O M U M U

The Omumu Manual

Everything the platform can do

Edition 23 November 2026

The Omumu Manual
Edition 23 November 2026

Contents

Getting Started 2
Courses & Content 8
Selling 56
Pages & Sales Pages 74
Audience & Email 90
Automations 109
Teaching & Feedback 118
The Participant Folder 124
Podcasting 131
Analytics 144
The Agent Surface 148
The Command Line and Companion Apps 234
Product Skills 238

CHAPTER 1

Getting Started

An Omumu site is yours from the first minute: your own address, your own students, your own money. This chapter is the first hour — what the sign-up hands you, the handful of settings that make the site yours rather than a template, how money and email get connected, and where to look when something says it needs a different plan. It is the only chapter with almost no conversational path: setting up a business is paperwork, and most of these pages have no agent tools. The one exception, your brand, has them.

A path such as *Settings* → *Business Details* names a group in the admin's navigation — the rail on the left on a laptop, behind the Menu button on a phone — and the page under it.

Concepts

A site and its Headmaster. Everything on Omumu belongs to a site: courses, pages, offers, people. You, the person who created it, are its Headmaster — the role that can change anything on it. Other people on your site are students, or staff you invite to work with students. Which site you are on is always visible in the address; signing in to your site is separate from any account you may hold on omumu.com.

The admin and its home. Every admin page shares one navigation: Home and five groups — Products, Customers, Sales and marketing, Integrations and

Settings. Home is a status page. Its first card, *Set up your site*, is a checklist computed from what is actually true on the site, not from what you have clicked: edit your front page, take payments, connect email, add your first course, create your first offer, connect your AI. Items tick themselves when the thing is done, and the card can be dismissed.

What works before you connect anything. Sign-in links, receipts and the email a person gets when they are given access to a course always go out through Omumu. Invoice requests work without a payment provider. Everything else that leaves the site in your name — card and Vipps payments, email sequences — needs a connection you make once, and the pages for those connections say whether they are connected.

Your plan decides some doors. A site runs on a plan. The plan decides which features the admin offers and the limits on storage, courses and people; a page the plan does not include says so and points you to Omumu rather than erroring. How you reach Omumu support is part of the plan too — email support on most plans, priority email on the higher ones, phone support on the top one. This manual does not list plan names or prices: they are being consolidated, and *Settings → Platform Subscription* always shows yours.

Hands-on

Create your site

Mira starts at omumu.com's *Create your sales machine* page. Step one asks for her email and her name; step two asks for a card, with nothing charged today — the card is so the site keeps running after the fourteen-day trial, at the trial plan's monthly price in Norwegian kroner. She lands signed in on her new site's Home, with the setup checklist waiting. A welcome email tells her the site is ready and how to sign in again.

A second door exists for people who come through a partner: a voucher that creates the site on a different starting plan. Either way the site, the checklist and this chapter are the same.

There is no agent path to creating a site.

Tell the site who you are

Settings → *Business Details* holds the words the platform uses in your name: the site's name and description, the sender address on emails to your users, the support address shown to students, the address invoice requests go to, and the site's language (English, Norwegian, or whatever the visitor's browser asks for). The *Legal* tab takes your short and full company name and organisation number; the *Address* tab your street address. These appear on receipts and legal pages, so fill them before you sell. The page also shows your plan's storage limits and what you have used.

These settings have no agent tools; this is admin work.

Make it look like you

Mira says: *"Set my brand: primary colour #1F4E79, accent #F2A900, the Sora font, and rounder corners."*

`omumu_brand_update` merges those choices into the site's brand *draft*; `omumu_brand_get` returns the draft and the published brand, the derived light and dark colour roles, and the editor's address. The draft is not live: your pages keep the published brand until you publish on the Branding page. In the admin, *Settings* → *Business Details* → *Branding* shows a live preview — a course card and a lesson in your colours and font — with pickers for primary and accent colour, which of the two drives your buttons, the page colour behind everything, a catalogue of fonts in four families with an optional second font for headings, corner roundness, and uploads for a logo mark and a logo type, each with an optional version for dark backgrounds. *Save draft*, look at the preview, then publish. Until you publish, the site wears Omumu's defaults.

Get paid

Settings → *Payment Processors* shows Stripe and Vipps, each marked connected or not. Stripe connects through a single *Connect* button that opens Stripe's own onboarding; Vipps takes the four credentials from your Vipps merchant portal — client id, client secret, subscription key and merchant serial number. Invoice requests work with neither. The Selling chapter is about what you sell once a provider is connected.

Send email in your own name

Settings → *Email Integration* shows Resend (recommended) and SendGrid, each marked set up or not. Resend asks for your domain and a region and gives you DNS records to add at your domain host; once they verify, sequence emails leave from your domain. Until then, sequences hold their emails rather than sending them, and the pages that feed sequences say so. Sign-in links and receipts never wait on this.

Connect your AI

Integrations → *Set up your AI assistant* is the page that connects Claude, ChatGPT, Cursor, VS Code or a terminal to your site and shows a green "Connected" the moment one of them talks to it. Once connected, the assistant's first calls are `omumu_site_context` — which site it is on, your role there, the features the plan enables — and `omumu_profile_get`, which names the connection after you and the site. The Agent Surface chapter covers the connection, keys and consent in full; the setup checklist's last item is this page.

Bring in your team

Customers → *Users* → *Create user* adds a person by name and email with a role — student, staff or Headmaster — and optionally a course to give them at once. Omumu emails them a link to sign in; there is no password to hand over. Staff see the admin groups their role allows; a second Headmaster sees everything you do.

