

Save & Load Deluxe

Implementation Guide · v1.1.0

By Sierra · Original save menu by TDS

Save & Load Plus and related projects by tommy, kanpeen, CJYKK,   , FoG, KoffinKrypt, and akusien

This doc is for mod developers. It shows how to detect Save & Load Deluxe from your mod, put your own data on the save screen, react to saves and loads, and restyle the menu without breaking it. Everything here is written against v1.1.0, and every code sample was run against the actual plugin before it went in.

Looking for the settings?

They live in Mod Configs Redux, in the SAVE & LOAD DELUXE block. The mod works without Mod Configs – it falls back to defaults and keeps anything the save menu itself changes (like the grid resize gesture) in save/saveloadplus.json – but the settings menu is where the fun is. This doc is about building against the menu, not using it.

Quick start

You want to know whether the save menu your mod is talking to is this one. Check the core marker:

```
(function () {  
  
    var SLD = window._TDS_ && _TDS_.OmoriSaveLoad;  
  
    if (SLD && SLD.saveLoadReduxCore) {  
        // Save & Load Deluxe is the save menu  
    }  
  
})();
```

saveLoadReduxCore is a plain true that only this fork sets. SLD.version holds the version string, "1.1.0" at the time of writing. The two companion plugins check exactly this marker when they load, and say so on the console if it is missing.

The old detection flags cannot tell the menus apart

Imported.TDS_OmoriSaveLoad and the _TDS_.OmoriSaveLoad namespace are set by this mod and by the original Omori Save & Load, deliberately, so ecosystem mods keep working

across both. If your code actually cares which one it is talking to, `saveLoadReduxCore` is the only flag that answers.

How the menu is put together

One scene, `Scene_OmoriFile`, same as vanilla. Inside it:

- ◆ **Save cards** – one per slot: map snapshot, portrait, save name, level, playtime, location, and any badges other mods pinned on (below).
- ◆ **The detail window** – pages about the selected slot: SUMMARY, PARTY (a portrait carousel), DETAILS, and MODS, which lists every mod the loader had active when the save was made. Players flip pages with Q / W, the page up / page down keys, or the mouse wheel over the box. The window itself has three looks (the Details Text Style setting – see the player settings section); your registered rows render in all three.
- ◆ **The command list** – SAVE, LOAD, MOVE, COPY, RENAME, DELETE, plus one extra entry: CLOUD when a cloud provider is present, PONG when one is not. The two trade places automatically – the arcade cabinet moves in when the cloud moves out. BACK on the command list quits the game outright. The cabinet is the exit, legally speaking.

Two things worth taking from that. Your mod shows up on the MODS page of every save made with it enabled, read straight off the loader manifest (id, name, version) – nothing to do, just naming. And if you walk the command list, expect more entries than vanilla ever had.

Putting your data on the save screen

The detail window is extensible, and the pattern has two moments:

- ◆ **When the player saves**, your collect function runs. Whatever it returns is JSON-serialised and stored in the save's metadata, under your id.
- ◆ **When the menu shows a slot**, your rows and pages get built. You are handed the data that was collected, plus the slot's info object, and you hand back what to draw.

That split is the whole idea. The menu can show any save at any time, long after your mod's game state is gone, so anything you want on screen has to ride in the save metadata. Data lands in `info.omoriSaveLoadProviders` – keyed by your id, "row:" + id for standalone rows. It lives in the save metadata next to the slot, never inside the save contents: vanilla save files in, vanilla save files out.

Your data only exists on saves made after you registered

collect runs at save time, and only for saves made while your mod was loaded and active. A save from before, or one made with your mod disabled, has nothing under your id – your builder gets undefined for it. Show nothing, or show a placeholder. Do not assume the data is there.

