ChatGPT

Best Practices for Writing Gherkin BDD Feature Files

Behavior-Driven Development (BDD) with Gherkin helps teams define software behavior in plain language. Gherkin feature files serve as living documentation and acceptance tests for desktop, web, mobile, and even embedded software. Below, we outline general best practices for writing clear, effective Gherkin scenarios, followed by specific considerations for embedded systems (from bare-metal firmware to multi-service embedded Linux devices).

General Best Practices for Gherkin Scenarios

- **Focus Each Feature on a Distinct User Need:** Organize your Gherkin by having one feature per file, aligned with a specific capability or requirement that delivers value to the user [1]. Provide a short **Feature** description or *narrative* (e.g. *"As a <role>, I want <feature>, so that <benefit>"*) to clarify the feature's intent [2]. This makes it easy to find features and ensures the team understands the purpose from the outset.

- **Keep Feature Files Manageable:** Avoid bloated feature files. Limit the number of scenarios per feature to a reasonable count (about a dozen or fewer) [1] [3]. If a feature is complex, consider breaking it into sub-features with separate files rather than one monolithic file. This keeps each file focused and prevents readers from wading through hundreds of lines to find a scenario.

- **One Behavior per Scenario:** *The Cardinal Rule of BDD:* each scenario should cover exactly one behavior or business rule [4] [5]. In practice, this means one **When-Then** pair per scenario – a single action and its outcome(s). If you find multiple unrelated "When… Then…" sequences in one scenario, that's a sign it should be split into separate scenarios [4] [6]. Keeping scenarios atomic makes failures easier to triage (a failing scenario pinpoints a specific behavior) and avoids confusion about what's being tested.

- **Follow the Given/When/Then Flow Properly:** Structure each scenario in the natural **Given – When – Then** sequence. **Given** steps set up the initial state or context, **When** steps perform the single action, and **Then** steps observe outcomes. Do not intermingle these or repeat them out of order (e.g. no extra When after a Then in the same scenario) [4]. This strict ordering improves readability and enforces the "one scenario, one user procedure" rule (multiple When-Then pairs indicate a problem [5]). Use **And**/**But** for additional conditions or outcomes *only* if they pertain to the same Given, When, or Then context.

- **Write in a Declarative, User-Centric Style:** Scenarios should describe *what* the system does, not *how* it does it [7]. Use language from the user's perspective or domain language – avoid mentioning UI elements, technical details, or internal implementation. For example, write **"When Bob logs in"** instead of step-by-step UI actions like *"Given I go to /login, When I enter username and password and click Login"* [8] [9]. The Gherkin should read like a requirement or acceptance criterion, abstracting away low-level clicks and keystrokes. This declarative approach makes scenarios shorter, clearer, and more robust to UI changes [10] [11]. Ask yourself: *"Would this scenario's wording need to change if the UI or implementation changes?"* If yes, rephrase it to be more abstract [12].

- **Keep Scenarios and Steps Concise:** Aim for brevity and clarity in each step. Each scenario should ideally have only 3-5 steps (and rarely more than 7) [13]. Long scenarios are harder to understand and maintain. Likewise, each step line should be short (commonly under 120 characters [14]) and state one clear intent. If a scenario is getting lengthy or has many "And" steps, consider whether some details are unnecessary for understanding the behavior, or if the scenario should be broken up. Prefer high-level phrasing that collapses multiple low-level actions into one intent (for example, **"When the user correctly fills in the registration form"** instead of listing every field input [15] [16]). Keeping scenarios terse helps readers grasp the behavior at a glance.

- **Avoid Compound Actions in a Single Step:** Each Gherkin step should represent a single action or check. Do not use conjunctions like "and" *within* a step to string together multiple actions or assertions [17]. For example, instead of a step that says, *"Given the user is logged in and has an item in the cart,"* split it into two steps: **"Given the user is logged in" And "the user has an item in their cart."** This makes steps more reusable and clearer [18] [19]. The only time **And** or **But** should appear is as a separate keyword for additional steps, not as a way to cram two behaviors into one step line.