Your plan and billing

Settings → *Platform Subscription* shows your plan, renewal and next billing date, the card Omumu charges, your invoices, and *Compare Plans*. Cancelling returns the site to the free plan at the end of the paid period; you do not lose the site.

What to know about the edges

Stated plainly: the *Support* link in the admin header mails your own site's support address — that is the right address for your students, and the wrong

one for you; reach Omumu through the channel your plan gives you. *Account → Billing* in the header is you as a customer of your own site, so it is empty on a new site; your Omumu card and invoices are under *Settings → Platform Subscription*. And plan names, prices and the comparison page are in flux, so trust the subscription page over anything printed.

Reference

`omumu_brand_get`

Gets the site's Brand: the published choices and revision the site wears, the draft choices and whether they differ from published, the draft's derived light and dark color roles and any contrast adjustments, the font catalog identifiers, roundness steps and CTA colors to choose from, and the admin editor URL. Publishing is a human step: the creator publishes the draft in admin at editorUrl.

- **Write category:** read-only
- **Access:** requires the `BRAND_READ` permission

No parameters.

`omumu_brand_update`

Merges any of the given choices into the site's Brand draft; choices left out stay as they are, and an invalid color, unknown font or logo that is not this site's refuses the whole update. Returns the Brand as omumu_brand_get does. The draft is not live: the Site's pages and its public Brand JSON keep the published Brand until the creator reviews the draft and publishes it themselves in admin, at the returned editorUrl. There is no tool to publish.

- **Write category:** standard write
- **Access:** requires the `BRAND_WRITE` permission

Parameter	Type	Required	Description
`accentColor`	string	no	The accent color as #rrggbb. Becomes the tertiary role.
`ctaColor`	string	no	Which color drives calls to action: primary (default) or accent. Filled buttons (site, quiz, checkout, opt-in) follow this choice; headings, links, progress bars and meters stay on the primary regardless.
`font`	string	no	A font identifier from the catalog that omumu_brand_get lists under fonts, e.g. raleway.
`headingFont`	string	no	A font identifier from the catalog that omumu_brand_get lists under fonts, for headings only; body text keeps font. Give "text" to clear it back to the text font.
`markDarkResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo mark for dark surfaces (square, at least 1024 px). It is checked like an upload in the Brand editor.
`markResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo mark (square, at least 1024 px). It is checked like an upload in the Brand editor.
`primaryColor`	string	no	The primary color as #rrggbb, e.g. #215fa6. Kept exactly wherever it can carry readable text; otherwise nudged just enough to pass contrast, reported under adjustments.
`typeDarkResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo type for dark surfaces (square, at least 1024 px). It is checked like an upload in the Brand editor.

CHAPTER 2

Courses & Content

A course is the thing you deliver: what a buyer gets after the checkout in the Selling chapter, what an invite hands out, what a student opens on your site. This chapter is about building it — the tree of modules and lessons, the text and video inside them, the checklists that keep students moving — and about the few settings that decide how students move through it.

On Omumu you build a course by describing it. Say Mira has a webinar's worth of notes on automating invoice reminders: she tells her assistant what the course is and who it is for, and the assistant creates the course, its modules, its lessons and their content through the course tools, asking her before each step it is unsure about. Everything it makes is an ordinary course she can open and edit in the admin. The rest of the chapter shows the conversation and, for each step, the pages in the admin that do the same. A path such as *Products → Courses* names a group in the admin's navigation — the rail on the left on a laptop, behind the Menu button on a phone — and the page under it.

One rule to know before you start: **there is no draft step.** A course, a module, a lesson or a content block is live for students with access the moment it is saved. Build before you sell, or build on a copy.

Concepts

The course tree. A course holds modules in order; a module holds lessons in order; a lesson holds content blocks in order. Each level has a title, a description and an optional image. The description is what a student reads on the course's home page or the module list; the blocks are the lesson itself — the "meat". A content block is rich text (you can write it in Markdown) with, optionally, one attached file: a video, a PDF, an audio recording, an image.

Resources. Every uploaded file on your site is a resource in one site-wide library: course videos, lesson PDFs, podcast audio, generated images, transcripts. A block attaches a resource; a resource can be attached in several places, and it cannot be deleted while anything still uses it. Videos stream from the content delivery network, not from your site, so a lesson with video plays the same for one student and for a thousand.

Checklists and completion. A lesson can carry a checklist: short, self-attested tasks ("Log every repeated task for five working days"), each worth a few points of XP. Ticking them is how a student tells the course they did the work. Completion is derived from content, never configured: a lesson is complete when the student has opened it, ticked every checklist item, watched every video to the course's watch threshold, and handed in its assignment if it has one; a module when all its lessons are complete; a course when all its modules are. Adding content later adds requirements for students who are not yet done; it never un-completes anyone.

Gating. A course is either *open* — course access opens every module — or *sequential*, where completing one module unlocks the next. That is the whole mechanism: set the mode, and the unlocking is built in. The *watch threshold* says how much of a video counts as watched (a quarter, half, three quarters, 90 % or all of it; three quarters by default). There is no drip by date: Omumu unlocks on what a student has done, not on the calendar.

Access. A student gets into a course by holding access to it — bought through an offer (the Selling chapter), accepted through an invite, or granted by you — and signs in to open it. A course can instead be switched to *open access*, which lets anyone view it without signing in. You, as the site's Headmaster, can open every course you built.

