Guide contents

Choose a documented workflow

Understand what the plugin contains

Prepare the editor, dependencies and a test profile

Make one visible edit and test it

Package and diagnose

Record and asset release checklist

Troubleshooting by symptom

Publishing and provenance

Sources / References

Creation Kit editor and first plugin

Quest stages, objectives and aliases

Dialogue scene and player choices

Assets, BA2 archives and release testing

Papyrus activation, compiling and diagnostics

Records, load order and compatibility patches

Settlement site records and workshop links

Documentation: In progress Offline copy from ModDocs. Source links remain clickable.

Fallout 4 modding

The PC **Creation Kit** is Bethesda's editor for Fallout 4 content. It can create plugins that change existing records or add objects, locations and quests. Bethesda's modding announcement confirms the tool. This guide includes the Creation Kit, quest, dialogue, records, Papyrus, BA2 and settlement-site chapters in full below, so the main page works as a single documentation reference. Each chapter also keeps its own URL and downloads. Source review does not mean an in-game test was performed.

Choose a documented workflow

You want to	Read next	What that page covers
Learn the Creation Kit and make a first plugin	Editor and first plugin	Forms versus references, the Object/Cell/Render windows, save/reopen, and an installed test
Plan quest progression	Quest stages, objectives and aliases	Quest lifecycle, indexed Papyrus methods, alias failures, and a source-reviewed dialogue-to-stage design
Author a first NPC conversation	Dialogue scene and player choices	Speaker alias, greeting restriction, choice routing, exit paths and random replies
Understand an override or reconcile two plugins	Records, load order and compatibility patches	Open the real load order in FO4Edit, compare field values, decide whether a patch is needed, and test it
Add or diagnose Papyrus behavior	Papyrus activation, compiling and diagnostics	Script source and compiled output, event attachment, log evidence, and release checks
Package custom art or other external files	Assets, BA2 archives and release testing	Loose assets, BA2 choices, archive inspection, and clean-profile validation
Study an exterior settlement site	Settlement site records and workshop links	Location, zone, boundary, workbench links, initialization and a current-build validation plan

Each chapter is also linked at the top of this page and has PDF, self-contained HTML and Markdown downloads. The whole documentation ZIP contains the guide and chapters for offline use. Use the contents list to jump to a chapter in this document, or open its standalone page.

Understand what the plugin contains

A plugin records changes relative to the game's masters. An edit to an existing record can conflict with another plugin changing the same record; a new record needs its own ID and may need placement or another way to obtain it. Art files, scripts and other

external resources have their own paths and may not live inside the plugin. This distinction matters when a mod works on the author's machine but ships without a required file.

Goal	Likely starting point	Test before expanding
Adjust an existing item	Item record	Compare the original and changed behavior
Add a new item	New record, distribution or placement	Confirm it can be acquired and equipped
Build an interior	Cell, references, door links	Enter, save, reload and leave
Create a quest	Stages, aliases, dialogue, scripts	Run a complete minimal stage sequence

---

Prepare the editor, dependencies and a test profile

Install the Fallout 4 Creation Kit version that matches your game version. Bethesda's support page is a starting point for current setup information. The former Creation Kit wiki currently redirects to a maintenance page, so it cannot serve as the setup instructions. Because Fallout 4 and its tools have been updated, check the game and editor versions before copying an old tutorial's exact instructions.

Use a dedicated test save and minimal mod profile. Inventory the installed game build, Creation Kit build, DLC and plugin dependencies. A plugin is not self-contained when its records reference forms from another master: record those masters before publishing and do not casually remove them from the header. Load the relevant master data and save a new plugin under a unique name. Before making content, close and reopen the editor, load your plugin as active, and confirm it can be saved again. This establishes that you can reopen the actual file you intend to ship. Do not modify or redistribute the game's master files.

Keep three distinct locations: editable project/source, staged release, and installed test copy. If the test installation can see loose development assets, it may hide a missing-file problem in the staged release. Preserve an untouched source backup and a pre-test save before the first installed-game run.

Make one visible edit and test it

Find a record whose effect is easy to observe, such as a basic item. Decide whether this is an **override** of an existing form or a **new form**. An override deliberately changes a base-game record and can lose to a later plugin. A new form has its own identity but needs a valid path into gameplay, such as placement, distribution, crafting or a script. If you intend a new item, duplicate the record and assign a unique editor ID; if you intend a global change, edit the existing record deliberately. Change one property at a time. Save the plugin, enable it in the test profile, launch the game and inspect the item. Repeat after saving and reloading the game so you know the change persists.

After the editor save, open the plugin with the current Fallout 4 mode of xEdit. Confirm the plugin loads with the intended masters and inspect the changed record against its source. Record the form identifier, editor ID where present, changed fields and expected in-game effect. The records chapter explains the override columns and patch workflow. Do not treat a conflict color as proof of an error; compare field values and intended behavior.

For an interior, make the smallest traversable room first: a cell, floor, lighting and a door pair linked to a reachable area. Add collision and navigation tests before decorating it. For a quest, test the first stage, a failure or alternate path, and completion on a fresh save. A Creation Kit preview alone is not enough evidence that the mod works in the running game.

Package and diagnose

Ship the plugin plus only original assets it requires. Record its DLC/master dependencies and supported Fallout 4 version. Test from the packaged copy with the working files removed from the load path. If the plugin does not load, inspect activation and masters. If a new item exists in the editor but cannot be found in game, check distribution or placement. If models are missing, inspect asset paths and packaging. For a scripted feature, keep the authored `.psc` source with the editable project and ensure the required compiled `.pex` is present in the installed package; the Papyrus chapter covers the event and log checks. For custom art, compare the plugin's resource path with the staged loose-file tree or BA2 contents using the asset chapter.

Record and asset release checklist

Treat the plugin and its external files as separate parts of one release. A record can point at a model, texture, sound, or script that exists in your development installation but is missing from the distributed archive. Make a copy of the intended release in an otherwise clean test profile, disable the development copy, and use that copy for the final test. Keep a manifest of every file you created, the plugin that refers to it, and the source project you can edit later.

For an **override** of a base-game record, write down the original form, the field you changed, and the desired result. For a **new** record, write down its editor ID, how the player encounters it, and any required assets. This makes a later conflict review much more precise than testing only whether the plugin loads. Do not copy unrelated game assets into the release simply because the editor can see them.

Check the following in a clean test installation:

1. The plugin loads with exactly the master files listed in its header. Remove a dependency only after you have confirmed no surviving record refers to it.
2. Every custom asset resolves from the packaged release. Inspect the actual archive or loose-file tree, not just the editor's preview.
3. The changed record is visible through the intended gameplay path. A new inventory object with no placement, recipe, leveled-list entry, or scripted grant may be valid yet unreachable.
4. A save made with the mod active loads again and still exhibits the change. Also test a fresh save for initialization behavior. Do not recommend removing a scripted mod from a valuable save as a routine test.
5. The mod is tested with another plugin that edits the same record. Compare the resulting behavior with each plugin alone; load order may determine which override wins, while scripts and assets can produce additional interactions.