- **Use Meaningful Titles for Features and Scenarios:** Write descriptive titles that summarize the behavior being tested. A scenario title should convey the situation and expected outcome in one line [20]. For example, **"Scenario: User sees an error message on incorrect login"** is clearer than a vague title like "Scenario: Login error." A good title allows someone scanning the file (such as a developer or manager) to understand the essence of the scenario without reading the steps [21] [22]. Similarly, the Feature title should be concise yet indicative of the feature's intent (avoid overly generic titles or extremely long ones). Consistent, clear titles make the suite more navigable and maintainable.

- **Use Backgrounds and Scenario Outlines Wisely:** Gherkin provides structures to reduce repetition, but use them with care:

- **Background:** If every scenario in a feature file shares the same initial context, put those common Given steps in a **Background** section. These steps will run before each scenario [23]. For example, if all scenarios require the user to be logged in first, the Background can handle that login Given. This DRYs up the scenarios and improves readability [24] [25]. *However,* avoid putting too many steps in the Background – if a reader must scroll up and combine five setup steps with each scenario to understand it, consider if all are truly needed globally. Keep background setup minimal so scenarios remain understandable in isolation.

- **Scenario Outline:** Use Scenario Outlines for data-driven scenarios where the same steps repeat with different inputs [26] [27]. This is preferable to duplicating scenarios that only differ in values. Define the scenario steps with placeholders, and supply examples in an **Examples** table. For instance, an outline can cover a behavior like **"Given a customer with <status> membership, When they request a refund, Then <outcome>"** with an Examples table of `<status>` vs `<outcome>` pairs. This technique handles variations compactly (even replacing the need for an "Or" in steps) [28] [29]. As a best practice, keep each outline focused – don't try to cover completely unrelated permutations in one outline; if it gets too large or complex, break it into separate scenarios or outlines for each case.

- **Maintain Good Grammar and Consistent Style:** Treat feature files as documentation. Use proper spelling and grammar in steps, and maintain consistency in phrasing. Gherkin keywords (Feature, Scenario, Given, When, Then, And, But) should be capitalized, while the rest of the step sentence should not be (unless it contains a proper noun) [30]. Write steps in third-person present tense ("the user *does X*" / "the system *does Y*") for clarity [31] [32]. For example, **"Given the home page is displayed"** is better than *"Given I navigated to home page"* (which mixes perspective and tense). Consistent tense (preferably present for actions and outcomes) makes scenarios read as timeless behaviors rather than scripts tied to a specific moment [33] [34]. Paying attention to these details keeps the Gherkin readable and professional.

- **Tag Scenarios for Organization and Traceability:** Leverage Gherkin **tags** (the `@tag` syntax) to categorize and manage scenarios. Tags are powerful for grouping tests, controlling execution, and mapping tests to requirements [35] [36]. For example, you might tag scenarios with components or feature areas (`@login`, `@payments`), with priority (`@critical`), or with requirement IDs (`@REQ-123`) to trace coverage [37] [38]. In regulated environments, tags can link scenarios to specific requirements for coverage reports [39]. You can also use tags to indicate test type or scope (e.g. `@smoke`, `@regression`) and to include/exclude certain tests in different test runs. Additionally, if some scenarios are intended for manual execution or cannot be automated, mark them with a tag like `@manual` so they can be filtered out in automated test pipelines [40]. Establish a standard tag naming scheme (all lowercase, hyphen-separated words, no duplicates) for consistency [41].

- **Make Scenarios Automation-Friendly:** Write your Gherkin steps such that they are easy to bind to automation code. Use parameters in steps to generalize actions instead of hard-coding values. For instance, `When "Alice" logs in with valid credentials` can be implemented to accept any username, rather than having separate steps for each user [8] [42]. In many BDD frameworks, putting variables in quotes or angle brackets will pass them to step definitions (e.g. `When "<username>" logs in`) [43]. Reusing step definitions for variable inputs keeps the step vocabulary small and avoids duplicating automation code. Ensure each step has a clear pass/fail check (each step should verify the state before and after the action, if applicable, to catch failures at the right step) [44] [45]. By writing steps that map cleanly to code, you facilitate linking these scenarios with unit tests or API calls (for example, using Python to script interactions). The goal is to integrate BDD scenarios into your CI pipeline – for example, run them on every build against the application (or even on target devices in the case of hardware) to catch regressions early [46]. If certain setup or data seeding is required for automation, hide those details in support code or hooks rather than in the Gherkin text, so the scenarios remain high-level. In summary, think about how you'll implement the steps as you write them, and favor straightforward, action-oriented sentences that can drive test code.

