

MASTER THE ESSENTIALS OF UI TEST AUTOMATION

Automation in the real world: a walk-through the most critical aspects of getting a successful, maintainable, valuable automation in place



CONTENTS

Chapter One: Introduction	3	Chapter Six : Automation in the Real World	21
Chapter Two: Before You Start	5	Developer Tester Pairing	21
Why Automate?	5	Maintainable Tests	22
Do You Have the Environment to be Successful?	6	Backing APIs	23
Chapter Three: People	9	Testable UI	24
Getting or Growing the Right People	9	Surviving Legacy UIs	26
Building UI Automation Skills	10	Remember: Take the Long View!	27
Your Team's Worth the Investment	11	Chapter Seven : Improving on Your Success	28
Chapter Four: Resources	12	Are You Solving The Right Problems?	28
Development/Test Systems	13	Gathering on Feedback	28
Build/CI Server	13	Retrospectives	28
Execution Agents	14	Sharing Feedback	29
System Under Test	14	Team Lunches	29
Leveraging Resources to Their Utmost	14	Build a Knowledge Base	29
Moving Forward	15	Trumpet Your Successes	29
Chapter Five : Look Before You Jump	16	Are the Stakeholders Happy?	30
How's Your Process?	16	Keep Working, Keep Learning	30
Clarify Expectations	16		
Start With a Pilot	16		
Practical Flow: Putting Concepts to Work	17		
Learning from the Pilot	20		

Chapter One

INTRODUCTION

The goal of this handbook is to help you understand the right questions to ask of you, your team and your organization. There won't be any Best Practices; there won't be any silver bullets. What we hope is to convey the right information to help you get started on the right foot and get through some of the most common problems teams hit when starting out with UI test automation.

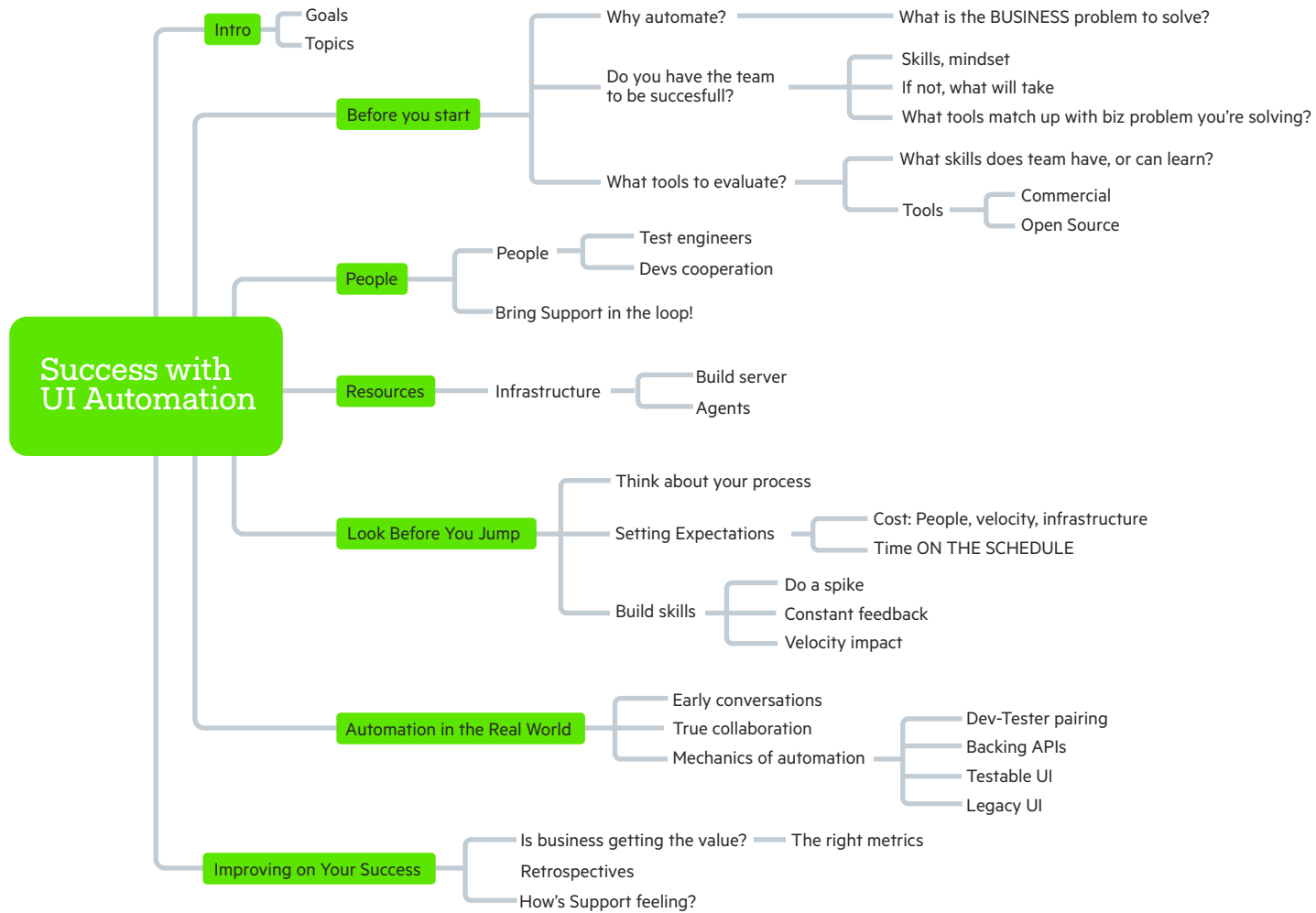
This handbook will walk you through what we think are the most critical aspects of getting a successful, maintainable, valuable automation effort in place. The chapters will include:

- **Before You Start:** What are the most critical things you need to think about before starting? We'll walk you through some of the questions to answer as you head off on this journey.
- **People:** You need a great team with specific skills to succeed. We'll help you understand how to build that team.
- **Resources:** Automation requires tools and infrastructure. We'll help you identify things to address as you move forward.
- **Look Before You Jump:** Test the assumptions you made

during your planning phase by working a prototype, spike or pilot project. Make sure the toolset, skills and process are close to what you need--and adjust.

- **Automation in the Real World:** Now it's time to put things into your real delivery pipeline. You'll read tips for easing your testing process, learn how to really collaborate with developers, and find how backing APIs and creating testable UIs can head off long-term pain.
- **Improving Your Success:** You've headed off on your effort. Now, how can you make it even better and guarantee your long-term success? We'll walk through how to create useful feedback loops that will help you smooth out any rough spots and leverage what's going well.

Here's a mind map of the chapter sequence including some of the topics discussed in each.



We hope this handbook will help you plan for your own UI automation projects, or potentially help you identify ways to improve projects on which you're currently working.

Chapter Two

BEFORE YOU START

“Plans are useless, but planning is indispensable.” This quote from American President Dwight Eisenhower is one of my favorite quotes. Sure, he's talking about the run up to D-Day in World War II, but it's applicable to so many things in life—especially software development.