Bethesda's August 2026 Fallout 4 update notes changed both game archive handling and Creation Kit archiving behavior. Record the game and Creation Kit builds used for packaging. Rebuild and retest after either tool changes; an older successful test is not evidence for a newer release.

Troubleshooting by symptom

Symptom	Inspect first	Confirm the fix
Plugin cannot be enabled	Missing masters, wrong game or editor build, disabled plugin	Load the plugin in the editor and in a clean game profile
New object is absent	No placement or distribution path, hidden conditions, wrong cell	Make a temporary, reachable test placement and verify it in game
Changed object keeps old behavior	Another plugin overrides the same form, or a save retains state	Test the plugin alone on a fresh save, then compare the combined load order
Missing model or texture	Asset path in the record, archive contents, loose-file layout	Test the packaged release after removing development files from the load path
Script never runs	Attachment, compiled output, event conditions, initialization timing	Check the exact trigger on a fresh save and inspect the game's diagnostic output

These checks narrow the problem; they are not a substitute for a record-specific tutorial. Keep a copy of the failing package and test save before changing several things at once.

Publishing and provenance

For a PC release, include the plugin, required original assets, a concise dependency list, the game build tested, installation and removal notes, and the steps used to reproduce the main feature. Keep the editable project and an unmodified copy of each released package. If you publish through Bethesda.net, Bethesda's upload instructions describe the Creation Kit's **Upload Plugin and Archives** flow. That is a distribution route, not proof that the uploaded files work on every platform. Bethesda's current Creation Kit announcement asks creators to use the latest tool before uploading.

The previously linked Creation Kit wiki currently leads to a maintenance page. Keep that provenance link, but do not make a reader depend on it for the basic workflow above. Where this guide does not provide a checked field-by-field recipe, mark that gap instead of guessing a current editor value.

This guide remains **unfinished**. Its Creation Kit, quest, dialogue, Papyrus, xEdit conflict, BA2 and settlement-site chapters are source-reviewed, but have not been run or tested in Fallout 4 for this publication. The quest chapter now sketches one dialogue-to-stage/objective path, but no fragment has been compiled or bound in a recorded editor build. A version-tested complete quest, placeable workshop object, authored asset build, property-bound Papyrus feature, and named two-mod conflict example still need worked chapters and in-game validation.

Sources / References

External project and documentation links cited in this guide. Check the upstream release and version before following version-sensitive steps.

- modding announcement
- support page
- Creation Kit wiki
- xEdit
- August 2026 Fallout 4 update notes
- upload instructions
- current Creation Kit announcement
- maintenance page

Full documentation chapters

The chapters below are included in this guide and also remain available at their own URLs.

Creation Kit editor and first plugin

Open this chapter separately · Chapter PDF

Source status: The editor interface and initial load procedure were checked against the maintained Sim Settlements Creation Kit 101 tutorial and Bethesda's Creation Kit

availability notice on 2026-11-11. The local steps below have not been executed by ModDocs. Sim Settlements uses its own framework in later lessons; this chapter uses only the general Fallout 4 Creation Kit concepts from its introductory lesson.

What the editor changes

Fallout 4's game data consists of forms in master and plugin files. The Creation Kit displays those forms in the **Object Window**. A form represents a definition such as an item, actor, quest, door or static object. A placed **reference** is an instance of a form in a cell. Editing an existing form creates an override in your plugin; duplicating or creating one gives you a new form. Keeping that distinction in mind prevents two common mistakes: editing a base-game item when you intended a separate variant, and creating a new item but never placing or distributing it so that it can appear in play.

The plugin records the form edits and dependencies. A model, texture, compiled script or other external file may be a separate asset that the plugin references. The assets chapter covers staging those files. A plugin that loads in the editor can still fail in game when a required asset is absent from the release.

Prepare the game and editor

Record the installed Fallout 4 version, Creation Kit version, DLC, and active mod profile. Bethesda's August 2026 update notes include Creation Kit and archive changes, so do not use an old tool build as an unlabelled baseline. Preserve a copy of the current plugin and a disposable test save. Keep the editable project, release staging folder and installed test copy distinct.

In the Creation Kit choose **File > Data**, select `Fallout4.esm` and only the additional masters needed for the intended change, then load them. The Sim Settlements tutorial documents `Fallout4.esm` as the initial master and explains that loading may take time. Before editing, save a uniquely named plugin and reopen it as the active file. If you cannot reopen and save it, resolve the editor or file-path problem before authoring content. Never save changes into a game master.

Learn the working windows

The **Object Window** has a category tree and a filtered form list. Search for a known item or world object and inspect its Editor ID and form type. The **Cell View** selects a worldspace and a cell; interiors are separate cells, while exterior worldspaces contain connected areas. Double-clicking a cell loads it into the **Render Window**. The Render

Window is for placed references and visual layout. The Sim Settlements lesson documents camera orbit, pan and zoom, selection, and transform helpers. Work slowly with an isolated test cell: accidental movement of an existing reference is an actual plugin change, even if your intended task was only to look around.

An editor preview answers a narrow question: what the current editor has loaded. It does not prove that the plugin was enabled in the game, that the target cell is reachable, or that packaged resources resolve. For a change to an existing form, the Object Window and a record comparison in FO4Edit are usually better first checks than opening a random cell.

Create one controlled form change

Choose a form with a visible, repeatable result. Decide in advance whether you need a new form or an override of an existing one. If you need a new form, duplicate a suitable form in the relevant Object Window category, give it a distinctive Editor ID, and change one property. If you need to alter an existing item globally, edit that form deliberately and note its Form ID, the field changed, and the intended outcome. A unique Editor ID helps you locate your authored form; the Form ID identifies the record in plugin comparisons. Do not assume a renamed Editor ID creates a separate record if the editor is offering to rename the existing one.

Save the plugin, close and reopen it, and inspect your changed form. Open the plugin in FO4Edit, check the required masters and compare its record with earlier columns. The records and conflicts chapter explains how to distinguish an intentional override from a conflict needing a patch. At this point, record the exact plugin filename, Form ID, Editor ID, changed fields and whether the form is an override or new definition.

Make the result reachable and test it

A new form does not automatically appear in the player's inventory or world. Give the test an explicit route: for example, place a reference in a disposable test cell, add the item to a controlled distribution mechanism, or use a version-checked test command. Record the route so another person can reproduce it without guessing. For a placed reference, inspect the selected cell and reference after saving; for an item, verify the acquisition path and its actual in-game behavior.