- **Collaborate on Scenario Creation:** Develop Gherkin scenarios as a team (developers, testers, product owners) rather than in isolation. This ensures the scenarios capture a shared understanding of expected behavior [47] [48]. Team review can also remove unimportant details (developers might spot overly low-level steps, or business stakeholders might spot missing cases). Writing scenarios before or alongside development (e.g. during refinement or via Example Mapping workshops) helps validate requirements and guide implementation. Once everyone agrees a scenario accurately describes the acceptance criteria, it can be automated and linked to the code. This collaboration prevents scenarios from being either too technical or too ambiguous, and it cements BDD as a **conversation tool** rather than just test documentation.

By following these general practices, your feature files will be readable, maintainable, and effective at capturing the system's behavior. In the next section, we address how to apply these principles to embedded software projects, which have their own nuances and constraints.

BDD for Embedded Systems (Bare Metal and Embedded Linux)

Applying BDD to embedded systems can greatly improve clarity of requirements, even though the environment is lower-level than typical web or desktop apps. Gherkin scenarios can describe the expected behavior of firmware or devices in a human-readable way, aligning developers, testers, and other stakeholders. Here we cover best practices specific to embedded contexts – from small microcontroller-based applications to complex embedded Linux devices with multiple services:

- **Define Behaviors at the Interface Level, Not Hardware Internals:** Just as with other software, describe *what* the embedded system should do in observable terms, avoiding direct references to registers, memory addresses, or low-level interrupts. For a bare-metal application, focus on external signals or system responses. For example, **"When the temperature sensor reads above 80°C, Then the overheat LED turns on"** is a clear, testable behavior. This is preferable to detailing the ADC register values or bit toggling in the scenario. BDD scenarios for embedded systems should express the device's response to inputs or conditions as a user or integrator would see it (LEDs, display output, actuators activated, messages sent, etc.), thereby serving as an executable specification of the firmware [49] [50]. Keeping scenarios at the interface level also means that if the underlying implementation (e.g. sensor driver or RTOS timing) changes, the Gherkin steps remain valid as long as the outward behavior is consistent.

- **Use Off-Target and Simulation Testing Where Possible:** Because running Cucumber or Robot frameworks directly on a tiny microcontroller isn't always feasible, a common practice is to run BDD scenarios on a host machine or simulator that can exercise the embedded code. For example, you might run the device logic in a simulation (or use a hardware-in-the-loop setup) where step definitions toggle inputs and observe outputs via an interface or API. This allows using higher-level languages (Python, etc.) to automate scenarios against the embedded code. Ensure your embedded code is structured to support this (e.g. dependency injection for hardware interfaces, so they can be stubbed in tests). By using BDD in off-target tests, you can verify embedded module behaviors quickly and consistently as part of CI, resulting in "known-good" components before deploying to real hardware [51]. For time-sensitive or asynchronous behavior, frameworks like Robot or custom test harnesses can wait for events (possibly using techniques like polling or callbacks) to verify outcomes. The key is to treat the embedded system or module as a black box that you stimulate and observe according to the scenario steps.

- **One Behavior per Scenario – Even for Firmware:** The single-behavior rule is crucial in embedded testing too. Resist the urge to script a long sequence of device operations in one scenario (e.g., power on, then connect to network, then send data, then receive update). Instead, break them into separate scenarios focusing on each distinct feature or state. For instance, have one scenario for power-up behavior (e.g. **“Device boots into normal operation within 5 seconds”**), another for network connectivity, another for data logging, etc. If a higher-level use case involves multiple stages, use multiple scenarios to cover each stage independently. This yields more pinpointed tests; when a scenario fails, you know exactly which aspect of device behavior is broken. It also encourages testing at the interface of each component or state change rather than one giant end-to-end with many possible failure points. In practice, adhere to one When-Then pair per scenario on devices just as you would in a web app (for example, do not combine “When the button is pressed, Then the LED lights, When it is pressed again, Then it turns off” in one scenario – split that into two scenarios) [6]. This discipline keeps your embedded BDD scenarios simple and **behavior-focused** rather than turning into complex test scripts.