Assignments. An assignment is a piece of work you expect back, attached to exactly one scope — a course, a module, a lesson, or the site itself — with the rating criteria the AI rater and your staff read when they judge a hand-in. Attaching one to a lesson puts the hand-in form on that lesson's page and

makes handing in part of completing it. An assignment attached to the site is an *All courses* assignment: it lives outside every course, gets its own page, counts toward no course, and is rated against whatever courses each student can open. Students find every assignment they have on their *Assignments* page, the All courses ones first and then one group per course. This chapter shows how to create one; the Teaching & Feedback chapter is about what happens when work comes in.

Quizzes. A quiz is a set of questions, one per screen, and it comes in two kinds. An *outcome quiz* sorts: each answer is worth points towards one or more *buckets* — the result categories — and whoever takes it lands in the bucket with the highest score. A *score quiz* measures: its questions belong to *scales*, each answer carries points, and the person gets a percent per scale and a total. The total picks the bucket (each bucket has a minimum percent), each scale shows the *level* text its percent reaches, and the scale with the most to gain gets its own text. In both kinds a bucket can act: tag the person, start an email sequence, grant a course, redirect to a page, show a message. A score quiz adds two things driven by the answers rather than the score: *notes*, paragraphs shown when a given answer was chosen, and *doors*, the one next step offered at the end, chosen by the first matching condition. A quiz can ask for a first name on its start screen and an email before it starts or after it ends, so it works on visitors as well as students, and a score quiz's result lives at a link the person can come back to. It is a tool for finding out who someone is as much as what they know; the Audience & Email chapter picks up the audience side.

Hands-on

Build a course by talking to your assistant

Mira says: *"Create a course called Invoice Reminders on Autopilot — for solo consultants who chase late payments by hand. Three modules: find the pattern, build the reminder, hand it off. Two lessons each. I'll give you the content."*

The assistant creates the course with its title and tagline, then each module and each lesson in order, and reports the course's address on your site. If you have the `create-course` skill (the Product Skills chapter), the assistant will interview you first and propose the structure before it builds anything; without

it, it builds what you asked for. Then Mira pastes her notes lesson by lesson — *"Lesson 1.1: here are my notes, make them the lesson"* — and the assistant writes each one as a content block.

Underneath: `omumu_course_create` makes the course (title, tagline) and returns its public address; `omumu_module_create` and `omumu_lesson_create` build the tree; `omumu_lesson_resource_create` writes a block, in Markdown, into a lesson. A lesson's description, a module's and the course's take Markdown too. `omumu_course_update` sets the description, the gating mode and the watch threshold; `omumu_course_get`, `omumu_module_list` and `omumu_lesson_resource_list` read the tree back.

In the admin: *Products → Courses → New Course* asks for a title, a description and an image and lands you on the course overview — the modules, assignments and students cards, the open-access and gating switches, the watch threshold. *Modules → New Module* lands on the module list; a module's page lists its lessons; a lesson's page has *Edit* (title, description, image), *Resources* (its content blocks: a title, the text, and an optional file) and *Checklist. View lesson* opens the lesson as a student sees it.

Two things to know about what comes back. Reads return the content in the editor's own JSON, not the Markdown you wrote, so an assistant asked to *edit* an existing block has to work from that. And a module or lesson your assistant creates cannot be deleted or reordered by it — those are admin-UI actions today. Build in the order you want to keep.

Give students something to tick

Mira says: *"Add a checklist to lesson 1.1: log every invoice you sent this month, mark the ones paid late, note how many reminders each took. Ten XP each."*

`omumu_checklist_batch_create` writes the whole list in one call (up to a hundred items); `omumu_checklist_item_create`, `_update`, `_reorder` and `_delete` adjust it. In the admin, a lesson's *Checklist* page adds items one line at a time and drags them into order.

Add a video

Mira records a screencast of the reminder spreadsheet and says: *"Here's the video for lesson 2.1."*

A video never travels through the conversation. The assistant asks for an upload slot with `omumu_resource_upload_video_presign`, the file goes straight to storage, and the assistant attaches the resulting resource to a content block (`omumu_lesson_resource_create` or `_update` with the resource's id). Any other file — a PDF, an audio recording, a document — goes the same way with `omumu_resource_upload_presign`. `omumu_resource_list` and `omumu_resource_get` find what is already uploaded, with each video's processing status; `omumu_resource_update` renames or describes a file.

In the admin, the site's Resources page (it has no entry in the navigation yet; open `/admin/resources`) takes drag-and-drop uploads and shows, per file, where it is used; a content block's form uploads a file directly into the block. A video counts as watched at the course's threshold, which feeds completion.

On a plan with transcription, attaching a video or audio file to a lesson queues a transcript automatically; the transcript can be saved as a text resource of its own, and it is part of what the AI rater reads when it judges hand-ins against that lesson.

Give it a picture

Mira says: *"Make a cover image for the course: a tidy desk, one envelope, morning light."*

`omumu_generate_image` makes an image from a prompt and, if you name a course, module or lesson, attaches it at once (landscape by default — the shape course cards use); `omumu_set_image` attaches an image resource that already exists. In the admin, the image field on the course, module and lesson forms takes a dropped file or generates one from a prompt in place. Image generation is a plan feature, and a higher plan uses a stronger model.

Unlock modules in order

Mira says: *"Make the course sequential, and count a video as watched at 90 %."*

`omumu_course_update` with the gating mode and the watch threshold; in the admin, the switches on the course overview. Under sequential gating a student sees every module in the navigation but can open only the ones they have reached, and completing a module — all its lessons, each with its checklist, videos and assignment — opens the next.