Enable the staged plugin in a clean test profile with its masters. Check the intended result in game, then save, reload and repeat. Compare against a fresh test save when initialization may matter. A failed result should be narrowed in this order: plugin activation, missing masters, wrong record or cell, unreachable content, later override,

missing asset, then script/runtime behavior. Do not keep adding changes to a plugin whose first small test is still unexplained.

Release boundary and remaining work

Copy only the plugin and its authored dependencies into release staging. Test that staged copy with development files removed from the load path. Preserve source files, an unmodified prior release and the pre-test save. Publish the game/tool builds, master requirements, changed records, installation instructions, and removal limits. The BA2 release chapter covers asset packaging; Papyrus activation covers scripts. A complete field-by-field item, NPC or settlement object tutorial still needs its own source audit and recorded in-game run.

Quest stages, objectives and aliases

Open this chapter separately · Chapter PDF

Source status: This chapter maps the Fallout 4 `Quest` and `ReferenceAlias` interfaces in the maintained Papyrus Index and the preserved Creation Kit quest-tutorial hierarchy, checked 2026-11-11. The tutorial mirror currently rejects direct page fetches from this research environment, so this chapter does not invent its editor clicks or claim a completed quest build. The API entries below are source-reviewed; the workflow remains untested in game.

Separate the pieces of a quest

A quest record is more than a title and description. It owns a lifecycle, stages, objectives, aliases and optional dialogue or script fragments. A **stage** is a progression marker; an **objective** is a displayed task the player can see; an **alias** binds a quest role to an actor or reference. A stage can run script fragments, but setting a stage does not by itself guarantee that every objective is shown or completed in the way you intend. Plan the progression as a small state table before editing:

State	Trigger to inspect	Objective expected	Reference or actor needed	Exit condition
Not started	Start rule or explicit script call	None	Any startup aliases	Quest initialized

State	Trigger to inspect	Objective expected	Reference or actor needed	Exit condition
Initial task	Stage transition	First objective visible	Target alias resolves	Player reaches target
Task resolved	Interaction or dialogue	First objective completed	Same target remains valid	Completion stage runs

The table is a design aid, not a prescribed set of stage numbers. Use unique IDs and actual editor data from your project. Do not attach a long dialogue chain until a minimal start, state change and completion can be observed reliably.

Quest lifecycle and script methods

The indexed `Quest` script exposes `OnQuestInit()` after startup aliases have filled and while the startup stage runs. `OnQuestShutdown()` occurs when the quest shuts down and aliases are cleared. `OnStageSet(int auiStageID, int auiItemID)` reports a stage-setting event. These are different observation points: a handler requiring a filled alias belongs after successful startup, while a handler running after shutdown must not assume the alias still resolves. The Quest API index also lists Story Manager events, but which event starts your authored quest requires a separate editor design and version-specific test.

Interface	Parameters and return	Use and limit
`Start()`	`bool`; no arguments	Starts the quest; latent and waits for startup and its fragments. Check the returned result before assuming the quest began.
`SetStage(int aiStage)`	Stage index; `bool` result	Attempts a stage transition. A nonexistent or unsuccessful stage leaves state unchanged. It can wait for startup and stage fragments.
`SetObjectiveDisplayed(int aiObjective, bool abDisplayed=true, bool abForce=false)`	Objective index and display flags; no result	Shows or hides an objective. `abForce` requests display even if it was displayed earlier.

Interface	Parameters and return	Use and limit
`SetObjectiveCompleted(int aiObjective, bool abCompleted=true)`	Objective index and completion flag; no result	Marks an objective completed or incomplete. Check the visible journal state after the call.
`GetAlias(int aiAliasID)`	Alias ID; returns `Alias`	Retrieves an alias on that quest. The returned alias still needs to contain the expected runtime reference.

These signatures describe the indexed Fallout 4 API, not a generic Skyrim-only recipe. Keep the source game and script version visible when using a cross-game tutorial.

Alias and objective failure checks

When a quest starts but its interaction never fires, check whether the relevant alias filled with the intended actor or object. The ReferenceAlias index lists events such as `OnActivate(ObjectReference akActionRef)` and `OnDeath(Actor akKiller)` that can be received through an alias, but the event only helps if the runtime alias points to the correct reference and the script is attached in the right place. A missing actor or reference is a different problem from a stage that never changes.

When the journal objective is wrong, inspect stage state and objective state separately. Record which stage was requested, whether `SetStage` returned success, which objective index was displayed or completed, and whether a prior save had already advanced. If multiple stage fragments exist, the SetStage documentation warns that fragments can run concurrently rather than one after another; do not rely on an unstated ordering between them. Use logged evidence and a fresh test save to isolate initialization and repeat behavior.

Dialogue and release boundary

The first dialogue scene chapter covers a speaker alias, greeting, player choices, scene phases and exit paths. The following worked design connects **one** player response to a quest stage and objective. It is an original design derived from the indexed Fallout 4 `Quest` API, not a plugin we built or ran. A Creation Kit fragment must still be compiled, its properties bound, and its exact location checked in the installed editor before this can be called an executable example. The 2016 author tutorial on quest variables through

dialogue is a related route for storing a conversation decision; it does not establish that this particular design has been tested.

Work one response through the quest

Use a small quest with a single named NPC alias and the greeting/choice scene from the dialogue chapter. The stage and objective numbers below are **example IDs** for a new plugin. They are not IDs to paste into an existing quest. Assign the final IDs in the editor and keep the table with the project source.

Moment	Example stage	Example objective	Intended player-visible result
Quest starts	10	10 displayed	Player is asked to speak to the NPC
Player accepts the task in dialogue	20	10 completed; 20 displayed	Journal changes to the new task
Player declines	Stay at 10	10 remains displayed, or is intentionally hidden	Conversation ends without accidental acceptance
Task completion trigger	30	20 completed	Quest can finish after a separate completion check

1. **Create the quest's first state.** Give the quest a unique editor ID and its NPC reference alias. Decide how it starts in gameplay; a developer console command is only a diagnostic. Add stages 10, 20 and 30 and objective IDs 10 and 20. On stage 10, arrange to display objective 10. Keep the objective text specific to the target alias so that a wrong or unfilled alias is visible in testing. Confirm the alias can fill *before* adding dialogue conditions that depend on it.
2. **Attach the conversation to the same quest.** Reuse the dialogue chapter's greeting and player choice structure. Restrict the speaker to the intended quest alias, then make the accepting response point to the scene's exit or next phase. Give the declining response its own deliberate exit. A spoken line or a scene ending does not itself set a quest stage; add an explicit stage transition at the accepting response's authored fragment or other verified trigger.
3. **Request the stage change once.** In a fragment that has an actual, editor-bound `Quest` property, call `SetStage(20)` on that property and check its `bool` result. The

Fallout 4 SetStage API says a nonexistent or unsuccessful stage returns `false`. The call is latent and waits for startup and stage fragments. Do not infer success from the dialogue closing. Keep the declining path free of this call unless decline deliberately has its own stage.