You need to ensure you're spending your time and effort wisely before you jump in and start hacking away at automated tests. Here are a few things I've found helpful to work through as part of the planning process.

Why Automate?

Why are you considering bringing in test automation to your delivery process? Take some time to get very specific about the problem you're trying to solve. More importantly, make sure you're tackling a concrete business problem. UI automation's a nifty tool, but it's an expensive one to get adept with, and it's a very costly tool to deal with if you use it badly. It's also only one form of test automation, and by itself it's not going to solve much of any delivery “challenges” you're having.

Here are a few business-related areas in which UI automation may be helpful:

- Long release cycles due to time required for manual regression testing
- Testing falling behind development during release cycles/iterations/sprints
- Rework costs due to regressions of high-value features
- High support costs due to escaped bugs around high-value features
- High cost of testing high-value features against multiple browsers and operating systems
- Exploding cost of testing against multiple mobile platforms and browsers

Worst of all, here are two areas that trump all other concerns:

- Loss of executive-level trust in the team's ability to deliver good software
- Loss of trust and business from your end users or customers

The last two, especially the last one, should concern every team member. It's really bad if you've lost higher-level management's trust. It's potentially disastrous if you're losing customers.

There are some areas for which you should not consider UI automation as a solution:

- Validating cross-browser look and feel
- Guarding against layout and style regressions
- Saving money by cutting number of manual testers

UI test automation may help you solve technical or process problems, but ensure you're first asking **why** and focusing on solving business problems first.

Do You Have the Environment to be Successful?

Once you've decided that UI test automation is a good choice for you, the next critical factor is whether your team is able to be successful at UI automation. Several different aspects come into play:

Do You Have Stakeholder and Sponsor Buy-In?

Adopting UI automation will require significant changes to how you work. You'll need team members with the right skills, you'll need resources (no, "resources" are not people), and you'll need additional time.

Adding UI automation into your delivery process will decrease your velocity. You have to make sure your stakeholders and sponsors truly understand that. They have to support the decrease in velocity for the improvement in delivered business value.

"Decrease in velocity" is always something that concerns the business side of the organization. I've found it very helpful to tie back to the specific business-level issues discussed earlier in this post. Say this:

"Yes, we're going to slow down a bit, but the goal is to cut the support costs we're incurring through escaped bugs. We're also hoping to gain back revenue we've lost from the decline in license and service renewals."

This links your efforts back to the things the stakeholders really care about: the organization's mission and bottom line.

Do You Have the Right Communication?

Good communication is the foundation of every successful human effort, software notwithstanding.

- Are your teams able to get good information about features in a timely fashion?
- Do designers, developers and testers all talk regularly about how UX/UI work is accomplished? Can your testers get early input on UI design to help make testable screens?
- Do your testers understand what the stakeholders' highest priorities are for each feature, and do they understand the business value behind those features?

If the answers to any of those questions are “No,” you will need to address those issues as you move forward with your automation—or suffer the friction and stress that falls out.

Great communication helps ensure everyone involved in UI automation knows the priorities before they start their work. It helps everyone understand how to balance automation work with focused manual testing, and it helps focus automation work on the right parts of the system.

Do You Have the Right Team Structure?

UI automation rarely succeeds when testers are expected to create all automation in a silo or walled-off room. I already mentioned the importance of early, frequent communication between developers, stakeholders, testers—basically the entire team.

If your teams are fragmented into highly constrictive silos, you will likely see additional friction and difficulties when trying to clear up basic fundamentals such as getting good locators on elements around which you’re building automation scripts.

The best structure for any team is an open environment that empowers frequent communication directly between concerned team members. Co-located teams always function the best; however, geographic dispersion can't be an excuse for poor communication.

Clearing communication problems can often be difficult; however, two steps can often reap huge benefits. First, get communication bottlenecks out of the way. Guide project managers and other mid-level management to avoid requiring communication to flow through them. Communication should be as direct between people as possible. The wheel/spoke communication model is long outdated...

Second, encourage your testers to reach out directly to developers whenever possible. Breaking down this wall is critical for seeing smoother automation in the long-run.

Setting up a team's structure for success means as few roadblocks and walls to frequent, candid discussions.

What Tools Can You Use?

Notice I've left the actual tooling for last. Yes, yes: the toolset you use is critical, but it doesn't matter what tools you select for automation **if you haven't answered the harder questions first.**

You can finally jump into tool selection once you've addressed that you've got a clear case for why you're going to use automation, and what problems you're trying to solve.

There are a huge range of UI automation test tools available. Some are open-source, some are free and some are commercial. There are also many types of tools, from drivers to frameworks to entire suites.¹

Selecting the right toolset for your team means answering a few more questions:

- **What communication mechanisms will you use for tests?**

Do you need a grammar-based specification? Will recorded tests with coded steps be clear enough? Are 100-percent coded tests clear enough for everyone?

- **Who writes the tests?** Will testers be solely responsible for test creation? Or, will developers have a hand in it, as well?

- **Who maintains tests?** Will the team that writes the tests maintain them, or do you have an outside contractor writing tests and handing them off to an internal team? If you have two (or more) different teams, make sure the teams have the skills, aptitude and time to take on the selected tool.

- **Who uses the tests?** How will you run your tests? Will the suites be triggered manually? Will they be part of a scheduled suite to be run via a [Jenkins CI server](#) or Team Foundation Server build? You'll need people who can handle the care and feeding of those environments, and you'll need the infrastructure required, too.

- **Who uses test results?** Who in your organization needs what level of information about your tests? Keep in mind that your

stakeholders often need one set of data, while your team needs another.

- **Do we have the skills?** Lastly, you'll need team members with the right skillset to build, manage and maintain all the pieces necessary for a successful automation effort. Your team doesn't need those skills right now, but they'll need support to develop those skills in a timely fashion.

Telerik put out a [“Buyer’s Guide” in 2014](#) that answers these and other questions.

¹An automation driver is responsible for driving the UI application around. Think of Selenium WebDriver or the Telerik driver. An automation framework sits atop the driver and normally gives teams a grammar-based approach for writing tests (think Cucumber, Fitness, SpecFlow and so on).

Chapter Three

PEOPLE

As with everything else in software development, success comes through getting good people and giving them the tools they need to succeed. Lining up the right people for your successful project means finding ones with the right skills, or giving them time and support to grow those skills.

Getting or Growing the Right People

Please get this clear in your mind immediately: you can't take developers with no UI automation background and expect them to succeed at test automation without help. You can't take manual testers and expect them to succeed at test automation without help. You can't take non-testers, give them a fancy tool and expect them to succeed at test automation without a lot of help. Ensure you're setting your organization up for success by ensuring you're able to get or grow the right team members.

UI automation requires a specialized set of skills to be effective. Becoming adept at UI automation requires time and mindful practice. ("Mindful practice" is a term used for carefully chosen work/study/practice meant specifically to improve one's skills or knowledge. You

can't just "go through the motions" and expect to improve. You need to dedicate yourself to improvement. Read more in Andy Hunt's [Pragmatic Thinking and Learning](#).)

