UiPath Coding Best Practices

Contents

Overview

This guide outlines a series of UiPath best practices designed to enhance code quality, maintainability, and project scalability. It addresses guidance applicable to both standard automation projects and those built on the Robotic Enterprise Framework (REF). While REF-specific recommendations are identified where appropriate, all general UiPath best practices are inherently applicable to REF implementations. However, not all REF-specific practices are universally relevant to non-REF projects—think of it as: all squares are rectangles, but not all rectangles are squares.

Activity Naming Standards

All activities should be assigned clear, descriptive, and context-specific names. This practice improves searchability within UiPath Studio, making it easier to pinpoint activities during debugging or maintenance. It also accelerates troubleshooting by preventing ambiguity—no more asking, “What does this activity actually do?”

Thoughtful naming conventions play a vital role in disaster recovery as well. In scenarios where contextual screenshots are missing, no Process Design Document (PDD) exists, or the PDD is outdated, meaningful activity names can help engineers reconstruct the automation’s intent and flow

For example, if you create a new sequence to log into an application the default name of the sequence is just “sequence.” Rename this to something more relevant such as “Log in sequence for app xyz.”

Use of Annotations

Annotations in UiPath serve the same purpose as comments in traditional programming languages, offering contextual information that improves code readability. Unlike the "Comment" activity—which can introduce visual clutter and obscure the automation flow—annotations are embedded directly within activities and flow control structures such as sequences and flowcharts.

They allow developers to capture detailed insights about logic, configurations, or exceptions right at the source, helping to answer common questions like “What does this flow do?” or “Why is this activity configured this way?”

When combined with meaningful activity names, annotations significantly enhance clarity and maintainability—especially in handover or recovery scenarios. Always ensure annotations are pinned to keep them visible during code reviews and troubleshooting.

Take, for example, a standard login sequence. Authenticating a user often involves multiple steps—from launching the application to navigating login screens—all of which may be encapsulated within a single sequence.

Each activity within that sequence should include a clear annotation describing its specific function, no matter how trivial it may seem in the moment. For instance, an annotation like “Double-clicking the application icon on the desktop” provides immediate clarity—not only indicating which UI element is being interacted with but also specifying the action type (double-click vs. single-click), which can affect both performance and troubleshooting.

Sequences, Flowcharts, and Visual Clarity

Visual clutter often signals weak coding standards or limited experience. Excessive nesting—especially with flowcharts—can render workflows difficult to read and debug. Limit nesting of flowcharts to a maximum of two levels. Even with thorough annotations, tracing logic through deeply nested flowcharts is unsustainable for anyone but the original developer (and only if they still recall the structure).

This problem worsens when paired with missing annotations and vague activity names. In the absence of a current Process Design Document (PDD), engineers may have to inspect each selector individually to determine function—an inefficient and error-prone approach. Only nest flowcharts when necessary.

Sequences, when used properly, offer a clean way to compartmentalize logic. For example, grouping all login-related activities into a sequence named “Login sequence for app xyz” provides structure and improves readability. Annotating these sequences adds context, especially valuable when isolating complex logic or troubleshooting issues.

That said, sequences can also be overused. Avoid nesting them beyond two levels. This frequently occurs when default containers—such as those inserted by activities like Check App State—are left untouched. Developers often drop prebuilt sequences into these shells without optimizing for clarity. In most cases, a single, well-structured sequence is more effective than unnecessary layering.

Overall, the goal is to reduce complexity and increase legibility for both developers and stakeholders. Clean structuring and minimal nesting help future engineers maintain the project efficiently and answer key questions like “What exactly does this do?” with minimal guesswork.

Process Logging Best Practices

Effective logging during process runtime is essential for identifying faults, failures, and business exceptions. Logging can generally be categorized into two types: "Where am I?” and "What am I?”

“Where am I?” Logs
These are informational entries placed at the beginning and end of major workflows or compartmentalized sequences. Examples range from simple statements like “Began login flow” to more timestamped entries such as “4/16/25 9:19 AM – Began searching accounts.”

When paired with a similarly structured log at the workflow’s conclusion, they allow you to estimate performance metrics like average processing time or total queue duration. In the event

of a bot failure, these logs help isolate the fault to a specific sequence, particularly useful when Orchestrator logs lack detail.

“What am I?” Logs
These logs capture and validate key data points, often for diagnostic comparison. For example, logging an account number when an exception occurs enables teams to review whether the issue stems from edge-case data or a flaw in the automation logic, business rules, or technical setup. These logs also offer a reference point against source-of-truth systems.

Recommendations
Include a “Where am I?” log at the start of every significant workflow or at decision-tree branches to trace logic flow. Use “What am I?” logs to capture critical data, such as account numbers, dollar amounts, or outcomes like “processed” vs. “skipped.” Together, these logging techniques provide visibility, speed up troubleshooting, and help future-proof your workflows for operational support.