4. **Move the journal state.** The stage-20 fragment can call `SetObjectiveCompleted(10)` and `SetObjectiveDisplayed(20)` on the same quest; these are separate objective API operations. An alternative is to place those calls on the accepting response after a successful stage change. Choose one owner for these changes so repeat dialogue cannot create two competing paths. Record that choice beside the quest diagram. Stage fragments on multiple items can start concurrently, so do not depend on one item's fragment finishing before another.
5. **Add a real completion trigger.** Stage 30 should be set by the actual task condition: an activation, item hand-in, area entry or other event appropriate to the mod. Verify the event's alias, reference and script attachment independently. Complete objective 20 only when that condition is met. If the quest should stop or be marked complete, author and test that separately; a stage number alone is not proof of a finished quest.

Do not copy a fragment name, autogenerated property name or VM binding from a tutorial into another plugin. Those are project-specific. The meaningful invariant is that the script invokes a `Quest` instance representing this quest, and that the editor has filled the property before a response can execute. The quest-variable tutorial demonstrates a different dialogue-linked state technique; use a variable when you need to remember a choice independent of a stage, and document its initialization and save behavior.

Verify each boundary on a disposable save

Check	Evidence to capture	If it fails
Start rule and alias	Quest starts; target alias resolves to the expected NPC	Inspect start conditions, alias fill rules and reference persistence before changing dialogue
Greeting and choice	Only the target NPC says the greeting; both choices appear	Recheck the forced alias and dialogue conditions
Decline	Scene exits; stage 20 is not done	Check for a shared fragment or stage call on the wrong response

Check	Evidence to capture	If it fails
Accept	`SetStage(20)` succeeds; objective 10 completes and 20 displays	Check the bound quest property, stage ID, fragment compile result and objective IDs
Repeat talk	Stage and objectives stay in the intended state	Add appropriate conditions or guards; do not assume the scene is one-shot
Completion	Task trigger sets stage 30; objective 20 completes	Inspect the actual trigger event, alias/reference and save state

The Fallout 4 Quest API distinguishes `GetStageDone(stage)` from `GetStage()`: the latter reports the highest completed stage and can mislead when stages are revisited or set out of order. For the acceptance check, inspect the specific stage and the journal objective separately. Retest with a fresh disposable save and a controlled save made before the conversation. Keep a copy of both saves and the exact plugin/build used. No ModDocs in-game test, compiled fragment, voice export, or current editor screenshot has been recorded for this design.

For a quest release, list plugin and master dependencies, scripts and voice or other assets, a reproducible player-facing start condition, stage/objective test path, supported game build, and known save limitations. A quest that progresses only through a developer test command is not yet a validated player-facing quest.

Dialogue scene and player choices

Open this chapter separately · Chapter PDF

Documentation status: In progress. **Sources checked:** 2026-11-11 against two Fallout 4 Creation Kit tutorials by Seddon4494 and the Fallout 4 Papyrus Scene/TopicInfo API index. **Technical validation:** the author demonstrates parts of this workflow in a 2016 game build; ModDocs has not made the example plugin or repeated the test on a current game and Creation Kit build. Editor labels and voice behavior may differ. Record your versions and inspect a current vanilla scene before relying on an older click path.

This chapter creates a small conversation with one NPC: a greeting, player choices, an optional second phase, and a deliberate exit. It follows Basic Dialogue and Longer Conversation and Random Responses, which are direct tutorials by the person

demonstrating the editor. The wording and example structure here are new. It does not require a dialogue-extension mod. The quest stages and aliases chapter covers the separate problem of quest progression.

Plan the conversation before opening the editor

Make a new plugin in an isolated test profile and preserve a pre-test save. Prepare one **unique NPC** already placed where the player can reach it. Use a distinctive editor ID, voice type, quest, alias, scene and topic names so you can inspect them again after saving. In the tutorial, a unique voice type helps separate the NPC from generic lines. Do not remove an NPC's factions as a general fix: factions can control other behavior. If you are editing an existing NPC, compare its current voice and faction data before changing either.

Use an explicit flow map. The following is a **design example**, not a captured plugin or tested Fallout 4 scene:

Step	Player sees or does	NPC response	Next state
Greeting	Activates the test NPC	Brief introduction	Choice phase
Choice A	Asks about the location	Gives one answer	Choice phase again
Choice B	Asks for a task	Gives a short request	Second phase
Choice C	Says goodbye	Acknowledges departure	End running scene
Second phase	Accepts or declines	Confirms the choice	End running scene

Write the *prompt*, the player's actual response text, and the NPC's response separately for each choice. In Fallout 4 the short choice prompt and the line spoken by the voiced protagonist are distinct editor fields in the tutorial. If you have not authored or licensed voice files, expect silence for new lines; an on-screen prompt is not evidence that spoken audio exists. Plan an exit from every reachable phase. A loop is useful for repeated questions only when another choice can leave it.

Bind the speaker through a quest alias

In the Creation Kit, load the required masters and set **your plugin** as active. The tutorial opens **Character > Voice Type**, creates a unique voice type and assigns it to the new NPC. Save the plugin, then create a new quest under **Character > Quest**. The dialogue quest is the owner of the alias and scene; it need not be presented as a journal quest simply because it holds dialogue. Reopen the quest after its first save if the editor has not exposed all tabs yet, as the video demonstrates.

On the quest's **Aliases** tab, create a **Reference Alias**, give it an identifiable name, and fill it from the unique actor. Verify the chosen actor rather than accepting the first similar name in the picker. The alias is the role the scene uses for its speaker; an empty or wrong alias can make a carefully authored scene appear to do nothing or use the wrong actor. Reopen the saved quest and inspect the alias fill before working on responses.

Create a new scene on the quest's **Scenes** tab and mark it as **Player Dialogue**. Add the NPC alias as the scene actor. Give the scene and each phase a distinctive name. The tutorial's editor sometimes did not show a newly named phase until the author closed and reopened its dialog. If a phase is absent from a later phase picker, save and reopen before creating duplicate phases.

Add a greeting and a choice phase

Create a **Start Greeting** topic for the actor and add a short NPC response. Do not assume that choosing the actor alias on the scene automatically confines every greeting response. In the basic tutorial, the first test let an unintended actor say the greeting; the author returned to the greeting's topic info and assigned its **Forced Alias** to the intended NPC alias. Reinspect the saved greeting topic info and test with the intended NPC *and* a nearby unrelated NPC. The second test is the one that catches a global greeting leak.