Keep in mind that it's not enough to simply learn UI automation skills—just as you want developers who write great maintainable code, you also want people who understand how to write great maintainable automation scripts.

Finding great people in any corner of the software development domain is hard; finding people with great UI automation skills is much harder. It's a specialty that's underappreciated and not practiced widely enough. Make sure you have your time frame and budget expectations properly set if you're looking to expand your team by hiring.

What I've found more effective in my experience is to find current team members, or people elsewhere in my organization, and bring them to my team, then develop their skills. Poaching from other teams ("stealing" is **such** a harsh word!) can be politically tricky at times, so make sure you're not sawing off the branch you're standing on, metaphorically speaking.

Building UI Automation Skills

You'll need to work hard at building up your team's skills. This means you'll need both resources to learn from, plus a plan on how to learn.

Resources for Learning

Thankfully, there are a lot of great resources around to help you learn UI test automation. Better yet, many of the fundamental concepts are the same, regardless of the specific toolset you're working with, so you can look to industry experts rather than just tool experts.

Of course Telerik offers [training specific for Telerik Test Studio](#), but you can also look to some of these resources to help you build your domain-level knowledge of UI automation.

- Dave Haeffner's [Elemental Selenium newsletter](#).
- Dave's The Internet project on [GitHub](#) and [Heroku](#) is a set of common automation problems like forms-based authentication, asynchronous actions, drag and drop and so on. It's great for learning fundamentals.
- Marcel de Vries' Pluralsight course on [Coded UI](#) (subscription required).
- Richard Bradshaw is prolific both on [Twitter](#) and [his blog](#). His post on [handling setup and configuration for WebDriver](#) is full of great ideas.

Getting Your Team Learning

Coaches, trainers, and consultants in the testing domain commonly echo a similar refrain: teams need directed, thoughtful practice to become adept at testing automation. Often that “practice” can be directly related to the work that’s being done, but the crux of the matter is your teams will need time to master this tricky domain.

Remember: as you get to this point, it's critical that you've got support from stakeholders and your team. That support is key to getting through the learning process.

Here's an outline of a process that may be useful to you:

- 1. Find a guide:** Look for someone with UI automation experience to help you and your team. Is there someone on a different team in your organization? See about getting them on your team, at least part-time. If not, hire an external guide to come in and help. Regardless of whether your guide is internal or external, this has to be a long-term commitment. You'll need more than a 2-day workshop; you'll need a relationship that will be highly involved for weeks, then ramp down over several months.
- 2. Make a plan:** Lay out a roadmap for your team's learning experience. You'll need to think of things like general tool competency, creating backing APIs/helpers and, of course, creating the actual tests. Ensure all this work is in your backlog,

work item tracker, Kanban board and so on. No training, skills building or actual automation work gets hidden!

3. Pair up: [Pairing](#) isn't just for developers. Pair less-adept team members with those who have more experience. You'll see benefits even if your team has no experience—putting two novices together, with frequent oversight and constant encouragement, can result in those two talking through problems and coming up with solid responses. (**NOTE:** This doesn't mean you can skip getting good guides for your team.)

4. Focus on value: You'll need to start with automation scripts that focus on small, “[low-hanging fruit](#)” features with which your team can be successful. That said, focus on scripts that will check honest business value for your stakeholders. Avoid wasting time on things like look and feel.

5. Trumpet successes: Make sure the team sees successes, especially as they're struggling to get up the learning curve. Figure out what metrics make sense to monitor for your team, and then get those up on a [Big Visible Chart](#) that everyone can see.

6. Share knowledge: Make sure that your team has the tools in place to share knowledge. Lessons learned, both positive and negative, are a huge help in the team's climb up the learning curve. Get some form of a knowledge base in place via

Evernote, a wiki, or similar tools. Make sure it's easy to edit and search. Add “lunch and learn” [brown bag](#) sessions for the group to demonstrate concepts. Whatever you do, create the mindset that knowledge sharing isn't an option; it's expected.

7. Constant feedback: Software delivery benefits tremendously from constant feedback. Successful UI test automation projects, especially so. Prompt members to discuss automation during the teams' [daily standup](#). Ensure your [team retrospectives](#) bring up good and bad points of your automation efforts. Most importantly, get stakeholder feedback on how they feel the effort is helping them: do the business owners feel they're getting better information to help them make informed decisions?

Your Team's Worth the Investment

Getting the right people lined up and empowered to succeed is crucial. Learning to master automation takes a long time, but the payoff is worth it: better value and higher-quality software delivered to your users.

Chapter Four

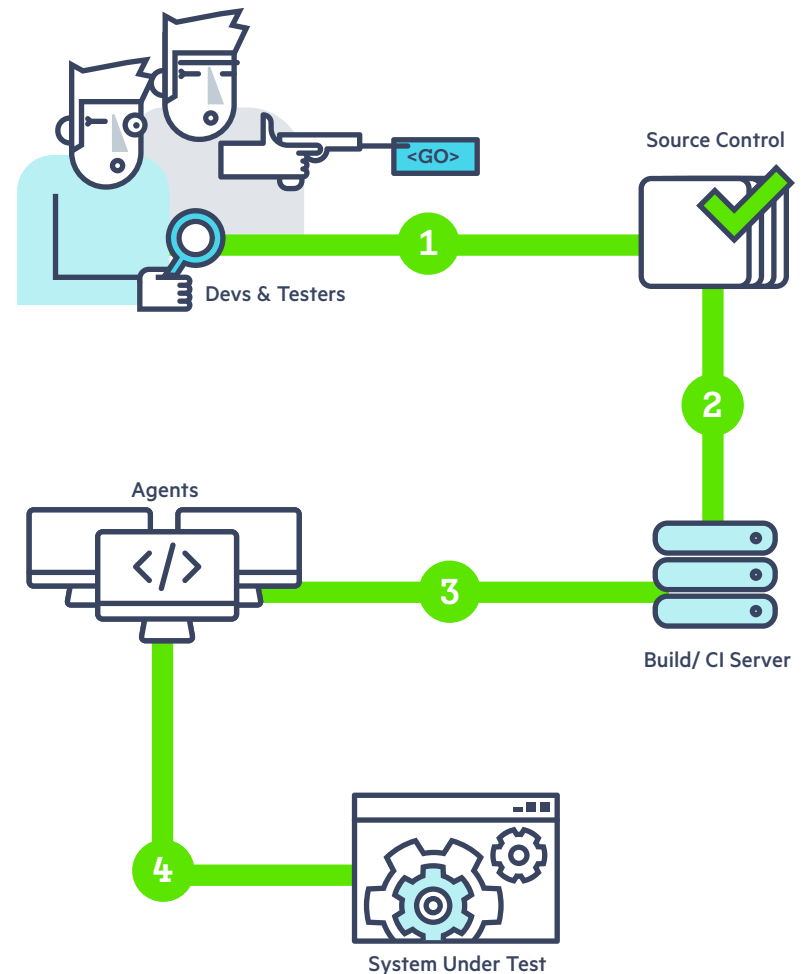
RESOURCES

We've walked through getting a plan in place for having your team build their automation skills. Now you've got to consider another aspect of the team's success: proper tools and infrastructure to help them get their work done. Your team will need a number of pieces to create, execute and maintain your UI automation suites effectively.