Universalizing Selectors

When automating UI elements—such as buttons, fields, or windows—selectors should be cleaned or universalized wherever possible. Recorded selectors often include user- or session-specific data, such as account numbers or usernames, which makes them fragile and difficult to maintain. To increase reliability and portability, replace these unique values with wildcards (e.g., *) or adjust the selector to focus on stable HTML attributes.

For example, in Epic, a window selector may include personalized details like PRD – FirstName LastName – 1234567788, effectively locking the process to a specific user profile. Editing the selector to a form like PRD-* strips away volatile identifiers, allowing the bot to function across multiple accounts without modification.

Universal selectors are generally more resilient and support credential rotation without requiring updates. However, not all selectors can be universalized. In cases where the selector remains unstable, consider alternative strategies such as anchor-based targeting, image recognition, or leveraging reliable parent-child relationships in the UI tree.

Variables and Scope

In UiPath, variable scoping functions similarly to public and private variables in traditional programming. Every variable must exist within a container—such as a sequence, flowchart, or other control structure—which defines where that variable is accessible. Activities within the same container can reference the variable, while those outside cannot.

Because UiPath structures workflows hierarchically, variable scope also follows a top-down model. Variables defined closer to the project root behave like global variables, accessible across the entire automation. Mid-level variables are visible within their specific branches, while those scoped at the lowest levels are available only within their immediate containers.

Given this hierarchy, variables—especially those holding critical data—are typically scoped as high as possible to ensure accessibility throughout the process. Lower-scoped variables should only be used when isolating logic or storing temporary data that differs in structure or type from higher-level variables. While this may differ from traditional development practices that favor minimizing scope, UiPath's architecture favors broader visibility to support seamless access across nested activities.

Naming Consistency: Project, Package, and Process

Ensure that the *project name*, *package name*, and *process name* are identical across all environments. This consistency simplifies traceability—making it easier to map a faulted process back to its corresponding codebase for troubleshooting and maintenance.

Avoid using spaces in names. When publishing to Orchestrator, UiPath Studio automatically replaces all spaces with periods (.), which can lead to inconsistencies and confusion. While there's no current setting to override this behavior, adopting a space-free naming convention prevents formatting issues at publish time.

Common naming styles include PascalCase (e.g., ClaimProcessorBot) and snake_case (e.g., claim_processor_bot). Select one and apply it uniformly to maintain clarity and support automation governance.

Robotic Enterprise Framework (REF)

The Robotic Enterprise Framework (REF) is designed to bring resilience and standardization to automation projects, especially those prone to instability. While robust, REF introduces its own set of quirks.

A core requirement of REF is its integration with the UiPath queueing system. For more reliable input/output handling, consider using an SQL database as the primary source of truth. During the initialization phase, clear the UiPath queue and use the *Bulk Add Queue Items* activity to import

fresh data directly from a SQL query or file export. This ensures a one-to-one mapping between source data and queue items and avoids the need for separate dispatcher bots—a model commonly recommended by UiPath but often unnecessary when the performer can manage both tasks effectively.

REF projects also rely on an external configuration file (Config.xlsx) to control framework behavior. A critical field in the Constants tab in this file is “ShouldMarkJobAsFaulted,” which should always be set to “TRUE” to ensure proper fault logging and visibility into runtime errors.

Invoke Workflow vs. Custom Libraries

The Invoke Workflow activity, while useful, can complicate debugging—especially when workflows are invoked either from local project files or externally from Orchestrator. This duality often makes it difficult to trace what’s being executed, particularly when an invoked file doesn’t reflect the latest changes or exists in multiple variants across projects.

While Invoke Workflow is commonly used to reuse code, a more scalable and maintainable alternative is building a **custom library**. Libraries offer centralized code management within a single project, making activities easier to distribute across environments via Orchestrator. Enhancements made in the library are inherited by all dependent processes, promoting standardization and consistency.

In contrast, invoked workflows often become fragmented and inconsistent over time, introducing duplication and technical debt. Replacing them with reusable components from a custom library not only simplifies version control but also reduces the time spent troubleshooting and maintaining shared logic.

Whenever possible, opt for a custom library or template—it’s a long-term investment that pays off in clarity, efficiency, and stability.

Use Incognito Mode by Default

When developing browser-based automation, always configure the bot to launch the browser in **incognito mode**. This prevents the browser from retaining cookies, cached data, or session states between runs, ensuring a clean slate every time the automation starts.

This is especially important in fault recovery scenarios, where retained session data could cause a bot to misinterpret the site’s state and enter a repeated error loop. This risk is particularly high in Robotic Enterprise Framework (REF) projects, where unintended persistence can prevent proper recovery and inflate exception handling overhead.

By building with incognito mode from the outset, you avoid the need to write custom logic for login detection or session resets. It streamlines development, improves reliability, and reinforces consistency across runs—saving time while reducing complexity.