Add a phase with a **Player Dialogue** action. Enter each player's choice prompt and response text, then the NPC response in the corresponding response area. The basic lesson uses the positive, negative, neutral and question slots to demonstrate Fallout 4's choice layout. Those slots are a UI arrangement, not proof that every conversation must have four meaningful branches. Use only branches you can write and test; keep the prompt truthful about the longer response text.

For Choice A in the example map, direct its response back to the choice phase with a **Start Scene** action targeting the same scene and named phase. For Choice B, add a second phase and target that phase. In the follow-up tutorial, an action was used to advance to another phase and a selected response was routed back to a named phase. After editing each destination, close and reopen the scene, then confirm that the action

still names the intended scene and phase. A visual arrow or an action label alone is insufficient if it points to a stale name.

For Choice C and every terminal branch in the second phase, set **End Running Scene** on the final response. The first tutorial's in-game test showed the choices working but trapped the player in a loop because no exit had been authored. The author then added an end flag but did not demonstrate a second in-game run for that correction. Treat the exit as a required test, not as proven by the video's first run.

Optional: more than one reply to a choice

The follow-up lesson adds multiple response entries under one NPC answer and marks them as random, ending the random group at its final entry. If you use this, give **every** possible reply the same intended next state or an intentionally different state. The author initially forgot to return each random reply to the choice phase and corrected that before the game test. A branch that works with one random line but ends unexpectedly with another is an incomplete scene.

Random responses should be tested as a set. Re-enter the conversation until each reply has appeared, record the reply and destination, and confirm that the player can still exit. The tutorial's later game segment demonstrates more than one random response, a longer second phase and an ending. It does not establish current-build behavior for this site's own plugin.

Inspect the saved records and run the test

Save and close the Creation Kit, reopen the plugin as active, then inspect the quest, alias, scene, topic infos, greeting restriction, response actions and end flags. In FO4Edit, confirm the plugin has only expected masters and records; check for accidental NPC or quest overrides. Do not infer that a clean xEdit record tree proves the dialogue will play.

Run a fresh disposable save with only required mods active. Talk to the intended NPC, then to a different NPC nearby. For each choice, record the exact visible prompt, actual subtitle or audio behavior, NPC reply, next phase and whether the player can leave dialogue. Repeat on a reload. Check the loop path twice, both choices in the second phase, and every random reply you authored. A missing greeting can come from the wrong alias, conditions, quest startup or voice setup; a greeting spoken by another NPC points first to topic-info restrictions. A choice that never exits points to a missing end flag or a loop target. Change one condition or action at a time and keep the failing test save.

The Fallout 4 Scene API lists `OnBegin`, `OnEnd`, `OnPhaseBegin` and `OnPhaseEnd`; the TopicInfo API lists `OnBegin` and `OnEnd`. Those are possible diagnostics for a later scripted scene. This chapter does not add a Papyrus script or claim that those events fired in this example. A quest dialogue release still needs a tested start condition, voice-asset and subtitle review, stage/objective integration, localization plan, and save-state tests.

Sources and limits

- Fallout 4 Creation Kit Tutorial — Basic Dialogue, Seddon4494, published 2016-04-28. Its demonstrated game run covers the initial greeting and choices; the exit correction was described after that run.
- Fallout 4 Creation Kit Tutorial — Longer Conversation and Random Responses, Seddon4494, published 2016-04-29. Its game run covers a longer conversation, alternate replies and an ending. These videos were reviewed through auto-generated English captions; exact spoken words may have transcription errors.
- Fallout 4 Scene API and TopicInfo API, community-maintained Papyrus Index. Used only for the named event interfaces, not to claim a tested scene script.

Assets, BA2 archives and release testing

Open this chapter separately · Chapter PDF

Documentation status: In progress. **Sources checked:** 2026-11-11. **Technical validation:** the packaging workflow has not been run against a local Fallout 4 build for this publication. This chapter covers PC mod assets and distribution, not copyrighted game-asset redistribution.

A Fallout 4 plugin stores records, but custom meshes, textures, materials, sounds, and compiled scripts are separate files. A working development profile can conceal a broken package because loose files elsewhere in `Data` satisfy paths that the release never included. The release test must therefore start from an explicit file manifest and end in a clean profile.

Inventory the files before packing

Create a staging directory containing only files you made or have permission to redistribute. Preserve the relative paths that the plugin refers to. A small example

release might look like this:

```text
ModDocsExample.esp
Meshes/ModDocsExample/example.nif
Materials/ModDocsExample/example.bgsm
Textures/ModDocsExample/example_d.dds
Scripts/ModDocsActivationProbe.pex
```

This is an illustrative file tree, not a claim that these example assets exist. The compiled `.pex` is a runtime file; keep the editable `.psc` in your source archive, but include only files needed by the installed mod in the runtime package. Check every path referenced by a record or script property against the staging tree. Record the source of each file and its license. Do not repack Bethesda's game archives or include extracted base-game files in a release without rights to do so.

Choose the correct archive type

Fallout 4 uses BA2 archives for mod assets. The publicly readable STEP Archive2 guide documents separate **DDS** texture and **General** asset modes. Its installation section is outdated, so use a current Creation Kit source for installation. A texture archive is commonly named `<Plugin> - Textures.ba2`; a general archive is commonly named `<Plugin> - Main.ba2`. Keep the archive base name aligned with the plugin name and verify that the intended game build loads both archives. The names shown here are a convention to test, not a substitute for a load test.

For a texture archive, stage only the texture tree, choose Archive2's DDS format, import the staged files, save the archive, then reopen it and verify paths and file counts. For meshes, materials, scripts, and other non-texture files, create a separate General archive. A Nexus Archive2 guide warns that sounds and precombined meshes may require uncompressed General packing, but that URL returned HTTP 403 in our link audit. Treat this as an unverified compatibility warning: identify such files in your release and test their actual behavior rather than applying one compression setting to every asset.

Do not assume an extracted DDS is bit-for-bit identical to its source. The ByroRedux archive reader documentation explains that a texture BA2 stores mip chunks and requires its reader to reconstruct a DDS header. Keep your original editable texture sources and verify the archive in game. An older Nexus technical note, which returned HTTP 403 in our link audit, reports compatibility differences between archive/tool builds; use a matching Creation Kit and game build for the release you intend to support. Do not change archive version bytes by hand as a routine compatibility fix.

