous integration pipeline, to ensure DevOps readiness in the development environment (duly supported by stubs or virtualization of services or third-party application behavior, in certain cases). This way, testing is automated using the interface layer rather than the UI tier, which is much slower.

---

**Quick Take**

**Automating Testing at a Large Investment Bank**

We helped a large investment bank improve its approach to testing and speed defect management. The bank’s cash equities portfolio consisted of regional order entry applications, a fix engine, order crossing and other applications that communicated with order management systems and trade reporting applications using a custom protocol based on TCP/IP. The applications followed weekly or fortnightly release cycles.

Although the bank employed test automation through the UI, a tighter maintenance window for automation scripts resulted in the need for more timely feedback. A custom server-side automation solution was developed to simulate trade entry, replay fix messages and check trade status in the order management system, as well as verify trade reporting status across various suites. A single test case could be executed in a few seconds that enabled the execution of thousands of tests within a few hours. This resulted in a drastic reduction of cycle time across the portfolio, as well as the ability to run thousands of tests on demand.

More than 60,000 tests a year could be run to certify the quality of software through server-side and UI automation. For applications with mature development processes, the automated suites were integrated in a continuous integration pipeline, providing continuous feedback on quality and reducing the defect detection and defect fixing lifecycle.
Potential benefits of our approach include:

- **Greater requirement stability and traceability**: Tests break less frequently as interfaces are well-defined and business process inputs and outputs undergo less change and deliver stable responses, hence bringing stability to the automation bed. These characteristics make the automation robust and ready to shift left (test execution) from the testing phase to the development phase of the SDLC.

- **Early and faster automation via parallel and multiple testing**: Techniques such as orthogonal array and combinatorial testing can be achieved only if tests can be performed beyond the UI. This is because UI-level validation prevents the code coverage from being exercised on edge conditions.

- **Rapid development support**: The feedback obtained from the API or service tests can be used to fine-tune the functional behavior at the UI layer, ensuring a better customer experience. These tests are well-suited for Agile/DevOps orchestration because each test runs in a few seconds, and is not susceptible to UI changes. Hence, it works well in continuous integration execution mode and can be made ready for DevOps.

---

**Quick Take**

**Saving Test Time and Effort for a Large Commercial Bank**

A large commercial bank in the U.S. sought to automate its trade order management and settlement systems.

The upstream applications book and manage trades throughout the lifecycle. The trades are then routed using RESTful Web services to an application server, and then to a Temenos banking application. In addition to existing UI-based automation with HP Unified Functional Testing, we recommended using service-based automation for faster feedback and setup of a continuous feedback mechanism using a continuous integration methodology (see Figure 2).

A Java-based solution was used in the services layer to inject trades into the booking application and further verify the state of the trade throughout the lifecycle and its final status in the trade database. Approximately 100 tests are run in roughly 45 minutes and provide quick feedback for different trade types. The entire test cycle span was reduced to one day for both service and UI tests. The automated test could be readily integrated with TeamCity for continuous integration.

**Beyond UI: Continuous Testing Across the Lifecycle**

![Figure 2](image-url)
Looking Forward

Testers usually employ black box testing techniques, with a range of input conditions, using the UI and verifying outputs. This leaves the core business logic residing in the API layer to be tested by developers, many of whom are not well schooled in the art and science of quality assurance.

The right approach for maximizing test coverage is to create a balance between UI and services/API automation. This approach provides more timely feedback and an ability to test early in the development process, whereas UI automation aims to validate logic in the UI layer and integration layer and mimics the actual end user of the system.

Tests at the non-UI layer will also increase code covered at the branch level, providing an option to test more edge conditions. Our experience has been that automation coverage can comprise service and server-side/API automation to a range of 60% to 70%, and UI automation to a range of 30% to 40%. However, a measured approach is to implement impact-based testing techniques that measure and map tests that cover all application code and suggest gaps where lines of code are not covered.7

With the adoption of cloud and the advent of microservices architecture,8 API and services-based testing becomes inevitable. QA organizations need to ascertain the quality of components early and independently in the development lifecycle before releasing them for further consumption by related components or subsystems.

Footnotes
6. “Shift left” is an approach to software testing in which testing occurs early (to find defects early) in the SDLC lifecycle as opposed to limiting testing to just the testing phase of the SDLC. For more information, see https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html.
8. Microservices architecture is an approach to developing software and enterprise applications with the ability to scale, covering a range of applications (Web, mobile) on multiple devices and platforms running in a cloud or hybrid ecosystem, and moving away from simple, monolithic Web applications. For more information, see https://en.wikipedia.org/wiki/Microservices and Mark Russinovich, “Microservices: An Application Revolution Powered by the Cloud,” Microsoft Azure, March 17, 2016, https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/.
About the Author

Nandan Shinde is Associate Director of Projects with Cognizant’s Quality Engineering and Assurance business unit and is part of the company’s Technology Center of Excellence. Currently a test automation architect, he previously worked as a test automation expert, project lead, automation strategist and program manager over a period of 11 years with Cognizant’s banking and finance services clients. His responsibilities pivot around building and socializing automation frameworks, as well as implementing solutions - from front-end, middle-tier and data automation strategies - to leading automation program for large banking customers. He handles automation consulting engagements across the vertical industries that Cognizant serves. He can be reached at Nandan.Shinde@cognizant.com.