Copy what works

Mira's second course reuses the time-audit module from her first. *"Copy Week 1 from Automate Your Week into the new course."*

`omumu_module_copy` moves a copy of a module, lessons and blocks included, into another course; `omumu_course_duplicate` copies a whole course. In the admin, every course, module and lesson has *Duplicate*, and modules and lessons have *Copy into…* for another course or module. A duplicated course keeps the same title, gating mode and watch threshold, so rename it.

Ask for work back

Mira says: *"Add an assignment to lesson 3.2: hand in a screenshot of your first automatic reminder. Rate it on whether the reminder names the invoice, the amount and the date."*

`omumu_assignment_create` attaches an assignment to exactly one of a course, module or lesson, with its rating criteria, as a draft; `omumu_assignment_update` publishes it and can move it to another scope in the same course; `omumu_assignment_list` shows them. In the admin, *Products → Assignments* lists every assignment on the site, *All courses* first and then one group per course, and the course overview's *Assignments* card opens it filtered to that course. *New Assignment* asks where the assignment belongs first: *All courses*, or a course with an optional module or lesson.

For one that belongs to all courses, leave the course out: *"Add an assignment for all courses, Evening reflection: three lines on the day — what went well, what you learned, what you take with you. Ask for a check-in light."* `omumu_assignment_create` with no course, module or lesson creates it; `omumu_assignment_list` without a course lists the All courses ones in order, each with its position, and `omumu_assignment_reorder` moves one to another: *"Put Morning reflection first."* Only All courses assignments have an order to change. One cannot be moved into a course, nor a course's assignment out to All courses — that is a different assignment. In the admin, the *All courses* group shows them in the order your students see them: a new one goes last, and dragging a row by its handle onto another puts it in that row's place and shifts the ones between. The course groups keep course order and offer no dragging. What a student sees, how the AI rates a hand-in and how you reply is the Teaching & Feedback chapter.

Build a scored quiz

Mira wants to know, before someone buys, how much of their invoicing is already automated. *"Make a three-question quiz that sorts people into 'By hand', 'Half way' and 'Automated', and tag them with the result."*

Six calls build it: `omumu_quiz_create` (a label), `omumu_question_create` per question (single- or multiple-choice), `omumu_answer_create` per answer, `omumu_bucket_create` per result with what it should do (a tag, a sequence, a course, a redirect, a message), and `omumu_option_point_batch` to give each answer its weight towards each bucket. `omumu_quiz_get` returns the whole scoring matrix; `omumu_quiz_update` links an opt-in form before or after the quiz and, on a site connected to HighLevel, sets whether the quiz delivers its respondents there and which HighLevel contact field receives each respondent's result link. In the admin, *Sales and marketing → Quizzes* has the same editor and a results page per quiz, and the quiz's page shows the public address to share. What a quiz does for your audience — the buckets' sequences, tags and courses — is the Audience & Email chapter.

Build a score quiz from a document

A self-test with a result page is bigger than six calls, so it is built from one definition. Say a leadership coach has a test in his book: twelve statements rated *Sjelden* to *Nesten alltid*, four per scale, three profiles by total, a next step chosen by who the person is, and a few background questions. He hands the document to his assistant: *"Build this as a score quiz on my site. Three scales with low, middle and high texts at 0, 50 and 83 percent, the exercise for the scale with the most to gain, the three profiles at 80, 50 and 0 percent, the call door for leaders who want company soon, the webinar door for those with the book, the book offer for everyone else."*

One call, `omumu_quiz_build`, takes the whole definition — quiz, scales with levels, questions with answers and points (and an *answer tag* on any answer that should mark the person, a book owner say), buckets with minimum percents, doors, notes — checks every reference and creates everything or nothing. The assistant uses its own keys for scales, questions and answers inside the definition and gets the real ids back, so a note that quotes a free-text answer (`[answer:<questionId>]`) is added right after with `omumu_note_create`. Result copy may use placeholders: `[firstName]`, `[highestScale]`, `[lowestScale]`,

`[highestQuestion]` and `[scale:<scaleId>]`. Then the follow-up: a post-quiz opt-in form linked with `postQuizOptinFormId`, which saves the email as soon as it is valid and asks for whatever else the form declares, a mobile number say, a *completion tag* on the quiz, a *capture tag* for everyone who leaves the email, finished or not, an email sequence per profile whose first email carries `$quiz_result_link`, and, as an ordinary automation, *form submitted → wait an hour → has the completion tag? no → start a reminder sequence*. The person who takes it gets a first-name screen, one question per screen with a progress bar, the email form, and the result page: greeting, profile, a meter and text per scale, the exercise, the notes that apply and the one door. A door can also carry a checkout in place of its button: give it the embed link of an offer from an allowed payment provider (EdPay for now), `checkoutEmbedUrl` on `omumu_door_create`, `omumu_door_update` or a door in `omumu_quiz_build`, and the person can buy right there on the result page. In the admin, the quiz's page shows the whole score setup read-only, and the person's page under *Customers → Users* shows each result with its profile, total, scales and the result link.

Look at it as a student

View lesson on any lesson page, or the course's own address, opens the student side: the course home with its modules and any hand-in forms, then the lessons with their blocks, checklist and navigation. You can open any course you built in the admin. A course your assistant built is a different matter: the agent's create does not give you student access, so to walk it as a student, grant yourself access first (*Customers → Users*, your own user, *Manage Course Access*).