The diagram on the right shows what a typical infrastructure looks like. Obviously, some environments have many more moving parts.

Members writing automation scripts will generally build their scripts and check them into a **source control system** such as Team Foundation Server, Git or SVN (see flow #1 on the diagram). **Source control systems** are a repository that holds software changes in a fashion that each change can be separately identified and managed if needed.

Build servers handle tasks like building/assembling software. Build servers generally link to source control and also enable other tasks such as automated testing to be performed. A build server will pull the latest version of tests as a suite from source control (flow #2). The suite is compiled (if necessary), then the build server hands actual test execution off to one or more agents (flow #3). This test



pass, or job, is generally scheduled rather than running on a constant basis in a [continuous integration model](#)—UI automation tests are simply too slow and long-running to have them blocking other **continuous integration** builds and tasks.

Continuous Integration is an extension of build services and also encompasses team processes. CI normally builds a system, then deploys it to a specific environment where automated and manual tests are executed against it.

Test agents for UI testing can be very lightweight systems, often virtualized, that handle executing tests against the system under test (SUT) (flow #4). The agents can be a mix of different operating systems with different browsers. This helps to ensure you're getting proper OS and browser coverage. Agents can also run on mobile devices if the organization needs mobile coverage too.

Agents report test results back to the build/CI server, which then makes reports and notifications available to the team.

Let's dive further into each component of this diagram.

Development/Test Systems

Your team needs adequate systems to create the automation suites. There is plenty of evidence that team productivity is enhanced when they have access to solid, well-powered systems. Most developers end up with very high-performance systems.

Testers, or those mainly responsible for your automation scripts don't need quite that amount of power. Test automation projects shouldn't take much horsepower to build, but you do need to ensure slow systems won't leave your team hanging in the air while tests are compiling or running.

Your team will be using these systems to write, troubleshoot, execute and maintain your test suites. These systems will need access to the source control repository, build/CI server, SUT host(s) and agent systems.

(In these systems is where Telerik Test Studio, either standalone or Visual Studio version, live.)

Build/CI Server

Build servers come in many shapes and flavors. Team Foundation Server, Team City and Jenkins are just three of the most popular systems; there are many others.

The basic advantage of CI is, at a bare minimum, team members know if commits from multiple members have somehow broken the system—the infamous “builds on my system, but fails on someone else's” scenario. CI servers are generally configured to run additional tasks after the basic build, such as executing unit tests. (Remember from above that build servers can and often do a lot more than just unit tests.)

Execution Agents

Agents are small(ish) executable components hosted on one or more systems, separate from the build/CI and SUT servers. The build/CI server creates a job and dispatches it to the agent.

In the case of UI tests, the agents will spawn the system's application, either a desktop app or web browser, and navigate through the automated scripts. From there, the agent executes the task and reports back to the build server.

Agents give organizations the ability to scale out the coverage matrix. Agents can also run tasks in parallel, enabling large, long-running test suites to execute much faster.

(Telerik Test Studio Runtime Edition fills the “agent” role.)

System Under Test

The final piece in the diagram is the system under test (SUT). This very simplified diagram shows one unit; however, often, there are many components involved: web front ends, application servers, middleware such as Tibco or BizTalk, database servers and so on.

The SUT may be updated as part of the CI or scheduled build/execution process, or it may be a simpler model in which other team members update the SUT, as needed. Moreover, sometimes SUTs are in a shared environment, which causes additional complications for test data.

Leveraging Resources to Their Utmost

Getting to success in your test automation projects means getting all these components working well together, the earlier the better. Your team should view the automated deployment and execution of the tests, and everything around that process, as **the highest-value feature for the organization**. Being able to build, deploy, test and release your software with a metaphorical push of a button is an incredibly powerful concept!

You don't need to piece everything together at once. Start small and work from there. Here's one route you might take to get from zero to awesome:

- 1. Get tester/developer systems up and running:** You can write and run test suites locally as you build out the rest of your system.
- 2. Get your source control running:** This isn't optional. Period.
- 3. Get your SUT running:** Get a separate environment where you can totally control your SUT, even if it's a small VM to start with.
- 4. Get your build/CI running:** Use whatever tools with which your organization is already competent—don't try to reinvent the wheel. If the organization's not using a build/CI server, pick one that meets your needs. Start small with a simple build script, then wire up deployment of your SUT.

- 5. Create agents for execution:** Now find small VMs or old unused desktop systems and bring them into your environment as agents.
- 6. Create test jobs as necessary:** Configure scheduled jobs that pull the latest SUT and test suites, deploy as necessary and execute tests.
- 7. Rinse, lather, repeat:** Continue to evolve your tests and smooth out your automated build, deploy and execute processes.

Moving Forward

Infrastructure and the build/deploy/execute pipeline is critical to get in place as early as possible. Having the right environment in place lets you focus on the harder testing and domain problems.

Now that all the tools and people are in place, we'll next look at the practical aspects of getting rolling with your automation efforts.

Chapter Five

LOOK BEFORE YOU JUMP

By this point you should have a clear picture on the business-level problems you're hoping to solve, the team you'll need to build and the tools/infrastructure you'll need in place. Now it's time to stitch everything together and build some tests.

How's Your Process?

Before you jump, think carefully about your delivery process. I've already laid out a number of things about communication, tooling and infrastructure. Now's the time for you to sit back, as a team, and get serious about how your workflow will run.

Take the time to sit as a group and actually walk a feature through your flow. Figure out where you will include discussions about testing. Use a white board, Post-Its, notepads—whatever it takes to diagram the flow and all the discussions that should happen.

Clarify Expectations

As you walk through those discussions, see if there are any clarifications or modifications to expectations you've already worked on. Are there any new conversations you need to have with stakeholders and sponsors? Does your team have a good grasp on what's expected of them?

Make sure everyone's on the same page regarding initial expectations. Also, ensure everyone knows the expectations will likely be tweaked as you actually roll in to your work.

Start With a Pilot

If at all possible, teams should start automation efforts with a pilot, prototype or spike. Regardless of what you call it, carve out an area you can focus on for two weeks of full-time work. The goal of this pilot is to identify and resolve any problems your team may run into in the “regular” work. You'll be getting a feel for where you need to

have conversations, what the initial impact on velocity might be and places to tweak your infrastructure.

As with all pilots, setting the scope is critical. You can't bite off a huge amount of work. The pilot has to be small enough in scope that you can get work done, while discovering whether or not your tooling, process and communication will be effective.

Practical Flow: Putting Concepts to Work

Let's take a proposed feature addition as our pilot. We'll work it through our process and see where things work out or don't. We'll use adding in a feature that recommends additional related or interesting products to you as you're checking out in your shopping cart. The cart's built already; we're extending it to include the recommendations as part of the checkout flow.