The easy way: a stat list

```
var DetailAPI = _TDS_.OmoriSaveLoad.DetailAPI;

DetailAPI.registerStatList({
  id: 'my-mod:bestiary',
  title: 'BESTIARY',
  collect: function () {
    return { seen: $gameVariables.value(41) };
  },
  rows: function (saved) {
    return [
      DetailAPI.makeRow('ENEMIES SEEN', saved ? saved.seen : 0)
    ];
  }
});
```

That is a working page. It shows up after DETAILS, titled BESTIARY, with one row in it. saved is whatever your collect returned when the save was made – undefined on saves that predate you, which is why the sample guards with a ternary instead of pretending.

collect is JSON round-tripped before it is stored. Plain data survives; functions, live game objects and circular references do not. A collect that throws, or returns something JSON cannot express, is skipped with one console warning – the save itself goes through untouched. Anything over 32 KB logs a size warning too, and still saves; the number is a nudge, not a limit.

One row on an existing page

If all you need is a line on SUMMARY or DETAILS, register a row:

```
DetailAPI.registerRow({
  id: 'my-mod:deaths',
  page: 'DETAILS',
  label: 'DEATHS',
  collect: function () {
    return $gameVariables.value(42);
  }
});
```

No value given means the collected data is the value: this row reads the number out of the save metadata and draws it right-aligned, next to the label. Give it a value – string or function – if you want to format it instead:

```
DetailAPI.registerRow({
  id: 'my-mod:deaths',
  page: 'DETAILS',
  label: 'DEATHS',
  collect: function () { return $gameVariables.value(42); },
  value: function (saved) {
    return saved === undefined ? 'NONE YET' : String(saved);
  }
});
```

The page title is matched without caring about case, and a page that does not exist yet is created for you – so page: 'DETAILS' lands on the built-in page, and page: 'House' opens a HOUSE page of your own.

Field	What it does
id	Unique, yours. The key your data is stored under, prefixed row: in the metadata. Re-registering the same id replaces, quietly.
page	Which page the row sits on, by title. Created if missing.
label	The left-hand text. A string, or a function of (saved, context). Defaults to the id, uppercased.
value	The right-hand text. A string, or a function of (saved, context). Defaults to the collected data.
collect	Runs at save time; the return value rides in the metadata.
visible	Function of (saved, context). Return false to hide the row entirely.
richText	Draw the text through the escape-code renderer: \i[16] icons, \c[2] colours, the usual codes.
labelRichText / valueRichText	Same, per side. Escape codes are auto-detected anyway, so this is only for forcing plain text.
iconSize	Icon size for this row, 12 to 28. Default 18.
order / before / after	Where the row lands. See Ordering.

owner	Who owns the row, for scope disposal. Set for you if you use a scope.
-------	---

A row is drawn as a label on the left and a value on the right, value right-aligned, font shrinking down to a floor rather than clipping. Rows are cheap; the built-in pages carry a dozen.

A whole page

For anything bigger, register a provider and build pages yourself:

```
DetailAPI.registerProvider({
  id: 'my-mod:house',
  collect: function () {
    return { level: $gameVariables.value(7), name: $gameActors.actor(1).name() };
  },
  buildPages: function (saved, info, context) {
    if (saved === undefined) return null; // save predates us
    return DetailAPI.makePage('HOUSE', [
      DetailAPI.makeRow('OWNER', saved.name),
      DetailAPI.makeRow('LEVEL', saved.level)
    ]);
  }
});
```

buildPages gets (saved, info, context), and can return one page, an array of pages, or null to show nothing. info is the slot's own metadata – party, clams, steps, map, playtime, the works – so you can build pages out of data other mods and the base game already collected. makePage(title, rows, options) and makeRow(label, value, options) are helpers; anything you pass in options lands on the page or row (order, before, after, iconSize, richText).

Field	What it does
id	Unique, yours. The key your collected data is stored under. Re-registering replaces.
collect	Runs at save time. Same JSON rules as a row.
buildPages	Function of (saved, info, context). Returns a page, an array of them, or null.
order / before / after	Where your pages land among the others.
owner	Who owns the provider, for scope disposal.