What you cannot do yet

Stated plainly: modules and lessons have no draft state, so there is no way to stage a week before students see it; reads over the connection come back in the editor's JSON rather than Markdown; your assistant can create modules and lessons but not delete or reorder them; and unlocking is by completion only, never by date. A new site may also list one or two of Omumu's own onboarding courses on its student side — they are read-only copies for you, not for your students. A door embeds checkouts only from the payment providers Omumu allows, EdPay for now; adding another is a change to Omumu, not a setting.

Reference

`omumu_answer_create`

Creates an answer option for a question. On an outcome quiz, link this answer to one or more buckets using omumu_option_point_create after creating the answer. On a score quiz, set points so choosing this answer adds (or, for a reverse-worded statement, subtracts) that many points from its question's scale. tag, when set, is applied at completion to an identified respondent who chose this answer, alongside the bucket, door and completion tags — the way a single background answer ("do you have the book?") becomes a fact on the contact, usable by HAS_TAG.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`label`	string	yes	The answer label shown to users
`points`	number	no	Points this answer carries toward its question's scale (score quizzes only; default 0)
`questionId`	string	yes	The question ID
`tag`	string	no	Tag applied at completion to an identified respondent who chose this answer
`value`	string	no	The answer value (defaults to label)

`omumu_answer_update`

Updates an answer option's label, value, points (score quizzes only) and/or tag (applied at completion to an identified respondent who chose this answer)

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`answerId`	string	yes	The answer ID
`label`	string	no	New answer label
`points`	number	no	New points value (score quizzes only)
`tag`	string	no	Tag applied at completion to an identified respondent who chose this answer, or null to remove
`value`	string	no	New answer value

`omumu_assignment_create`

Creates an assignment attached to exactly one scope: a course, a module, a lesson, or the site itself. Provide at most one of courseId, moduleId, or lessonId — more than one is rejected. Omitting all three creates a SITE assignment, shown to creators as All courses: attached to the whole site and handed in on its own page /assignments/{aid}, where students write about everything they have rather than one course. Students find it first on their Assignments page (/assignments); creators find it in the All courses group of /admin/assignments. A new SITE assignment goes last in the All courses order; move it with omumu_assignment_reorder. It counts toward no course's completion and is rated against each student's own accessible courses. COURSE scope: the assignment appears on the course homepage and is visible across all its modules and lessons. MODULE scope: the assignment is shown within a specific module and reflects that module's work. LESSON scope: the assignment is tied to a single lesson and reflects its content. New assignments default to DRAFT status; pass status=PUBLISHED to make the assignment immediately visible to students. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`asksForLight`	boolean	no	Check-in: every hand-in must include the student's own Light — green, yellow or red. Defaults to false.
`courseId`	string	no	Attach the assignment to this course (COURSE scope). Must not be combined with moduleId or lessonId.
`description`	string	no	The assignment instructions or description (Markdown)
`lessonId`	string	no	Attach the assignment to this lesson (LESSON scope). The owning course is derived automatically. Must not be combined with courseId or moduleId.
`moduleId`	string	no	Attach the assignment to this module (MODULE scope). The owning course is derived automatically. Must not be combined with courseId or lessonId.
`notifyStaffOnHandIn`	boolean	no	Email every Headmaster and Staff member on the site when a student hands in. Defaults to true.
`ratingPrompt`	string	no	Prompt shown to instructors when rating a submission (optional)
`status`	string	no	Initial status: DRAFT (default, not visible to students) or PUBLISHED (visible)
`title`	string	yes	The assignment title

`omumu_assignment_list`

Returns all assignments (DRAFT and PUBLISHED) for a course, with each assignment's scope kind (COURSE, MODULE, LESSON, or SITE) and the scope's id stated explicitly. Without a courseId it returns the site's SITE assignments only — the All courses assignments, attached to the whole site rather than to any course, each handed in on its own page /assignments/{aid} — in the All courses order, with each one's 0-based sortOrder; order them with omumu_assignment_reorder. Course, module and lesson assignments report no sortOrder. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	no	The course whose assignments to list; omit to list the site's SITE assignments (All courses)

`omumu_assignment_reorder`

Moves a SITE assignment (an All courses assignment) to a 0-based position in the site's All courses order, the order students see first on /assignments and the admin sees in the All courses group of /admin/assignments. The other SITE assignments shift to make room; a position past the end places the assignment last. Draft ones hold their place in the order too. Only SITE assignments have an order: a course, module or lesson assignment is refused, as is an assignment of another site. Read the current order with omumu_assignment_list without a courseId. Returns the moved assignment with its new sortOrder. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`assignmentId`	string	yes	The SITE assignment to move
`newSortOrder`	integer	yes	The position to move it to, 0-based (0 is first)

`omumu_assignment_update`

Updates an existing assignment's title, description, ratingPrompt, status, and/or scope. Provide at most one of courseId, moduleId, or lessonId to set the new scope; omitting all three means the SITE scope, so omit them exactly when updating a SITE assignment. The new scope must belong to the same course as the assignment — a scope change across the course boundary (to or from the SITE scope) is rejected: that is a different assignment. A SITE assignment is an All courses assignment, handed in on its own page /assignments/{aid}; its position in the All courses order is changed with omumu_assignment_reorder, not here. Fields not provided are left unchanged. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`asksForLight`	boolean	no	Check-in: every hand-in must include the student's own Light — green, yellow or red. Omit to leave unchanged.
`assignmentId`	string	yes	The assignment ID to update
`courseId`	string	no	New COURSE scope. Must not be combined with moduleId or lessonId.
`description`	string	no	New assignment instructions (Markdown)
`lessonId`	string	no	New LESSON scope. Must not be combined with courseId or moduleId.
`moduleId`	string	no	New MODULE scope. Must not be combined with courseId or lessonId.
`notifyStaffOnHandIn`	boolean	no	Email every Headmaster and Staff member on the site when a student hands in. Omit to leave unchanged.
`ratingPrompt`	string	no	New rating prompt for instructors
`status`	string	no	New status: DRAFT (hidden from students) or PUBLISHED (visible)
`title`	string	no	New assignment title