Disclaimer: Of course this example will have lots of holes in it. It's not fleshed out; it's simplistic. Please cut me some slack here and focus in on the conversations around testing.

1. Envisioning:

Stakeholders need to understand the costs around testing, both manual and automated. Testers should be giving a **high-level**, rough idea of what the impacts of testing would be, so

that stakeholders can make an informed decision of the **total** cost of the feature.

“We'll need additional datasets, we'll have to modify the build flow to handle those, we'll need a modest number of new automated scripts, and we'll need to do quite a bit of exploratory testing. I'd guess we'll need two weeks' time, plus support from the DevOps folks for the data work.”

Things you might discover at this point:

- a. You don't have the right people in the conversations
- b. You're spending too much time getting too much detail and too many edge cases
- c. You may not be able to get as much support from your infrastructure folks as you thought

2. Early UX Design:

Testers should work with the UI/UX members to ensure everyone knows how to keep the UI as testable as possible.

“I'll need good IDs on these four areas of this screen so I can write good locators. What asynchronous actions will be happening on this page so that I can figure out the right conditional waits?”

Things you might discover at this point:

- a. Your UI may have technical limitations on what you can do to make it testable
- b. You may not have access to all the UI (think legacy or third-party systems)

3. Early Architecture Design:

Testers need to understand the high-level architecture to test effectively. This is also a great time to start discussions with developers around what testing will happen at the unit or integration level, and what tests will happen at the UI level. Discussions are generally still high-level and conceptual.

“The recommendation engine is a separate component that handles determining recommendations. It is sending results back to the cart via web services, right? Will you be testing the recommendation logic via those services? If so, then I can just write tests to ensure we're getting recommendations pulled back and rendered on the UI. I won't have to write tests to check that all the various combinations are creating the proper recommendations. I can also help you with those web service tests!”

Things you might discover at this point:

- a. Parts of the system may not adapt easily to automated testing. You may be relying on more UI tests than you would prefer—but you do what you can with the tools you have at hand.

4. Starting Work on the Feature:

Now's the time for the team to get into details. Discuss the specifics of the data you'll need. Talk about any backing APIs you will require. What edge cases and combinations get tested at which levels? Also, this is the time that you're able to start writing the scaffold of your tests, even before the UI is built—you can do that because you talked with the UX/UI people early in the game, remember?

“Can we work together to build a method to help me set parameters for recommendations? I could then use that as a setup for UI tests. I'll need some way to load these specific products into the recommendation database. Can I help you build up combinatorial/pairwise tests to run through the web service tests you're building? That way, we could cut the number of iterations you'd need to write, and I'd be able to focus on the main flows with the UI tests.”

Things you might discover at this point:

- a. Time constraints may force you to make hard choices about what parts of the feature to work on
- b. Code and architecture constraints may force you to do more UI testing than you'd prefer
- c. You may not have easy access to helper APIs, which may mean more UI code than you'd prefer

5. Working the Feature, Iterating on Feedback:

Hopefully you're at a point where development and testing is happening nearly concurrently. This means testers can get feedback to the developers very, very quickly. This enables teams to quickly fix issues EARLY in the game. This feedback happens best when teams are communicating directly to each other, not just relying on bug reports to percolate through the process.

“Hey, I realized we'd missed a couple edge cases with our test data. When I added them in, I found we're recommending motor oil instead of cooking oil when someone's buying a stir fry kit. I don't think that's what we meant. We need to modify the recommendation logic to pay better attention to an item's category.”

Things you might discover at this point:

- a. Your team's skills may not be at the level you need for effective automation. Take a tangent to shore up skills as needed. Additionally, make sure you're adjusting schedule/velocity predictions as needed to account for the slower learning.
- b. It's often hard for testers to keep pace with developers. You may need more testers, or you may need more collaboration between your developers and testers.
- c. Some members of the team may not value the fast feedback and increased collaboration. It's a culture change that's hard to adapt to, but it's worth the effort.

6. Rolling Test Suites into the Build Process:

You should be adding all your automated tests (unit, integration, UI) into your build process. This means configuring your CI builds and scheduled jobs to run your tests, as appropriate. Ensure your team has access to the reports they need.

“OK, so we're ready to add our integration and UI tests to the regularly scheduled jobs. All the UI and most of the integration tests are too slow to add to the CI build, but I think we should add these two relatively fast integration tests to the CI build to ensure we've got this part locked down. These five UI tests should go in the hourly UI job, and the remainder need to get added to the nightly job. Jane the stakeholder will see them showing up in her overall trend report, and we team members will see them in our detailed reports.”

Things you might discover at this point:

- a. You may need to expand your infrastructure to support the additional testing. More VMs, bigger build servers and so on
- b. You will likely need to adapt reports for all your data consumers—stakeholders, business sponsors, PMs and others

7. Maintaining the Suites:

Your automated test suites are a metaphorical living, breathing creation. You're going to have to spend time in the care and feeding of them. You'll need to fix them when the tests break (and take away lessons learned), update them with the systems change, and refactor or outright re-architect them on occasion.

“We've had two tests break last night due to changes in the helper APIs. That's roughly four hours to fix. We also had another four tests break due to changes in the checkout workflow. We think that's a day's work. Finally, we think we've got some duplication in a number of tests around the cart and recommendation engine. We want to take a half-day to pore over the tests and weed out any that are unnecessary.”

Things you might discover at this point:

- a. Your system may be more brittle than you expected—small changes may break multiple tests
- b. Your test suites may be brittle; learn to keep them as flexible and concise as possible

Learning from the Pilot

A pilot project is a gift—seriously! Dedicating time to trying out significant changes ensures you'll have the best possible chance to succeed.

Step back and have the entire team evaluate how things went. Some questions you might consider:

- Did the pilot appear useful to helping solve the business problems you identified up-front?
- Does the team have the right skills and ability to become adept at the technical aspects of the tools?
- Can your build/deploy process support the automation toolset?
- Does the increased communication help with the process, or is it too much of a cultural hindrance?

These are hard questions, but they're critical. Asking these gives your team the best chance to figure out if you're on the wrong track and need to completely re-assess; or if you're on the right track that can be adjusted to make everyone successful.

Chapter Six

AUTOMATION IN THE REAL WORLD

So here you are: ready and raring to get real work done. Hopefully, at this point, you're feeling excited about what you've accomplished so far. Your team has set itself up for success through the right amount of planning, learning and prototyping.

Now it's time to execute on what you've laid out. Remember: your best chance for success is focusing on early conversations to eliminate rework or waste, and being passionate about true collaboration. Break down the walls wherever possible to make the mechanics of automation all that much easier.

In the previous chapter, I tried hard to really hit the importance of collaboration and early communication as fundamental pieces of a successful project. Now it's time to focus on the actual mechanics of the work.

The next few sections dive in to areas I've found to be critical for pain-free, successful automation.