A provider with no `buildPages` is still worth registering – its `collect` still runs, so your data rides in the save for anything else to read. There is a reader, in case you want it: `DetailAPI.getSavedData(info, id, fallback)` pulls your (or anyone's) data back out of an `info` object.

Ordering

Everything on the detail window – pages and rows both – sorts the same way:

- ◆ **order** is a number. Lower floats up. Entries with the same order keep registration order. Missing counts as 0, which is why the built-in pages sit where they are and yours land after.
- ◆ **before** / **after** name another entry – its id, its title, or a row's label – and move you relative to it. `before: 'DETAILS'` puts your page in front of the built-in one.

Ordering only reorders within the shared list; it never moves the PARTY carousel or the MODS page, which are their own thing.

Visibility filters

A visibility filter sees every row on every page, yours or not, and can veto:

```
DetailAPI.addVisibilityFilter(function (row, context) {  
  // hide any row whose label starts with SPOILER on the title screen  
  if (context.scene && !context.scene._canSave) {  
    return String(row[o]).indexOf('SPOILER') !== o;  
  }  
  return true;  
});
```

Return false to hide the row. Filters that throw are warned and skipped, never fatal. This is also the honest way to hide built-in rows you do not want – a filter can match `row[o] === 'STEPS'` as easily as your own.

Cleaning up after yourself (hehe)

If your mod can hot-reload, or you just like tidy state, take a scope:

```
var scope = DetailAPI.createScope('My Awesome Mod Suite');  
  
scope.registerStatList({ /* as above */ });  
scope.registerRow({ /* as above */ });  
scope.registerBadge({ /* see Badges */ });  
scope.on('afterSave', function (payload) { /* see Events */ });  
  
// later, all of it gone in one call:
```

```
scope.dispose();
```

A scope stamps your owner string on everything registered through it – providers, rows, badges, event listeners – and `dispose()` removes the lot. `DetailAPI.clearOwner('My Awesome Mod Suite')` does the same without keeping the scope object around.

Registering while the menu is open just works

Every register, unregister and dispose call refreshes the live scene if there is one – your page appears the moment you add it, no re-open needed. Registering before the menu opens works too; the pages are built when a slot is shown.

Badges on the save cards

Little tags in the corners of the save cards. Yours in one call:

```
_TDS_.OmoriSaveLoad.CardAPI.registerBadge({  
  id: 'my-mod:hardcore',  
  text: 'HARDCORE',  
  corner: 'top-right',  
  condition: function (context) {  
    return context.info && context.info.valid;  
  }  
});
```

Field	What it does
id	Unique, yours. Re-registering replaces.
text	The label.
iconIndex	An IconSet icon instead of, or next to, the text.
corner	top-left (default), top-right, bottom-left or bottom-right. Badges on the same corner stack outward.
color	Text colour, any CSS colour string.
condition	Function of the card context. Return false to skip the badge on that card.
owner	Who owns the badge, for scope disposal.

The card context carries slotId, info, valid, empty (true when the slot has no save in it), window and mode. Badges on the same corner line up side by side; a badge that will not fit the card width is skipped rather than squashed.

Image badges on the portrait box

A second badge style: small art stamped at the top left of the portrait box instead of a text tag in a card corner. A badge takes this route when you set anchor: 'portrait', or automatically when it carries only image content (no text and no iconIndex). Three kinds of art work, checked in this order:

```
// 1. procedural: you draw straight onto the card contents
_TDS_.OmoriSaveLoad.CardAPI.registerBadge({
  id: 'my-mod:stamp',
  anchor: 'portrait',
  draw: function (contents, x, y, size, context) {
    contents.fillRect(x, y, size, size, '#443355');
  }
});

// 2. a picture from img/pictures, scaled into the stamp
_TDS_.OmoriSaveLoad.CardAPI.registerBadge({
  id: 'my-mod:seal', anchor: 'portrait', imageName: 'my_seal'
});

// 3. an IconSet cell, same iconIndex you already know
_TDS_.OmoriSaveLoad.CardAPI.registerBadge({
  id: 'my-mod:mark', anchor: 'portrait', iconIndex: 87
});
```