`omumu_bucket_create`

Creates a result bucket for the quiz. On an outcome quiz it's a category users are sorted into by plurality — after creating buckets, link answers to them using omumu_option_point_create. On a score quiz it's a profile: set minPercent (0-100) and the bucket with the highest minPercent at or below the respondent's total percent wins; a bucket with no minPercent never wins. Each bucket can trigger actions: send email sequence, grant course access, redirect

to URL. description may be Quill delta JSON when descriptionEncoding is QUILL_JSON. On a score quiz, description supports result-copy placeholders (see omumu_note_create for the full set). videoResourceId attaches one video, shown after the description on the result page — the resource must already exist on this site (via the presigned video upload) and be a video.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Bucket description
`descriptionEncoding`	string	no	How description is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`minPercent`	number	no	Score quizzes only: lower percent bound (0-100) of the quiz's total this bucket wins at or above
`name`	string	yes	The bucket name (e.g., 'Beginner', 'Advanced')
`quizId`	string	yes	The quiz ID
`videoResourceId`	string	no	Id of an existing video site resource on this site, shown after the description on the result page

`omumu_bucket_update`

Updates a bucket's properties including actions (email sequence, course, redirect, tag) and, for a score quiz, its percent-range profile fields (minPercent, descriptionEncoding). Pass null for an action to remove it, or leave the key out to keep it unchanged. emailSequenceId and courseId must be a sequence or course on this site. videoResourceId attaches one video (must belong to this site and be a video); pass null to remove it.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`bucketId`	string	yes	The bucket ID
`courseId`	string	no	Course on this site to grant access to; null removes the course
`description`	string	no	New description
`descriptionEncoding`	string	no	How description is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`emailSequenceId`	string	no	Email sequence on this site to subscribe user to; null removes the sequence
`minPercent`	number	no	Score quizzes only: lower percent bound (0-100) of the quiz's total this bucket wins at or above
`name`	string	no	New bucket name
`redirectUrl`	string	no	URL to redirect to after quiz; null removes the redirect
`successMessage`	string	no	Message shown after quiz completion
`tagName`	string	no	Tag to apply to user; null removes the tag
`videoResourceId`	string	no	Id of an existing video site resource on this site; null removes the current video

`omumu_checklist_batch_create`

Creates multiple checklist items for a lesson in one call (max 100). Atomic — all items are created or none. Useful for AI workflows that generate complete checklists at once.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`items`	array	yes	Array of objects with 'description' (required, max 1000 chars) and 'xp' (optional, 0-10000, default 0)
`lessonId`	string	yes	The lesson ID

`omumu_checklist_item_create`

Adds a new checklist item to a lesson. Checklist items are actionable tasks that students complete to earn XP. Items are auto-appended at the end of the list.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	yes	The checklist item text, max 1000 chars (what the student should do)
`lessonId`	string	yes	The lesson ID to add the checklist item to
`xp`	integer	no	XP points awarded when completed, 0-10000 (default 0)

`omumu_checklist_item_delete`

Permanently deletes a checklist item and any associated user progress. Remaining items are re-sorted automatically.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID to delete

`omumu_checklist_item_list`

Lists all checklist items for a specific lesson, ordered by sort order.

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`lessonId`	string	yes	The lesson ID

`omumu_checklist_item_reorder`

Changes the sort order of a checklist item within its lesson.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID
`newSortOrder`	integer	yes	The new sort order position (0-based)

`omumu_checklist_item_update`

Updates the description and/or XP value of an existing checklist item.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID
`description`	string	no	New description text, max 1000 chars (optional)
`xp`	integer	no	New XP value, 0-10000 (optional)

`omumu_course_create`

Creates a new course in the user's site with the specified title and tagline. The course is live immediately for learners with access — there is no draft step — and the result's publicUrl is the address of its learner-facing course page, which requires sign-in and access to the course. After creating, add modules (omumu_module_create) and lessons (omumu_lesson_create) to build the

course structure. To grant access via an opt-in form, pass the course ID to omumu_optinform_update(courseId).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`tagline`	string	no	The course tagline
`title`	string	yes	The course title

`omumu_course_delete`

Permanently deletes a course and all its content

- **Write category:** standard write
- **Access:** requires the `COURSE_DELETE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID to delete

`omumu_course_duplicate`

Creates a complete copy of an existing course including all modules and lessons

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID to duplicate

`omumu_course_get`

Retrieves detailed information about a specific course

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`includeModules`	boolean	no	Whether to include modules and lessons

`omumu_course_list`

Lists all courses in the user's site with optional module inclusion

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`includeModules`	boolean	no	Whether to include modules in the response

`omumu_course_update`

Updates the basic information of a course. The change is live immediately for learners with access — there is no draft step — and the result's publicUrl is the address of its learner-facing course page, which requires sign-in and access to the course.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`description`	string	no	New course description
`gatingMode`	string	no	Course gating mode: OPEN (default — course access opens every module) or SEQUENTIAL (completing a module unlocks the next; the unlocking is built in)
`tagline`	string	no	New course tagline
`title`	string	no	New course title
`watchThresholdPercent`	integer	no	How much of a video counts as watched, in percent. One of 25, 50, 75, 90, 100 (default 75)