Separate development, package, and preservation copies

Keep three distinct artifacts:

Artifact	Purpose	Contents
Editable source	Rebuild later	Plugin project notes, `.psc`, original art and export settings
Installable release	Player use	Plugin, required BA2 archives or allowed loose files, dependency list
Verification record	Reproduce the release	Game/editor versions, archive-tool version, test profile, file manifest, checksums, results

After creating BA2 files, remove or disable the staged loose files from the game's load path. Install only the release package in a clean profile. Confirm that the plugin loads, the custom item or scene appears, its model and material resolve, and any scripted behavior still runs. Repeat on a fresh save and a disposable existing save where relevant. If the packaged copy fails but the development profile works, compare the archive's internal paths with the record paths before changing scripts or load order.

For preservation, store the source project and final release together with a readable build note. A version number in a filename is useful but insufficient: include the actual Fallout 4 executable build and Creation Kit build used to create and test the archive. Bethesda's August 2026 update notes changed the supported BA2 archive count and Creation Kit archiving behavior, so earlier packaging assumptions need renewed tests after an update.

Troubleshoot the packaged copy

Symptom	First comparison	Follow-up
Plugin works but model is invisible	Record model path versus archive internal path	Check the mesh and material paths, then retest with staging files disabled
Texture is black, missing, or wrong	Texture record/material path and DDS archive mode	Reopen the archive, inspect the source DDS, and test with the matching tool build

Symptom	First comparison	Follow-up
Script stops running after packaging	Plugin attachment and `.pex` location	Confirm the compiled file is present, then use the Papyrus diagnostics chapter
Game rejects or ignores the archive	Game build, Archive2 build, archive type, and name	Rebuild with a compatible tool and test without other mods
Release behaves differently from development	Extra loose files or other enabled mods	Compare a clean install against the manifest, one dependency at a time

Do not infer success from a successful Archive2 save alone. The archive must be reopened, installed, and exercised in the game. This chapter does not yet document binary BA2 headers, mesh export settings, material authoring, animation pipelines, or a named asset built and tested end to end; those remain explicit coverage gaps.

Sources

- Fallout 4 update notes for archive and Creation Kit changes
- STEP Archive2 guide, publicly readable format and packing instructions; its installation section is historical.
- ByroRedux BA2 reader documentation, publicly readable implementation notes on General and DX10 texture archives.
- Nexus Archive2 packing procedure and BA2 technical note, preserved for provenance; both returned HTTP 403 in the 2026-11-11 link audit.

Papyrus activation, compiling and diagnostics

Open this chapter separately · Chapter PDF

Documentation status: In progress. **Sources checked:** 2026-11-11. **Technical validation:** source-reviewed example; it has not been compiled or run in a local Fallout 4 installation for this publication. The procedure targets PC Creation Kit projects, not console development. Verify the game, editor, and script-source versions in your own installation before shipping.

Papyrus is Fallout 4's gameplay scripting language. A `.psc` file is editable source; the compiler produces a `.pex` file for the game to execute. The Papyrus compiler documentation describes this source/output distinction and warns that Skyrim and Fallout 4 compiled scripts are not interchangeable. A plugin can hold a script attachment and property values while the compiled script is a separate file to package.

This chapter makes one small, observable behavior: a placed object writes a diagnostic line when the player activates it. It deliberately avoids changing inventory, quests, or saves. Once this works, the same event can become the entry point for a real feature.

Set up a test project and script sources

Use a dedicated Fallout 4 test profile and a new plugin with `Fallout4.esm` as a master. Keep an untouched copy of the plugin and a disposable test save. Install the Creation Kit for the game build you are targeting. Its distributed script-source archives supply declarations such as `ObjectReference.psc`, `Game.psc`, and `Debug.psc`; the BGS Handbook's Fallout 4 setup notes describe the source ZIPs under `Data/Scripts/Source` and the `Institute_Papyrus_Flags.flg` compiler flags file. Confirm the actual paths in your installation instead of copying a path from an older release without checking it.

Keep your own source under a distinct project name and record its location. In the Creation Kit, compile through the script manager or the script attachment workflow. For a command-line build, the Fallout 4 compiler reference documents `-import`, `-output`, and `-flags`. These switches identify the source search paths, compiled output directory, and user-defined flags file. Do not reuse a Skyrim compiler or flags file. Inspect the compiler's version banner and errors before assuming a `.pex` was produced.

Write a minimal activation script

Create `ModDocsActivationProbe.psc` with this original example:

```papyrus
Scriptname ModDocsActivationProbe extends ObjectReference

Event OnActivate(ObjectReference akActionRef)
    If akActionRef == Game.GetPlayer()
        Debug.Trace("ModDocsActivationProbe: player activated the reference")
    EndIf
EndEvent
```

The Fallout 4 `ObjectReference.OnActivate` reference documents the event parameter as the activating reference. The Game.GetPlayer reference returns the player actor, so

the condition excludes other activators. Debug.Trace writes a diagnostic message. The example's script and trace text are ours; the API names and signatures are checked against the cited references.

Compile the source, then attach the compiled script to a placed test reference that the player can reach and activate. The object must support activation and the attachment must be saved in your plugin. The exact editor panes and buttons can vary by Creation Kit build; verify the attachment in the saved plugin rather than relying on the script file existing on disk. Test on a fresh save first, then on a disposable existing save. Activate the reference with the player and look for the unique `ModDocsActivationProbe` text in Papyrus diagnostics.

`OnActivate` has important timing limits. The reference notes that the event can arrive even when normal activation is blocked, and that a pickup may already have moved into inventory by the time the handler runs. A locked door may also send the event on an attempted activation. Do not use this handler alone to infer that an item was successfully collected or a door opened. For those features, check the state you actually need and consult the corresponding event documentation.

Compile, test, and diagnose

Symptom	Check	Next action
Compiler reports unknown base type or function	Fallout 4 source imports and compiler version	Point the compiler at the matching Fallout 4 script sources and retry
Compiler reports a flag error	`Institute_Papyrus_Flags.flg` path	Use the flags file shipped with the matching editor
Compilation succeeds but no event arrives	Script attachment, plugin activation, placed reference, and activation conditions	Reopen the saved plugin and test a reachable reference on a fresh save
Event arrives but no trace appears	Diagnostic logging configuration and compile mode	Test a debug build and inspect the appropriate Papyrus log
Works in development but not in the package	Missing `.pex`, unexpected path, or unlisted dependency	Install the packaged copy in a clean profile with working files disabled

The compiler reference distinguishes debug, release, and final processing. In particular, release processing can remove debug code. A missing trace in a release build is not by itself proof that the event did not run. Keep a separate debug build for diagnosis, and test the actual release behavior with an observable feature rather than a trace alone.

The BGS Handbook setup guide also describes loose-file loading. If a `.pex` is correct but unavailable to the game, inspect the profile's loose-file settings or the archive contents. Do not tell users to overwrite base-game script archives to test your plugin. Preserve the source `.psc`, compiled `.pex`, plugin, editor/game versions, and build notes together so the release can be reproduced later.