Developer Tester Pairing

Although discussion around pair programming has always focused on developers, pairing is a wonderful tool that can be used by all members of a team! ²

Developers and testers should work together on UI automation for a number of reasons, including:

- Developers know the ins and outs of the system. They can head off wasted efforts by the testers.
- Developers know what asynchronous or disconnected operations are taking place behind the UI.
- Testers can help identify valuable edge cases, environmental concerns or data combinations a developer might miss.
- Developers can handle the more complex coding issues while testers focus on high-value test problems.

Far too many people think pairing means working side-by-side for eight hours a day. NO! Pairing doesn't have to be high-stress,

²Read more about pair programming on its page at the [Extreme Programming site](#).

full-time effort. Spend as little time together as needed to solve a particular problem, or spend as much time together as the members feel comfortable with.

Also, pairing shouldn't be dismissed because teams aren't co-located. Technology such as Skype, Join.me, GoToMeeting and others have allowed me to pair with team members in different cities, states and continents. It's a matter of the team members committing to working hard toward great communication.

Pairing at its root is knowledge sharing. Investing to encourage and expect the team to pair up reaps rewards over the long run.

Maintainable Tests

Perhaps the most crucial concept to get straight at the start is the need to treat your test code like production code—**because test code IS production code!**

Delivery teams absolutely have to invest time in creating maintainable system code through well-established conventions like low complexity, readability, carefully thought dependencies, etc.

UI test automation code needs to be treated exactly the same way. You need to take the same approaches to simplicity, clarity and so

on, regardless if you're writing browser tests in WebDriver or creating them in Telerik® Test Studio®.

Teams that don't approach their test suites (or system code) this way are doomed to suffer lost time due to brittle suites. Those teams are also going to pay a heavy price when trying to fix brittle tests, due to their complexity. It's easier to be successful when you start out right by giving your test suites some love and maintainability.

Three great principles I've made use of over the years have helped me keep my UI tests as maintainable as possible.

- 1. DRY:** Don't repeat yourself. Copy/Paste development means you've got duplication of effort scattered all over the place. One thing changes, and you're forced to fix that change in multiple places. Pay attention to these areas in particular:
 - a. Locator definition:** Ensure your find logic, or element locators, are defined in one place and one place only. Either use the Page Object Pattern³ or a tool such as Telerik Test Studio that centralizes those definitions in some form of repository.
 - b. Actions:** Don't repeat actions or workflows. Move those into a reusable test or method. Think about a logon example: you want it defined one place and called from many.

³Martin Fowler has a nice write up on the Page Object Pattern.

2. SRP: Single responsibility principle: tests need to be granular and concise. They should test one thing. Scripts shouldn't conflate multiple test cases—something that's often abused in data-driven scenarios. For example, don't mix checking if products can successfully be added to a shopping cart with checking if those products also get correct recommendations. Those are two separate tests.

3. Abstraction: Abstraction⁴ is a programming idiom that enables you to push common or complex actions to another unit of code. A logon operation is the classic example for this. Many tests will need to log on to your system; however, you should write this action only once in a separate test or method, then call that test/method from all other tests to perform the action. Abstraction is also helpful because it hides the implementation details from the calling test or method. With the logon example, each test doesn't know how the logon occurs, only that it's successful or not. If the system changes the logon workflow, say from username/password to username/password/PIN, no other test would need to be updated—only the logon method.

You'll save your team tremendous amounts of time, frustration and grief if you focus on maintainability **at the start** of your UI test automation projects.

Backing APIs

Backing APIs, sometimes referred to as test infrastructure, are abstraction tools. They're perfect examples of how testers and developers can collaborate to leverage each other's best skills.

I've found not many testers have deep programming backgrounds, which is absolutely fine. Few testers understand how to create authentication headers to successfully call web service endpoints. Nor do many testers understand how to create secure, performant, reusable connection pools to a database.

That's all fine, because testers' skills lie in testing. Developers, on the other hand, generally do those tasks on a frequent basis.

Working together, teams can build an abstraction layer of a backing API that enables testers to very easily call methods for setting up prerequisite data or turning off features to make the system more testable.

The great thing about abstraction is you don't have to know (or care) how the API does its work. Are new users created via a web service, or direct insertion to the database? Don't know, don't care. Backing APIs let you abstract all that away so you don't worry.

Moreover, if the developers create better methods for accessing the system, say moving from a stored procedure call to a web service, the tests won't be affected at all.

⁴See [Wikipedia's definition of abstraction](#).

That's all fine and dandy, but what practical things should you look to do with a backing API? Here are a few things I use in every project I've worked on:

- **Data and Prerequisite Setup:** Don't use the UI to create data you need for tests. Hand that off to a backing API. (Starting out using the UI, then transitioning to a backing API is just fine.)
- **System Configuration:** How do you test CAPTCHA or other complex third-party controls? Don't. Work with the developers to create system-wide configuration switches that will let you shut these things off, or swap them out for simpler components.
- **Test Oracles/Heuristics:** It isn't enough to check the UI. You also need to check the layer where things are really happening: the database, file system and so on. Backing APIs are a great way to abstract out calls to the database for verifying records were created, updated or deleted.

Backing APIs don't have to be complex, and you should only build them out as you need them. Be very lean as you create them

Testable UI

Far too many systems are built without testability in mind. Architecture and coding design decisions impact testing at both the system and UI layers. System-level testability requires specific

architecture and design decisions. Testability at the UI layer can be a much simpler matter of adding in good element IDs where possible.

Some web technologies such as Ruby on Rails add ID attributes to elements by convention. Nearly every web stack from Rails to ASP.NET WebForms makes adding IDs to regular elements a snap.

Additionally, developers and testers working closely together can easily solve problems such as dynamically generated IDs that hinder testability—or don't render them at all.

For example, Telerik® Kendo UI® has a Grid control that doesn't include IDs by default:

16	Earth	Seldon Foundation	Asimov	Isaac
17	Europe	Top Notch Music Academy	Beethoven	Ludwig
18	New Earth	Blue Sun	Cobb	Jayne
19	Eastern	Relativity, Inc.	Einstein	Albert
20	Midwest	Guidepost Systems	Holmes	Jim
21	Scotland	Bravely Bravely, LLC	Knight	Robin

```

<tbody role="rowgroup">
  <tr data-uid="5ccb079-424c-4eb3-b0e1-c4e8a745d4f8" role="row">...</tr>
  <tr class="k-alt" data-uid="bdfc190f-7034-46f9-a72d-1e5e5189d42f" role="row">...</tr>
  <tr data-uid="b06f077e-a094-4af9-bff4-353aac6def46" role="row">...</tr>
  <tr class="k-alt" data-uid="9ddbc58c-3762-4377-96e2-b0d81a10bfa4" role="row">...</tr>
  <tr data-uid="66f157a4-ec5b-434d-a1b5-f04e026878cc" role="row">...</tr>
  <tr class="k-alt" data-uid="f0ca1908-2a14-473d-b49a-82b700cc87f" role="row">...</tr>
  <tr data-uid="82ea4dc2-9243-43a3-b83d-2b2cba898c1b" role="row">...</tr>
  <tr class="k-alt" data-uid="9b85ce71-ddcc-45ae-a534-24911227a23e" role="row">...</tr>
  <tr data-uid="da30c0a6-cf1e-4d67-a615-b678368a5c86" role="row">...</tr>
  <tr class="k-alt" data-uid="3ea0c122-c105-48fb-a2ae-94c1b4f09d99" role="row">...</tr>
  <tr data-uid="2fb1bde4-f1f2-4af7-8d41-2da5c7f8109e" role="row">...</tr>
  <tr class="k-alt" data-uid="c88224a7-bc35-427c-a4cb-ba5ffc9d2525" role="row">...</tr>
  <tr data-uid="2ba1e41-9105-4452-9e15-972712b09904" role="row">...</tr>
</tbody>