- **Treat Each Service or Component as a Mini-Application (Embedded Linux):** For embedded Linux systems with multiple custom services or daemons, it’s effective to write BDD scenarios at two levels – the service level and the system level. Treat each service as its own application for BDD purposes: create feature files that focus on the behaviors of that individual service in isolation. For example, if you have a *SensorService* and an *UpdateService*, you might write a **Feature: Sensor Service Data Collection** with scenarios describing how SensorService handles sensor inputs and stores data (independent of other services). Each scenario would focus on SensorService’s response to certain inputs or conditions (simulated in the test). This approach aligns with microservices testing strategies where **each service is tested for its own behavior without needing the whole ecosystem running** [52]. You can use stubs or test doubles for interactions between services in these service-level tests, or use the actual IPC/communication if it’s easily controlled. By giving each service its own BDD specs, teams working on those services can understand and verify their functionality clearly.

- **Write End-to-End Device Scenarios for Integrated Behavior:** In addition to service-level tests, have feature files that describe the behavior of the entire device as a whole – essentially high-level acceptance tests spanning multiple services. These system-level scenarios treat the whole device as the "actor." For example: **“Feature: Device Power Management”**, with a scenario **“When the device is running on battery and battery falls below 10%, then all services enter low-power mode and a warning is logged.”** This scenario would implicitly involve perhaps a PowerMonitor service, the individual services responding to power events, and a logging service. At the Gherkin level, it’s just describing the end result observable at the device level (services slowed or stopped, log entry present). Such end-to-end scenarios ensure that the interactions between services achieve the desired result for the user or system integrator. It may be wise to tag these differently (e.g. `@integration` or `@device`) to distinguish from service-level scenarios. Keep these integrated tests focused as well – each scenario should still cover one high-level use case or condition on the device. If an integrated use case has multiple outcomes or branches, consider separate scenarios for each branch. Essentially, **design your BDD tests in embedded Linux similar to how you would for microservices: service-specific behavior tests plus a layer of contract/integration tests** to verify the services work together [53] [54]. This two-tier approach prevents confusion and makes failures easier to diagnose (you’ll know if a failure is within a service or in the integration between services).

- **Use Tags and Naming to Manage Embedded Test Variants:** In embedded projects, you might have scenarios that only apply in certain modes or hardware variants (for example, a scenario that only applies to devices with a specific sensor present, or only in a bare-metal build vs. an RTOS build). Use tags to mark these (e.g. `@HWv2`, `@linux-only`, `@baremetal`) so you can include/exclude them in test runs as appropriate. Similarly, clearly name your scenarios to reflect the context. For instance, **“Scenario: [Bare Metal] Motor shuts off on overload”** vs **“Scenario: [Linux] Service recovery after crash”**. The bracketed notes in titles or use of tags can help readers instantly know the applicable context. This is especially useful if you maintain one unified BDD suite for both a device’s firmware and its higher-level software – though you might also choose to keep them separate. The goal is to avoid confusion about which scenarios apply where. Proper tagging and organization will let you generate reports per subsystem or run only the relevant tests on certain targets.

- **Adapt Test Automation to the Hardware Environment:** Getting BDD scenarios to execute against embedded hardware requires creativity, but it’s achievable with the right tools. For small bare-metal systems, you might use a unit test framework (like Google Test) on host-simulated code for logic, combined with hardware-in-the-loop tests for actual device behavior. For example, testers have used cameras and OCR to verify screen outputs and robotic actuators to press buttons on devices under test [55] – all coordinated by high-level BDD scenarios. For embedded Linux devices, you can often run test code on the device or control the device via SSH/serial from a test runner PC. In either case, structure your step definitions to interact with the system through public interfaces: CLI commands, API calls, hardware I/O, etc. For example, a **Given** step might deploy a certain firmware state or ensure services are started; a **When** step might simulate an input (sending a message to a message queue or toggling a GPIO); a **Then** step might read a log file, check a hardware sensor output, or call a status API to confirm the outcome. Ensure that each scenario cleans up or resets state as needed (you might use Background or hooks to reset the device or test environment between scenarios). The specifics will vary, but the best practice is to integrate the BDD tests into your development workflow – for instance, running a subset of fast scenarios on every commit via simulation, and perhaps running full device tests on actual hardware nightly or in CI. This keeps the embedded BDD scenarios useful and executable, not just documentation.

