ChatGPT

GUT Testing Guidelines for Godot 4.5 Poker Game

GUT (Godot Unit Test) is a GDScript testing plugin. For Godot 4.5 you should use GUT 9.x (the latest is 9.5.0) [1] [2]. Install it by putting the `addons/gut` folder into your project and enabling the plugin (via **Project > Project Settings > Plugins**), then restart Godot [3]. After enabling, GUT's test panel appears in the editor and you can also run tests via the command-line. Below are guidelines on writing tests, simulating UI actions, handling your WebSocket backend, and running multiple instances for multiplayer testing.

Installation & Setup

- **Use GUT 9.x for Godot 4.x**. For Godot 4.5 use GUT 9.5.0 [1] [2].
- **Install GUT** from the Asset Library or GitHub. Unzip and place the `addons/gut` folder in your project, then enable "GUT" in **Project Settings > Plugins** and restart Godot [3].
- **Project structure:** Put your test scripts under a folder like `res://test/`. Ensure your tests run in the editor (note that GUT does *not* run in exported HTML5 builds, so all tests run on PC in the editor/CLI).

Writing Tests

- **Test scripts extend** `GutTest`. Every test file should start with `extends GutTest` [4].
- **Test methods** must be GDScript `func` with no parameters, and **must start with** `test_` [5]. For example: `func test_player_score():` ... . GUT will discover and run all such methods.
- **Assertions:** Inside each test use GUT's assertion methods (`assert_eq`, `assert_true`, `assert_false`, etc.) to check conditions. For example, `assert_eq(player.score, 100, "Score should be 100")`. If a test has no asserts or calls to `pass_test` / `fail_test`, it will be marked "risky" [5].
- **Setup/Teardown:** Optionally define setup/teardown hooks. You can implement `before_each()` / `after_each()` to run code before/after each test, and `before_all()` / `after_all()` around the whole suite [4]. Use these to create/reset game state or UI nodes.

Simulating User Input (End-to-End UI Tests)

- **GutInputSender / InputSender:** To simulate player actions, use GUT's input sender. In Godot 4, the `GutInputSender` (class name) or alias `InputSender` can send `InputEvent`s to nodes. Add your UI node (or the global `Input`) as a receiver.
- **Clicking UI:** Use methods like `mouse_left_click_at(position)` to simulate clicks [6]. For example, to click a button at its center:

```
var sender = GutInputSender.new(button_node)
sender.mouse_left_click_at(button_node.get_global_position())
await(sender.idle)  # wait for the click to process
assert_true(button_pressed)  # check effect
```

The `mouse_left_click_at` call generates a left-button down+up event at that position [6]. You can also call `mouse_left_button_down(up)` manually if you need more control. These events will be delivered to the node's `_gui_input` (and `_input`) handlers [7].

- **Chaining and Waiting:** You can chain events (e.g. press and hold) and insert waits. For example, `sender.key_down("jump").wait("2f").key_up("jump")` and then `await(sender.idle)` processes the sequence [6]. Always use `await` (or GUT's `wait_*` helpers) so the frame processes before asserting results [8].
- **Example:** To test a button press, add the scene (or node) to the tree in a test, then do:

```
var btn = MyButtonScene.instance()
add_child_autofree(btn)
var sender = GutInputSender.new(btn)
sender.mouse_left_click_at(btn.get_global_position())
await(sender.idle)
assert_true(btn.was_pressed)
```

Networking & WebSockets

- **Use your WebSocket server:** Since you already have a WebSocket backend, write tests that use Godot's `WebSocketClient` to connect. For example:

```
var ws = WebSocketClient.new()
ws.connect_to_url("ws://localhost:1234")
await wait_for_signal(ws.connection_succeeded, 5)
```

  This waits for the `connection_succeeded` signal (use `await wait_for_signal(...)` [8] with a timeout to avoid hanging tests).
- **Send/Receive Messages:** After connecting, you can use `ws.get_peer(1).put_packet()` or similar to send data, and listen to `data_received` signals. Use `await` or GUT's wait to pause until the response arrives.
- **Assertions:** Check that the game state or UI updates correctly when messages arrive. For example, after sending a "bet" message from one client, await the server's update broadcast and assert that each client's display changed as expected.

Multi-Instance Multiplayer Testing

- **Editor Support:** Godot 4's editor can run multiple instances of your project. In **Debug → Customize Run Instances**, enable **"Enable Multiple Instances"** [9]. Specify how many copies to run (e.g. 1 server + 3 clients).
- **Server vs Client:** If you have a "dedicated_server" mode, run one instance as the server (with that feature enabled) and the others as clients. All will connect via `localhost`. The forum notes "Enable Multiple Instances... need at least 2. One for the server and one for the client" [9].
- **Simulate each player:** In each client instance, you can manually (or via InputSender in separate GUT runs) simulate clicks on UI. For fully automated testing, you could script launching multiple Godot processes (e.g. using `OS.execute()` or separate CLI calls to the GUT command) and coordinate them, but that is complex. At minimum, use the editor's built-in multi-instance runner to visually test the interactions.
- **Verify Game State:** While tests may not easily run across instances, you can at least verify shared outcomes. For example, check that when Client A raises, Client B sees the updated bet.

Some of this logic can be tested by invoking network handlers directly in unit tests or by mocking peers, if end-to-end is too hard.

Running Tests

- **In-editor:** GUT adds a “Test” panel in the editor. Click “Run” to execute all tests, or select specific test scripts. Use the GUI to see passes/fails.
- **Command-line:** You can also run tests via CLI using `godot -s addons/gut/gut_cmdln.gd [options]` [10]. This is useful for automated or batch runs. (Even though CI isn’t required, this allows batch scripting of tests.)
- **Reporting:** GUT can export results in JUnit XML format if needed.

By following these guidelines, you can write end-to-end tests for your multiplayer poker game: simulate button clicks and actions with `GutInputSender`, await frame updates or network signals [6] [8], and (if needed) run multiple game instances together [9]. This gives you confidence that UI actions correctly drive gameplay logic and networking in your Godot project.

Sources: GUT documentation and community threads [1] [4] [6] [8] [9].

---

[1] [2] [3] GitHub - bitwes/Gut: Godot Unit Test. Unit testing tool for Godot Game Engine.
https://github.com/bitwes/Gut

[4] [5] Creating Tests — GUT 9.5.0 documentation
https://gut.readthedocs.io/en/latest/Creating-Tests.html

[6] [7] GutInputSender — GUT 9.5.0 documentation
https://gut.readthedocs.io/en/latest/class_ref/class_gutinputsender.html

[8] Awaiting — GUT 9.5.0 documentation
https://gut.readthedocs.io/en/latest/Awaiting.html

[9] Run in "dedicated server" mode from the editor - Help - Godot Forum
https://forum.godotengine.org/t/run-in-dedicated-server-mode-from-the-editor/62875

[10] Command Line — GUT 9.5.0 documentation
https://gut.readthedocs.io/en/latest/Command-Line.html