```

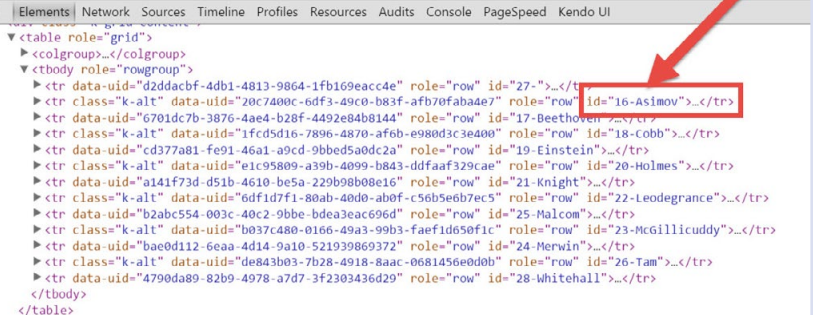

It's easy to add a bit of JavaScript to the Grid's definition, to create useful IDs that include data unique to each record:

```
dataBound: function(dataBoundEvent) {
var gridWidget = dataBoundEvent.sender;
var dataSource = gridWidget.dataSource;
$.each(gridWidget.items(), function(index, item) {
    //use next three lines for html ID attrib
    // with custom ID of database ID + lname
    var uid = $(item).data("uid");
    var dataItem = dataSource.getByUid(uid);
    $(item).attr("id", dataItem.Id + "-" + dataItem.LName);

    //Use this line to show html ID attrib with row #
    //$(item).attr("id", index);
});
$(".k-grid-add").attr("id", "create_btn");
},
```

Now the Grid's records each have a unique ID composed of the identifier from the database, plus the last name of the person on the row.

Earth	Seldon Foundation	Asimov	Isaac
Europe	Top Notch Music Academy	Beethoven	Ludwig
New Earth	Blue Sun	Cobb	Jayne
Eastern	Relativity, Inc.	Einstein	Albert
Midwest	Guidepost Systems	Holmes	Jim
Scotland	Bravely Bravely, LLC	Knight	Robin

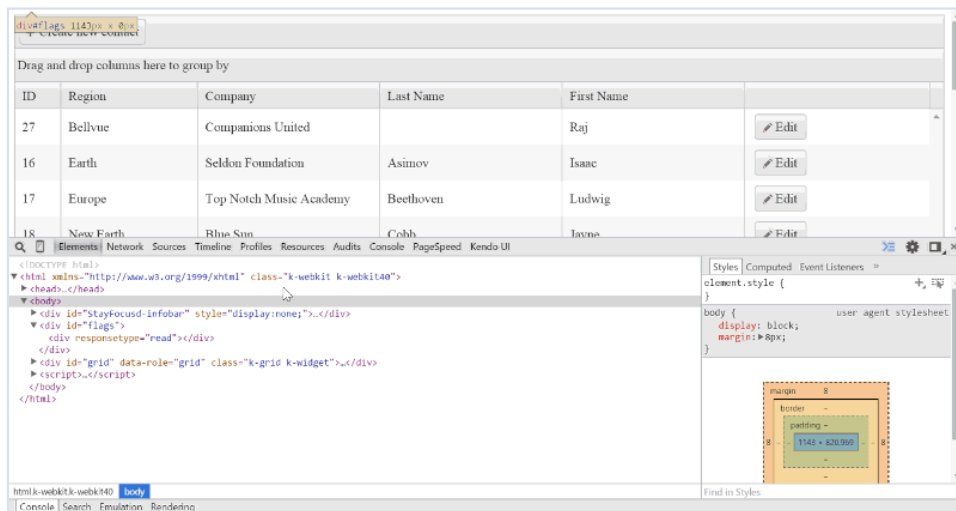


```
Elements | Network | Sources | Timeline | Profiles | Resources | Audits | Console | PageSpeed | Kendo UI
▼ <table role="grid">
  <tbody role="rowgroup">
    <tr data-uid="d2ddacbf-4db1-4813-9864-1fb169eacc4e" role="row" id="27">...</tr>
    <tr class="k-alt" data-uid="20c7a00c-6df3-49c0-b83f-afb70fab4e7" role="row" id="16-Asimov">...</tr>
    <tr class="k-alt" data-uid="6701dc7b-3876-4ae4-b28f-4492e84b8144" role="row" id="17-Beethoven">...</tr>
    <tr class="k-alt" data-uid="1fcd5d16-7896-4870-af6b-e980d3c3e400" role="row" id="18-cobb">...</tr>
    <tr class="k-alt" data-uid="cd377a81-fe91-46a1-a9cd-9bbed5a0dc2a" role="row" id="19-Einstein">...</tr>
    <tr class="k-alt" data-uid="e1c95809-a39b-4099-b843-ddfaaf329cae" role="row" id="20-Holmes">...</tr>
    <tr class="k-alt" data-uid="a141f73d-d51b-4610-be5a-229b98b08e16" role="row" id="21-Knight">...</tr>
    <tr class="k-alt" data-uid="6df1df71-80ab-40d0-ab0f-c56b5e6b7ec5" role="row" id="22-Leodegrance">...</tr>
    <tr data-uid="b2ab554-003c-40c2-9bbe-bdea3eac696d" role="row" id="25-Malcom">...</tr>
    <tr class="k-alt" data-uid="b037c480-0166-49a3-99b3-faefd650fc" role="row" id="23-McGillicuddy">...</tr>
    <tr class="k-alt" data-uid="bae0d112-6eaa-4d14-9a10-521939869372" role="row" id="24-Merwin">...</tr>
    <tr class="k-alt" data-uid="de843b03-7b28-4918-baac-0681456e0db" role="row" id="26-Tam">...</tr>
    <tr data-uid="4790da89-82b9-4978-a7d7-3f2303436d29" role="row" id="28-Whitehall">...</tr>
  </tbody>
</table>
```

This approach is obviously specific to this example; however, that's the beauty of this approach. Use your tools at hand to solve the specifics of the situation you're encountering. Maybe your IDs need part numbers, zip codes or something else. That's fine! Construct them as needed to get testable pieces in place.

A final piece about testable UI: You're not limited to just ID or other attributes. There are all sorts of things you can add to the UI to help testing. Think of flags you can add to handle complex asynchronous or queuing actions.