`omumu_door_create`

Creates a door on a quiz: the next step (heading, text, button label/URL or an embedded checkout, optional tag) offered on the result page, chosen from the respondent's answers rather than the score. Doors are tried in sortOrder and the first whose condition holds wins; a door with empty clauses always holds and is the default — put it last. A clause is {kind: "ANSWER_CHOSEN", questionId, answerId} (a specific answer was chosen) or {kind: "QUESTION_ANSWERED", questionId} (the question has any answer, non-blank for free text). Every clause in a door's condition must hold (AND only — no OR, no NOT); express "either of these answers" as two doors with the same copy. Every clause's question (and answer, for ANSWER_CHOSEN) must belong to this quiz. Works on both outcome and score quizzes. On a score quiz, heading and text support placeholders: Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write

- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`buttonLabel`	string	no	The call-to-action button's label
`buttonUrl`	string	no	The button's URL: absolute http(s) or a site-relative path
`checkoutEmbedUrl`	string	no	An embedded checkout shown in place of the button: an https URL on https://app.edpay.no (the provider's embed link for the offer, e.g. https://app.edpay.no/offer/518/10998/book?embed=true); framed as a sandboxed iframe on the result page
`clauses`	array	no	Inline answer-condition clauses: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWERED" questionId, answerId?}]. Empty or omitted makes this the default door.
`heading`	string	no	Heading shown on the result page
`name`	string	yes	Internal name for this door (e.g. "Book a call")
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Evaluation order, first match wins (defaults to last)
`tag`	string	no	Tag applied to the respondent when this door wins, so automations can start the right follow-up
`text`	string	no	Body copy shown under the heading
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML

`omumu_door_delete`

Deletes a door and its clauses.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`doorId`	string	yes	The door ID to delete

`omumu_door_update`

Updates a door's copy, checkout embed, sort order, tag and/or clauses. A clauses array replaces this door's condition wholesale rather than appending to it — include every clause you want kept. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`buttonLabel`	string	no	New button label
`buttonUrl`	string	no	New button URL
`checkoutEmbedUrl`	string	no	An embedded checkout shown in place of the button: an https URL on https://app.edpay.no (the provider's embed link for the offer, e.g. https://app.edpay.no/offer/518/10998/book?embed=true); framed as a sandboxed iframe on the result page; null removes it and brings the button back
`clauses`	array	no	Replaces this door's clauses wholesale: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWE questionId, answerId?}]
`doorId`	string	yes	The door ID
`heading`	string	no	New heading
`name`	string	no	New internal name
`sortOrder`	number	no	New evaluation order
`tag`	string	no	New tag applied when this door wins
`text`	string	no	New body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML

`omumu_generate_image`

Generates an image from a text prompt using AI and stores it as a resource. Optionally attach the generated image directly to a course, module, or lesson. When targeting an entity, aspect ratio defaults to 16:9 (ideal for course illustrations).

- **Write category:** standard write

- **Access:** requires the `MEDIA_GENERATE` permission

Parameter	Type	Required	Description
`aspectRatio`	string	no	Aspect ratio: 1:1, 16:9, 9:16, 4:3, 3:4. Default: 16:9 when targeting, 1:1 otherwise.
`highQuality`	boolean	no	Use high quality model (slower, requires Premium/Enterprise tier). Default: false
`prompt`	string	yes	Text description of the image to generate. Be descriptive for best results.
`targetId`	string	no	ID of the entity to attach the image to. Required if targetType is provided.
`targetType`	string	no	Entity type to attach image to: 'course', 'module', or 'lesson'. If provided, targetId is also required.

`omumu_lesson_create`

Creates a new lesson within a module. Returns a compact summary {id, title, sortOrder} only — the submitted description is not echoed back, to keep responses small when authoring long-form content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	The lesson description
`moduleId`	string	yes	The module ID
`title`	string	yes	The lesson title

`omumu_lesson_resource_create`

Creates a new content block within a lesson. Content can be Markdown or QuillDelta JSON (auto-detected). Optionally attaches an uploaded SiteRe-

source (e.g. a video) via resourceId. Use this to add rich text content to lessons - the 'meat' of course content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	yes	The content in Markdown or QuillDelta JSON format. Markdown supports: # headers, **bold**, *italic*, links, - bullets, 1. numbered lists, > blockquotes, `code`
`lessonId`	string	yes	The lesson ID to add content to
`resourceId`	string	no	Optional SiteResource ID (e.g. an uploaded video) to attach to this lesson part. Must belong to the same site as the lesson.
`title`	string	yes	Title of the content block

`omumu_lesson_resource_delete`

Permanently deletes a content block from a lesson.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`lessonResourceId`	string	yes	The lesson resource ID to delete

`omumu_lesson_resource_list`

Lists all content blocks for a specific lesson.