Field	What it does
anchor	'portrait' puts the badge on the portrait box. Optional when the badge has only draw or imageName content.
draw	function (contents, x, y, size, context) – paint anything inside the stamp square. Wins over the other art fields.
imageName	A picture name from img/pictures, drawn scaled to the stamp. It redraws itself once the image loads.
size	Stamp edge in pixels, 8 to 32. Default 20.

Several portrait badges line up left to right; one that would leave the card is skipped. condition, owner and re-registration behave exactly like the corner badges.

Save and load events

EventAPI is the notification side. Seven events, all of them fire-and-forget:

```
var EventAPI = _TDS_.OmoriSaveLoad.EventAPI;

EventAPI.on('afterSave', function (payload) {
  console.log('saved to slot ' + payload.slotId + ', success: ' + payload.success);
});

EventAPI.on('slotDeleted', function (payload) {
  console.log('slot ' + payload.slotId + ' is gone');
});
```

Event	When it fires	Payload
beforeSave	The player confirmed a save, before it is written	slotId
afterSave	A save finished	slotId, success, info
afterLoad	A load finished	slotId, success
slotMoved	A save was moved to another slot	slotId, toSlotId, swapped (false)
slotCopied	A save was copied to another slot	slotId, toSlotId
slotDeleted	A save was deleted	slotId
slotRenamed	A save got its name changed	slotId, oldName, newName

on(event, listener, owner) tags the listener with an owner string if you give one – scopes do it for you. off(event, listener) removes one listener. Unknown event names throw immediately, which is a feature: you find the typo at load time, not three hours in. A listener that throws is warned on the console, once per fire, and never takes the save down with it.

afterSave fires whether the save worked or not – check success. The info in the payload is the slot's metadata, the same object the menu shows.

Vetoing a save or a load

Events cannot veto anything. If you need to block a save – a challenge mode that forbids saving, say – the hooks can:

```
_TDS_.OmoriSaveLoad.addSaveHook(function (slotId) {
  if ($gameSwitches.value(20)) {
    console.warn('saving is locked right now');
    return false;          // the save does not happen
  }
});

_TDS_.OmoriSaveLoad.addLoadHook(function (slotId) {
  return true;             // return false to block the load
});

_TDS_.OmoriSaveLoad.addPostLoadHook(function (slotId) {
  // runs after a successful load, game state restored
});
```

A hook returning false cancels that save or load outright; anything else – including a hook that throws, which is warned and skipped – lets it through. If you block a save this way, do tell the player why: from their side the menu just buzzed.

Changing how the menu looks

Three hooks for presentation, all on PresentationAPI. All of them are asked, never required: return nothing and the menu carries on exactly as before.

Card backgrounds

The little map picture on a card normally comes from the save's captured map tile. A background resolver can override it per slot:

```
_TDS_.OmoriSaveLoad.PresentationAPI.registerBackgroundResolver(
  function (location, info, context) {
    if (info && info.modData && info.modData.some(function (m) {
      return m.id === 'my-weather-mod';
    }))) {
      return [4, 2];        // column, row in loadscreen_backgrounds
    }
    return null;           // no opinion, carry on
  }
);
```

The return value is a [column, row] pair indexing the 100×100 cells of the system sheet `loadscreen_backgrounds` – the same sheet vanilla uses for its location art. Resolvers run newest first, and the first one to return a value wins. A resolver that throws is warned and skipped.

Precedence is simple: an explicit resolver opinion beats everything, including the automatic captured map image and the legacy location art. Slots your resolver declines (returns null for) keep whatever they had – captures where captures exist, the stock art where they do not. This holds in every art mode; you do not need the player to switch any config on for your resolver to run.