For example, the image below shows a new element being added to the page after a Create action completes. This gives you something additional to “latch” onto when an action is complete.



This particular example is again in Kendo UI as a new method in the control's DataSource definition:

```
requestEnd: function (e) {
    var node = document.getElementById('flags');
    while (node.firstChild) {
        node.removeChild(node.firstChild);
    }
    var type = e.type;
    $('#flags').append('<div responseType=\' + type + \'>');
},
```

While this example is specific to Kendo UI suite, again the underlying concept is the same regardless of the technology stack you're using.

Surviving Legacy UIs

All this discussion about modifying the UI to be more testable is wonderful, but what about when you're stuck with a UI that can't be changed? Maybe it's a legacy system that's got limited maintenance. Perhaps it's something built on top of a third-party system or platform, say SharePoint, Sitecore, or Orchard.

In those cases, you'll need to work hard to learn flexible approaches for building locators that work with the system/stack/platform you're using. In many cases, you'll find yourself having to fall back to locators based on convoluted XPath or JQuery selectors. Evaluate those locators as carefully as possible to ensure you're using the best locator possible.

Please note I specifically said “...the best locator possible.” In many situations you won't be able to get a perfect selector. In those cases, you'll need to become adept at using combinations of things such as IDs, CSS classes, name and other attributes, plus some XPath to scope down to what you need.

InnerText remains one of my favorite locator strategies, because it enables you to find things such as table rows using data that should be in that row. It's also very handy when you're simply not able to find other usable locator strategies.

Remember: Take the Long View!

Success in software, regardless of whether you're writing multithreaded database transactions or user interface functional automated tests, is all about the long view. Of course you have to write tests that are solid, high-value and correct, but you absolutely have to keep your eye on how useful those tests will be over time, and how costly they'll be to maintain.

Work hard to keep your tests simple, concise and flexible. Make use of the suggestions we've laid out here.

Chapter Seven

IMPROVING ON YOUR SUCCESS

As you move through (or finish!) your first project, you should be looking to get as much feedback as possible. Constant feedback and learning is vital to any organization. It's especially important when you're diving into a brand-new approach to delivering your software.

Are You Solving The Right Problems?

First and foremost, go back and ensure you're actually contributing value to solving the problems you decided you needed to address. Have your automated tests help shorten your release cycle? Are you better able to handle your cross-browser/cross-device testing?

Most importantly, do your stakeholders and sponsors feel they're getting better information that helps them make more informed decisions? Remember: your automation work (all work, really) should be tied directly back to concrete business value in some form or another.

Gathering on Feedback

There are many ways to gather feedback. I've found retrospectives are the best, easiest method to implement.

Retrospectives

Retrospectives take many forms and can be run many different ways. Esther Derby's and Diana Larsen's [Agile Retrospectives](#) is a great resource on how to run retrospectives. [The Art of Agile Development](#) from James Shore and Shane Warden has a terrific section on retrospectives we've used as a template for many organizations.

Regardless of how you run your retrospectives, you'll be gathering up concrete, actionable topics to address. Many of those may be negative issues; that's fine. Of course you'll find things to improve upon. Hopefully, you'll also find positive things you can do more of.

Make Sure Support is in the Loop

Early on I mentioned the importance of involving your support team in identifying addressable problems. Keep them looped in!

Make sure you're talking regularly with your support team, to see if there are aspects of your system you need to shore up with other UI automation. (The best thing, frankly, would be to have support in your retrospectives.)

Take advantage of your support team's role as the frontline contact with disgruntled, upset customers having problems with your system. Not only will you get great information from them to put back into your feedback loops, you'll also help ensure they're feeling validated and understood.

Sharing Feedback

It's not enough to simply identify what you should be doing more or less of; you need to share lessons learned as widely as possible. Getting information out to various teams can be challenging, no matter the organization's size. Look to leverage as many different communications channels as possible.

Team Lunches

If your team isn't already, start regular "lunch-and-learn" or "brown bag" sessions during which your team can discuss problems, share resolutions they've learned or simply brainstorm different approaches. These shouldn't be overly formal meetings. Keep them casual, but try to get some rough agenda a day or two ahead—asking participants to put Post-It notes on a board to share their ideas is perfect.

Build a Knowledge Base

Get your lessons learned, code snippets and solutions to problems in some form of searchable, preferably editable online system. Use a Wiki or an organizational Evernote account, but use **something!** Don't spend your time building something custom—there are far too many products already available.

These knowledge-base articles should cover topics such as how your UI automation project is set up, what it takes to build/run/maintain tests and how to solve particular challenges in your system. Have some tricky asynchronous actions? Write an article explaining how you resolved that. Did your developers build an API helper to handle custom queuing that locks the UI? Write an article talking about how to use that method.

Trumpet Your Successes

At this point, you've likely made some significant progress toward solving the problems you identified at the start of your effort. Don't sit on those successes; spread the word to the rest of your organization so they'll adopt and adapt the approaches you've used.

Get five or ten minutes on the agenda at organizational and cross-departmental meetings. Or, write some short articles for a company newsletter. Don't dive too far into the weeds; just set the hook. Your true purpose is to **get people outside your team coming to visit or talk with you**. You want curious co-workers to see the improvements you've made, as well as some of the struggles you've gotten through.

Are the Stakeholders Happy?

It's important to make sure your stakeholders approve of the time, effort and money you're investing in your UI automation suites.

Are they seeing more useful information regarding the state of the system? Are they able to make better decisions about when to release and when to hold back?

If the answer's "yes," your effort has been well placed!

Keep Working, Keep Learning

We hope you've found this handbook helpful and a great use of your time. Our intent hasn't been to lay out "best practices" (**THERE ARE NONE!**) or specific answers to specific problems. Every team's situation is different environmentally, technically and culturally.

Instead, we've tried to lay out broad guidelines that are common across every UI automation project we've worked on:

- Be positive about your work: invest the time to clearly state the problems you're trying to solve
- Do the hard up-front work of planning and setting expectations
- Use pilots or spikes to quickly validate and adjust your plan
- Jump in and get building your tests, focusing on high-value cases

- Be disciplined about constantly refining your approaches through constant feedback
- Most of all, never stop learning and adjusting

Teams all over the globe have had tremendous success in UI automation, despite its myriad challenges. Your team certainly can be among those!

ABOUT THE AUTHOR



Jim Holmes

Executive Consultant at Pillar Technology

Jim is an Executive Consultant for Pillar Technology. He's also the owner/principal of Guidepost Systems. He has spent time in LAN/WAN and server management roles, as well as many years helping teams and customers deliver great systems.

Jim has worked with organizations ranging from startups to Fortune 100 companies, to help them deliver better value to their customers. Jim has been in many different environments but greatly prefers those adopting practices from Lean and Agile communities.



[Watch video](#) 

Telerik Test Studio

Test Studio is a powerful test automation tool that helps you create maintainable test suites for a wide range of platforms and browsers. It inspires testers and developers to collaborate on building high-value test automation and increase team velocity.

www.telerik.com/teststudio