- **Maintain Behavioral Specs alongside Requirements:** In embedded projects, requirements and design might be tracked in documents – try to keep your Gherkin features aligned with those. For example, if you have a requirement “The device shall enter safe mode on fault X,” ensure there’s a corresponding feature or scenario covering that behavior. BDD scenarios can then double as acceptance criteria for those requirements. In the case of devices comprised of multiple services, treat the BDD feature files as a unifying specification of how those services collaborate to fulfill requirements. This may even influence your design: by writing scenarios first (BDD is about *defining behavior before implementation*), you can identify how you expect the services to interact. If writing scenarios uncovers ambiguity (say, it’s unclear which service should handle a certain error), that’s a valuable discussion to resolve with your team. Thus, in embedded development, BDD can drive a more **behavior-focused design**, ensuring that even low-level software is guided by clear, testable expectations.

In summary, BDD can be applied across the spectrum of embedded systems development [56] [57]. By writing Gherkin scenarios for each component and for the integrated device, you create a *living documentation* of the embedded software’s intended behavior, accessible to engineers and stakeholders alike. This not only helps in testing but also in design and communication. Whether it’s C++ or Rust firmware with Google Test, or higher-level TypeScript/Robot tests on an embedded Linux device, the Gherkin best practices – clear intent, one behavior per scenario, abstraction of details – remain the same. Adhering to these practices will lead to BDD feature files that are clear, maintainable, and effective in guiding development and verifying that the software (or device) behaves as expected [49] [50].

Sources:

- Andy Knight, *“BDD 101: Writing Good Gherkin.”* Automation Panda Blog [4] [58]
- Cucumber.io, *“Writing Better Gherkin.”* (Official Cucumber Documentation) [7] [10]
- Jane Orme, *“Top 5 Gherkin Tips for BDD.”* Bluefruit Software Blog [20] [43]
- Kenan Rhoton, *“Gherkin Best Practices – 8 Tips.”* Redsauce.net [23] [13]
- SmartBear (CucumberStudio), *“Best practices for scenario writing.”* (Documentation) [5] [3]
- Steve Branam, *“Example of BDD Style TDD For Embedded Software.”* Embedded Online Conf 2024 [59] [50]
- Ronika Kashyap, *“Integrating BDD with Microservices Testing.”* DEV Community [52] [54]

---

[1] [4] [14] [26] [27] [28] [29] [30] [31] [32] [33] [34] [41] [58] BDD 101: Writing Good Gherkin | Automation Panda
https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/

[2] [6] [13] [15] [16] [17] [18] [19] [23] [24] [25] [44] [45] Gherkin best practices | 8 tips
https://www.redsauce.net/en/article?post=gherkin-best-practices

[3] [5] [47] [48] Best practices for scenario writing | CucumberStudio Documentation
https://support.smartbear.com/cucumberstudio/docs/tests/best-practices.html

[7] [8] [9] [10] [11] [12] [42] Writing better Gherkin | Cucumber
https://cucumber.io/docs/bdd/better-gherkin/

[20] [21] [22] [35] [36] [37] [38] [39] [40] [43] [46] [55] Top 5 Gherkin tips for writing BDD test scenarios
https://bluefruit.co.uk/quality/gherkin-bdd/

[49] [50] [51] [59] Example of BDD Style TDD For Embedded System Software - presented by Steve Branam
https://embeddedonlineconference.com/session/Example_of_BDD_Style_TDD_For_Embedded_System_Software

[52] [53] [54] Integrating BDD with Microservices Testing: A Practical, Human-First Walkthrough - DEV Community
https://dev.to/ronika_kashyap/integrating-bdd-with-microservices-testing-a-practical-human-first-walkthrough-598c

[56] An introduction to Behaviour Driven Development (BDD ... - YouTube
https://www.youtube.com/watch?v=4kjofLTXenA

[57] Six Case Studies with a Combination of Testing Techniques - Part 3
https://www.infoq.com/articles/testing-techniques-microservices-use-cases/