Portraits

Portrait resolvers are asked in two places: the PARTY page of the details box, and the portrait on every save card in the grid. The `context.type` tells you which – 'party-details' for the party page (with `context.window`), 'save-card' for a card (with `context.window`, `context.slotId` and `context.info`). A resolver can swap what either draws:

```
_TDS_.OmoriSaveLoad.PresentationAPI.registerPortraitResolver(
  function (member, context) {
    if (member.name === 'AUBREY' && $gameSwitches.value(30)) {
      return 'my_mod_aubrey_faces'; // a battle-style face sheet
    }
    return null; // no opinion
  }
);
```

Return a string to name a battle-style face sheet, or an object for full control: { `battleFaceName`, `menuFaceName`, `menuFaceIndex`, `menuActor`, `mode` }, where `mode` 'battle' or 'menu' pins which sprite kind is used. Same rules as backgrounds: newest wins, first non-null wins, throwers are warned and skipped.

Save cards draw battle-style portraits only, so on the 'save-card' context a plain string or the `battleFaceName` field is what counts – and the art has to actually exist, or the card quietly keeps the saved face. If your mod does not record a battle face for its leader, consider returning one here: the default fallback is the leader's vanilla battle sheet, which may be nothing like your character.

Where to register, and when

Registration placement trips people up, so here is the shape that works:

```
(function () {
  function register() {
    var MOD = _TDS_ && _TDS_.OmoriSaveLoad;
    if (!MOD || MOD.saveLoadReduxCore !== true) return false;
  }
})();
```

```

MOD.PresentationAPI.registerBackgroundResolver(myBackgroundResolver);
MOD.PresentationAPI.registerPortraitResolver(myPortraitResolver);
return true;
}
// top level of your plugin file is fine when load order puts Save &
// Load Deluxe first; otherwise defer one tick and check again
if (!register()) setTimeout(register, 0);
})();

```

Three things to know:

- ◆ **Register once, from code that runs once.** Plugin top level or a one-shot boot hook is right. Do not register from something that runs repeatedly – Scene_Map.onMapLoaded, an update loop, a menu open hook. The API deduplicates by function identity, so re-registering the same named function is harmless, but an inline anonymous function created fresh each call is a new identity every time, and you will stack duplicates that all run on every card.
- ◆ **Keep a reference if you ever want to remove.** removeBackgroundResolver and removePortraitResolver also match by function identity. A lambda you did not keep a reference to cannot be removed later – the register call returns your function back precisely so you can store it: `this._resolver = API.registerBackgroundResolver(function (...) {...});`
- ◆ **No scene requirement.** You do not have to wait for the save menu to exist. Registering while it happens to be open refreshes it live; registering before it opens just works when it does.

Command themes

The menu restyles itself as the player moves down the command list. You can get the same callback, per command:

```

_TDS_.OmoriSaveLoad.PresentationAPI.registerCommandTheme('cloud', function (scene, symbol)
{
    // called every frame while CLOUD is highlighted
    // dim the room, cue the thunder, whatever your mod does
});

```

Symbols are the command list's own: save, load, move, copy, rename, cloud, delete, pong. Your handler runs every frame the command is highlighted, so keep it cheap or gate it on a change. `unregisterCommandTheme(symbol, handler)` takes it back out.

Claiming your maps

If your mod adds maps, the menu's map snapshots already try to attribute them: it reads the mod loader's manifest for mapIds / maps / files entries and works out who owns a map. Registering a map provider makes it explicit and beats the heuristic:

```
_TDS_.OmoriSaveLoad.MapProviderAPI.register({
  id: 'my-mod',
  name: 'My Awesome Mod Suite',
  mapIds: [301, 302, 303]
});

// or a predicate, if your claim is not a list:
_TDS_.OmoriSaveLoad.MapProviderAPI.register({
  id: 'my-mod',
  name: 'My Awesome Mod Suite',
  ownsMap: function (mapId, location) {
    return mapId >= 301 && mapId <= 340;
  }
});
```

The provider is recorded with the map captures – in the capture log, and on the save's background descriptor – so a slot saved on your map can be told apart from one saved on a vanilla map that happens to share a display name. `unregister(id)` and `list()` are there too.