What this example does not cover

This is an event and packaging example, not a quest system, custom UI, or F4SE tutorial. It has no properties, persistence, or multiplayer behavior. A later Papyrus chapter must cover property binding, quest aliases, save migration, and a tested gameplay change before the Fallout 4 scripting area can be considered substantial. The official Creation Kit wiki was redirecting to a maintenance page at the time of review, so the cited community-maintained API index and compiler references are identified explicitly.

Sources

- Fallout 4 `OnActivate` event
- Fallout 4 `Game.GetPlayer`
- Fallout 4 `Debug.Trace`
- Fallout 4 compiler switches
- Fallout 4 Papyrus setup and source files

Records, load order and compatibility patches

Open this chapter separately · Chapter PDF

Documentation status: In progress. **Source checked:** 2026-11-11 against the xEdit project manual. **Technical validation:** the procedure below has not been run against a local Fallout 4 installation for this publication. The xEdit manual uses some Fallout 3 screenshots; its load-order and record-inspection concepts apply to xEdit's Fallout 4 mode, but do not treat the pictured forms as Fallout 4 examples.

This chapter is for a mod author whose plugin edits a Fallout 4 record also edited by another plugin. The goal is to identify the *specific fields* in conflict, decide which values should survive, and test a small compatibility patch. A colored conflict marker is evidence of an override relationship, not proof of a bug. The xEdit conflict manual explicitly distinguishes intentional overrides from harmful conflicts.

Prepare a reproducible load order

Make a separate test profile containing the base game, required DLC, your plugin, and one suspected conflicting plugin. Record the enabled plugin names and order before opening xEdit. Back up your plugin and test save. Start xEdit in its Fallout 4 mode (`FO4Edit` or the `-fo4` mode selector), select the plugins you need to inspect, and wait until loading finishes. The xEdit overview explains that Fallout 4's load order comes from files maintained by the mod manager; xEdit displays that order but does not change it. If you reorder plugins, close and reopen xEdit so the inspection reflects the new order.

The left tree groups records by plugin and type. Select a record to compare its versions in the right pane. The rightmost column is the last loaded override among the selected plugins. Read the individual fields, not just the record's color. The values that matter depend on the mod: an item may have a changed name, value, model path, keywords, or gameplay properties. A patch that copies one desired field but unintentionally restores another mod's old value can create a new incompatibility.

Isolate one conflict

Use **Apply Filter to show Conflicts** to narrow the view, then choose one record affected by both plugins. The xEdit overview lists this command in the navigation tree's context menu. Write a small comparison before editing:

Question	Record in your notes
What is the original form?	Plugin and form identifier shown by xEdit
Which plugins override it?	Every relevant column, in load order
Which fields differ?	Field names and each plugin's intended value
Which result should the player see?	The expected behavior in game
Can load order alone produce it?	Yes, no, or still unknown

For example, if one plugin changes an item's display name and another changes its value, neither plugin's whole-record override necessarily preserves both intents. Compare the fields in xEdit and test the outcome in game before deciding on a patch. If the later plugin already carries both intended changes, no extra patch may be needed. If a conflicting change is intentional and desirable, leave it alone.

Worked comparison pattern

The values below are a **schematic example**, not a captured Fallout 4 record or a tested mod. It shows the decision you should make from your own xEdit columns:

Column	Display name	Value	Interpretation
Original master	Standard name	10	Baseline for comparison
Earlier plugin A	Renamed item	10	A intends a name change
Later plugin B	Standard name	25	B intends a value change but carries the old name
Proposed patch	Renamed item	25	Combines the two stated intents, if no other fields matter

Do not copy only these two cells without inspecting the whole record. B may also change a keyword, model path, component list, or script property that its author intended to preserve. Conversely, A's name may deliberately depend on a different asset or condition. Write down each changed field and the reason for keeping it. The patch should be the last relevant override, with A and B as dependencies when their forms are referenced. Test the three unpatched cases before judging the patched result.

Choose the least invasive resolution

The xEdit manual presents four broad choices: accept the conflict, change load order, make a patch plugin, or stop using one incompatible mod. Use a load-order change only when one plugin's existing override already contains the full desired outcome. A load-order change cannot combine two distinct values by itself. Use a patch when the final record must carry selected fields from both plugins.

Create a **new** patch file for your own compatibility work rather than editing someone else's plugin. In xEdit, right-click the conflicted record and choose **Copy as override into...**, then choose a new file. The manual patch procedure describes this flow and the resulting master dependency. Give the patch a name that identifies the two mods it reconciles. Let xEdit add required masters, and verify the patch's master list before publishing. Load the patch after both source plugins so its override is the final one.

In the patch's record column, bring over only the fields needed for the intended combined behavior. Inspect the entire resulting record after each change. If the copied record references a form defined by another plugin, that plugin must remain a dependency. A patch that loads without its required master is not a valid release. Avoid changing form identifiers or master flags as a casual conflict fix; the xEdit manual warns that FormID renumbering can break saves and dependent plugins.

Automatic merged-patch output is a starting point, not a final answer. The xEdit patch discussion says the automated method cannot resolve every record type or infer the author's preferred result. Review every generated override that affects your mod before distributing it.

Test the patch and diagnose failures

Save the patch, restart xEdit with the intended load order, and confirm that your patch is the final column for the target record. Recheck the field values and read the Messages tab for load-time problems. The **Checking for Errors** command does not replace that review: the xEdit overview describes it as a check against record definitions, while missing-reference information appears during loading.

Then test in Fallout 4 with a fresh save and an existing test save. Compare three cases: your mod alone, the other mod alone, and both with the patch. Use a reachable item or location so the expected effect is observable. If the combined result fails but the record values look correct, investigate scripts, asset paths, initialization timing, and other records. A record-level patch cannot fix every interaction.

Package the patch with a short dependency list, the tested game build, the two source-plugin versions, and a description of the exact conflict it resolves. State that changing or removing either source mod requires renewed review. Keep the test profile and notes so a later game or mod update can be checked against the same scenario.

Scope and sources

This chapter documents the record-inspection and compatibility-patch workflow. It does not claim that every Fallout 4 conflict is a simple last-override case, and it does not provide a tested patch for a named pair of mods. The next validation step is a concrete two-mod example with captured xEdit record values and in-game results.

- xEdit overview and Fallout 4 load-order behavior
- xEdit conflict detection and manual patch procedure
- xEdit release and game-mode information

Settlement site records and workshop links

Open this chapter separately · Chapter PDF

Documentation status: In progress. **Source checked:** 2026-11-11. **Technical validation:** this chapter has not been run in the current Creation Kit or Fallout 4. The settlement construction source below explicitly describes a method written in 2015, although its archive page was edited later. Treat its exact property order and values as historical evidence. Verify each record against a working vanilla settlement and test on the game build you intend to support.