Bringing your own cloud

The CLOUD command is not the bundled cloud plugin. It is a contract, and the bundled plugin is one implementation of it. If you ship your own cloud sync, sign the contract and the button is yours:

```
window.OMORI_CLOUD_SAVE = true;
window.OMORI_CLOUD_SAVE_CALLBACK = function () {
  // the player picked CLOUD - open your interface
};
window.OMORI_CLOUD_SAVE_BUSY = function () {
  return mySyncInProgress; // true disables the CLOUD command
};
```

Global	What it is
OMORI_CLOUD_SAVE	Set to true to say a cloud provider exists.
OMORI_CLOUD_SAVE_CALLBACK	Called with no arguments when the player picks CLOUD. Open your UI here.

OMORI_CLOUD_SAVE_BUSY	Called to ask whether you are busy. True greys out the CLOUD command.
-----------------------	---

The command shows up in the list when the flag and the callback are both present – busy is optional, it only disables. Whoever sets the globals last owns the button, so do not sign the contract unless you mean it. `_TDS_.OmoriSaveLoad.describeCloudProvider()` returns a little report on the contract's current state – present, callback, busy, and a problems list – which is handy for debugging why a button did or did not appear.

The companion plugins

Save & Load Deluxe ships as one download: the core (`D_MAIN.js`) and two optional companions (`D_BGM.js`, `D_CLOUD.js`). Both companions are off by default, both are deletable – remove the files and the core does not care.

Custom BGM

Your own music in the save menu, plus a NOW PLAYING card. The track is a normal audio asset: drop my-track.ogg into audio/bgm and point the mod at it. The whole API is three setters:

Call	What it does
<code>setSaveLoadCustomBgmName(name)</code>	Which track to play. Extension stripped for you, empty falls back to the default.
<code>setSaveLoadNowPlayingText(text)</code>	The NOW PLAYING card text.
<code>setSaveLoadBgmLoop(from, to)</code>	Loop points in seconds. 0, 0 uses the file's own LOOPSTART / LOOPLength in metadata.

All three live on `_TDS_.OmoriSaveLoad`, take effect immediately, and the players' toggles still gate everything: the Custom BGM setting defaults to OFF, and nothing plays until the player turns it on. The plugin registers its three settings – Save/Load Music Volume, Custom BGM, NOW PLAYING – into the same SAVE & LOAD DELUXE block as the core.

Cloud Save

Free Puter account, background sync, local backups before every write, conflict prompts when two machines disagree. Strictly manual by default: nothing contacts anything until the player signs in, and it asks before even opening the browser. It signs the cloud contract above, which is why the CLOUD command appears when it is installed. Its settings – Cloud Auto-Sync (MANUAL / AUTO, manual being the default), Cloud Sound

Volume, and the Waiting Game, which lets you play PONG while a long sync runs – live in the SAVE & LOAD DELUXE block too.

Player settings that affect your stuff

These live in Mod Configs Redux, under SAVE & LOAD DELUXE. Worth knowing, because they change what your pages, badges and backgrounds look like at draw time:

Setting	Range
Details Text Style	LARGE, STRIP or CLASSIC
Party Stats	on or off
Party Cycle Time	1 to 10 seconds
Hide Details Box	on or off
Pixel-Perfect Detail Portraits	on or off
Slot Map Images	on or off
Full Map Backgrounds	on or off
Shared Location Backgrounds	on or off
Grid Left Extension	on or off
Widescreen Slot Stretch	on or off
Portrait Animation	on or off
Dynamic Portraits	on or off
Pixel-Perfect Portraits	on or off
Command Backgrounds	on or off
Mouse Interact	on or off
Mod List Scroll Speed	0 to 20

Details Text Style changes how your rows draw. Your registered rows and pages render in all three looks, but not identically. CLASSIC lists every row small on one page if possible. LARGE turns each row into a label-over-value card and splits long sections across subpages – long labels earn their room, but terse ones look better. STRIP marches rows left to right through a slim bar at the bottom, one section at a time, and only exists while a file is under the cursor. Do not assume a fixed rows-per-page count in any style; the menu paginates for you. If a section overflows, the subpage dots appear on their own.

Party Stats (default off) hides the built-in stat numbers – the averaged party rows on SUMMARY and DETAILS, and the per-member LEVEL, HEART, JUICE and EXP figures on the PARTY page. It does not touch anything you register: your rows and pages show regardless. If your mod's stat ranges make the vanilla averages look wrong, this toggle existing is your answer – you do not need to do anything.

Hide Details Box removes the box entirely while browsing. Your detail rows simply do not draw then; badges and card backgrounds still do.

The two capture toggles matter most for art: with Slot Map Images and Full Map Backgrounds both off, no map snapshots are taken at all, and your background resolvers are the only art cards will show. Widescreen Slot Stretch off pins cards to the vanilla 640×480 size under resolution mods – your badges and rows draw in less room.

Compatibility notes

- ◆ It detects other save/load mods and says so once, on the console: `_TDS_.OmoriSaveLoad.conflictingSaveMods()` returns the findings, and `saveMenuIntegrity()` reports any save-menu functions a later plugin hooked or replaced. Your mod is welcome to read either.
- ◆ Everything is layered aliases – the plugin wraps rather than replaces where it can, and it calls through to what it wrapped. Alias the same functions after it and you will stack fine; that is what the integrity check watches for.
- ◆ Save files stay vanilla format. The metadata this mod adds is additive, next to the slot, and the vanilla game ignores it. The per-save settings from Mod Configs Redux ride the save contents as documented in that mod's guide.
- ◆ If your mod hooks the save scene expecting vanilla's two-command menu, review those assumptions.

Things to be aware of

- ◆ The old flags answer the wrong question – `Imported.TDS_OmoriSaveLoad` is true for the original plugin and this one. Check `saveLoadReduxCore`.
- ◆ Your data is not on old saves – collect runs at save time only. Saves that predate your registration hand your builders undefined.
- ◆ A row with nothing to show still shows – if value, collect and saved data all come back empty, the row draws the string undefined. Give it a value, or a visible gate, or both.
- ◆ Listeners cannot veto – EventAPI is observe-only. Cancelling saves and loads is what `addSaveHook` and `addLoadHook` are for.
- ◆ Unknown event names throw – on purpose, so typos surface at load. The valid names are in the table above.

- ◆ collect is JSON-serialised – functions and live game objects do not survive the trip. Numbers, strings, booleans, arrays and plain objects do. Read what you need out of `$gameVariables` and friends and return that.
- ◆ Background resolvers return sheet coordinates – a [column, row] pair into `loadscreen_backgrounds`, not a file name and not a Bitmap.
- ◆ PARTY and MODS are not row pages – they render their own thing. Rows registered against their titles are stored but never drawn. Use SUMMARY, DETAILS, or your own page title.
- ◆ Command themes run every frame – while their command is highlighted. Cache your state; do not do work per call you can do once.
- ◆ Long values shrink, they do not wrap – the row font steps down to a floor and then clips. Keep values short, or use `richText` and mind the width yourself.

Getting help

If something breaks, or this guide is wrong somewhere, let me know on one of the modding Discord servers ([v_sierra](#)).

Do not run this alongside another full save/load menu – the original standalone, `saveloadplus`, anything equivalent. Only one save menu can own the plugin slot, and it might not be pretty if two try.