This chapter covers the **site that can become a settlement**, not a new item in the workshop build menu. Those are different projects. A placeable object needs its own object, recipe/menu integration and placement tests; a settlement site also needs a world location, build boundary, workshop workbench, linked markers and initialization. Start with the publicly readable settlement guide for the build boundary, workbench, resource container and markers. It stops before the initialization quests. The longer Nexus tutorial is listed under sources for provenance, but its page returned 403 in our link audit and may require a login. Its author warns that its method dates to 2015 and newer methods may exist. The Creation Kit editor chapter explains the Object, Cell View and Render windows used below.

Preserve a reference before editing

Use a separate editor profile and test save. Record the Fallout 4 and Creation Kit versions, installed DLC, active plugin and masters. Open a vanilla exterior settlement in the editor as a **read-only reference** and note its location record, encounter zone, workbench reference, build boundary, markers, linked references and workshop quests. Save your own work to a new plugin; do not save a copied vanilla settlement over the

reference. In xEdit, inspect the new plugin after every milestone so you can distinguish intentional new records from accidental overrides of world cells or workshop records.

The archived method is specifically for an **exterior** site. It assigns a new Location to a wilderness cell and connects that location to a workshop workbench and initialization quests. It is not a tested recipe for an interior settlement, a Nuka-World raider settlement, or a settlement added by another framework. Make a record map before placing objects:

Role	What to inspect in the Creation Kit	Evidence to record
Location and zone	Location parent, workshop-related location keywords, encounter-zone workshop flag and its Location field	Form IDs, Editor IDs, parent and keyword list
Cell	Cell's Location assignment and intended exterior worldspace	Cell ID, worldspace and changed fields
Workbench and boundary	Workshop workbench, build-area primitive/trigger and visible border	Reference IDs, position, scale and linked-reference keywords
Service markers	Center, spawn, attack and map markers; resource container	Which reference links to which target and under what keyword
Initialization	Workshop parent/add-location quest, alias collection and location-initialization quest	Quest IDs, alias fill conditions, script/property bindings and start/stop behavior

The archived tutorial lists `DefaultEmptyTrigger`, `WorkshopBorder`, `WorkshopResourceContainerSettlement`, a workshop workbench and several `XMarkerHeading` references. It also associates links such as `WorkshopLinkBuildableArea`, `WorkshopLinkCenter`, `WorkshopLinkContainer`, `WorkshopLinkSandbox` and `WorkshopLinkSpawn` with specific references. Use those names to find the *current* vanilla arrangement, then record the actual linked target and keyword for each edge. A marker that merely exists in the cell does not prove that the workbench or quest can find it.

Build a minimal, inspectable site

1. **Create a new Location and zone.** Give both unique Editor IDs. The historical source parents the Location to `CommonwealthLocation`, supplies workshop-related location keywords, and assigns the zone back to that Location. Compare those fields with a current vanilla exterior settlement before choosing them. In the target wilderness cell, assign the new Location. Record every existing cell field that changes; a new Location assignment is a real world-cell edit with compatibility implications.
2. **Place the boundary and workbench.** Add a build-area trigger or primitive, border, workbench and resource container in the target area. Keep the intended build volume visible during editing. Check the workbench's linked-reference list against the reference settlement and the map above. The archived source links the border to the workbench and links the workbench to center, container, sandbox and spawn references. Reopen each reference after saving to confirm the target and keyword persisted.
3. **Place and link service markers.** Put the center and spawn markers where their behavior can be observed in game. Keep the attack marker outside the intended build area as the historical tutorial advises, but test the actual attacker path and spawn behavior before release. Set a map marker and its location/name fields using a vanilla settlement as a comparison. Record the marker coordinates so a later update can be compared objectively.
4. **Inspect workshop initialization.** The archived method includes a quest that discovers workshops through a reference-collection alias and a separate location-initialization quest. Compare the current vanilla quest, alias conditions and script property names with the archive before creating equivalents. Do not copy a list of anonymous `Default/True` property positions from the old article: positions can change, and an apparently filled property list is not proof that the intended form was bound. In the editor, inspect each named property and form target; after saving, inspect the quest and alias records in xEdit.
5. **Save, reopen and inspect.** Close and reopen the editor with the plugin active. Verify the Location, zone, cell, workshop links, quests and aliases still point to your forms. In xEdit, check masters and the entire set of changed cells and records. Remove accidental edits from the new plugin in a backed-up working copy, then repeat the inspection.

Test the site as a system

Use a fresh test save or an isolated profile where the settlement has not already been initialized. Test discovery of the map location, activation of the workbench, the build boundary, placing and storing a simple object, resource counts, settlement ownership,

and settler spawning or recruitment. Record the actual in-game date and wait interval if testing delayed arrivals; the archive notes that settlers may not appear immediately. Re-enter the cell, save, reload and verify that placed objects and ownership persist. Then test with the DLC and other settlement mods declared as supported. A successful editor load is only an editor test; a working build menu is only one part of settlement initialization.

If the workbench activates but the build boundary is wrong, inspect the boundary reference, scale and `WorkshopLinkBuildableArea` edge. If resources or population fail to update, inspect the resource container, workshop object links, alias fill and initialization quest rather than moving every marker at once. If a map marker works but the site cannot build, separate the marker and Location fields from the workshop initialization path. For each fix, change one relationship, retest from a clean state, and keep the failing and passing plugin copies with their version notes.

This page does **not** claim a current, complete settlement recipe. A current-build reference capture, a worked plugin with recorded Form IDs, and in-game evidence for recruitment, persistence, navigation and DLC interactions remain to be added. The archived tutorial's optional overlap actor values and Nuka-World advice are deliberately not copied here because neither was independently validated for current builds.

Sources and limits

- Settlement tutorial (text) for Creation Kit, community author m0ds1984. **Access warning:** this URL returned HTTP 403 in the 2026-11-11 link audit; readers may encounter a login or access block. It labels itself an archive of a 2015 method; page edit date is 2025-01-05. Used for the role inventory and historic reference links, not for an unconditional current-build property recipe.
- Settlement Guide, a publicly readable, shorter guide covering boundary, workbench, resource container and marker links. It credits an earlier Nexus Forums guide. It does not document the full initialization path, and its steps have not been tested here.
- Sim Settlements Creation Kit 101, framework maintainer's editor orientation. Its building-plan tutorials describe **Sim Settlements add-ons**, which require that framework and do not replace native settlement-site setup.
- Bethesda's Fallout 4 update notes, used to justify recording the game and editor build for a reproducible release. It does not validate this settlement method.