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`lessonId`	string	yes	The lesson ID

`omumu_lesson_resource_update`

Updates an existing content block. Content can be Markdown or QuillDelta JSON (auto-detected). Optionally replaces the attached SiteResource. Pass an empty string for resourceId to clear the attachment.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON (optional)
`lessonResourceId`	string	yes	The lesson resource ID
`resourceId`	string	no	New SiteResource ID to attach (optional). Pass empty string to clear the current attachment. Must belong to the same site as the lesson.
`title`	string	no	New title (optional)

`omumu_lesson_update`

Updates a lesson's title and/or description. Description can be Markdown or QuillDelta JSON (auto-detected). Returns a compact summary {id, title, sortOrder} only — the submitted description is not echoed back, to keep responses small when authoring long-form content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New lesson description (Markdown or QuillDelta JSON)
`lessonId`	string	yes	The lesson ID
`title`	string	no	New lesson title

`omumu_module_copy`

Copies a module (including all its lessons and resources) from its current course into a different target course. Both the source module's course and the target course must belong to the user's site.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`moduleId`	string	yes	The ID of the module to copy
`targetCourseId`	string	yes	The ID of the course to copy the module into

`omumu_module_create`

Creates a new module within a course

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`description`	string	no	The module description
`title`	string	yes	The module title

`omumu_module_list`

Lists all modules in a specific course with optional lesson inclusion

- **Write category:** read-only

- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`includeLessons`	boolean	no	Whether to include lessons in each module

`omumu_module_update`

Updates a module's title and/or description. Description can be Markdown or QuillDelta JSON (auto-detected).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New module description (Markdown or QuillDelta JSON)
`moduleId`	string	yes	The module ID
`title`	string	no	New module title

`omumu_note_create`

Creates a note on a quiz: answer-conditioned copy shown on the result page between the lowest-scale section and the door, whenever its condition holds. Unlike a door, every matching note is shown, in sortOrder (notes compose rather than compete) — there is no first-match-wins and no default note. A clause is {kind: "ANSWER_CHOSEN", questionId, answerId} (a specific answer was chosen) or {kind: "QUESTION_ANSWERED", questionId} (the question has any answer, non-blank for free text; use this to quote a free-text answer back with [answer:]). Every clause in a note's condition must hold (AND only — no OR, no NOT). Empty or omitted clauses makes the note always show. Every clause's question (and answer, for ANSWER_CHOSEN) must belong to this quiz. A note has no button and no tag. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale]

(scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`clauses`	array	no	Inline answer-condition clauses: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWERED" questionId, answerId?}]. Empty or omitted makes this note always show.
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Display order among notes (defaults to last)
`text`	string	yes	The note's body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML

`omumu_note_delete`

Deletes a note and its clauses.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note ID to delete

`omumu_note_update`

Updates a note's text, sort order and/or clauses. A clauses array replaces this note's condition wholesale rather than appending to it — include every clause you want kept. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion]

(the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`clauses`	array	no	Replaces this note's clauses wholesale: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSW questionId, answerId?}]
`noteId`	string	yes	The note ID
`sortOrder`	number	no	New display order among notes
`text`	string	no	New body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML

`omumu_option_point_batch`

Outcome quizzes only. Efficiently set up quiz scoring by creating or updating multiple option points in a single call. If an answer-bucket pair already exists, updates its weight. Use this after creating questions, answers, and buckets to configure the scoring matrix. Example: [{answerId: 'a1', bucketId: 'b1', weight: 2.0}, {answerId: 'a1', bucketId: 'b2', weight: 1.0}]

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPoints`	array	yes	Array of option points, each with: answerId (required), bucketId (required), weight (optional, default 1.0)

`omumu_option_point_create`

Outcome quizzes only. Creates an option point linking an answer to a bucket with a weight. When a user selects this answer, the weight is added to the bucket's score. After all questions, the bucket with the highest total score becomes the user's result. Example: Answer 'I exercise daily' → Bucket 'Fitness Enthusiast' with weight 2.0. A score quiz does not use option points — give its answers points instead.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`answerId`	string	yes	The answer ID to link
`bucketId`	string	yes	The bucket ID to assign points to
`weight`	number	no	Point weight (default: 1.0). Higher weights have more influence on final bucket selection.

`omumu_option_point_delete`

Outcome quizzes only. Removes the link between an answer and a bucket, so selecting this answer no longer contributes to that bucket's score.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPointId`	string	yes	The option point ID to delete

`omumu_option_point_list`

Outcome quizzes only. Lists all option points for a specific question, showing which answers are linked to which buckets and their weights.

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

Parameter	Type	Required	Description
`questionId`	string	yes	The question ID

`omumu_option_point_update`

Outcome quizzes only. Updates an option point's weight to adjust scoring influence.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPointId`	string	yes	The option point ID
`weight`	number	no	New weight value

`omumu_question_create`

Creates a new question in a quiz. After creating a question, add answer options with omumu_answer_create. RADIO type shows single-choice radio buttons, CHECKBOX allows multiple selections. On a score quiz, assign the question to a scale with scaleId (created with omumu_scale_create) so its answers' points feed that scale; a question with no scaleId is an unscored background question.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`header`	string	yes	The question text
`quizId`	string	yes	The quiz ID
`scaleId`	string	no	Scale this question feeds (score quizzes only); omit for an unscored question
`type`	string	no	Question type: RADIO, CHECKBOX, TEXTAREA, RADIO_OTHER, CHECKBOX_OTHER

`omumu_question_update`

Updates a question's header, type and/or scale assignment (score quizzes only). The scale must belong to the same quiz.

- **Write category:** standard write

- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`header`	string	no	New question text
`questionId`	string	yes	The question ID
`scaleId`	string	no	Scale this question feeds, or null to make it unscored (score quizzes only)
`type`	string	no	New question type

`omumu_quiz_build`

Creates a whole quiz — fields, scales, questions with answers, result buckets, doors, notes and (outcome quizzes) option points — from a single definition object, all or nothing: every cross-reference is validated before anything is written, and a bad reference fails the call naming every problem found, leaving no quiz, scale, question, answer, bucket, door, note or option point behind. Use this instead of the granular tools (omumu_quiz_create, omumu_scale_create, omumu_question_create, omumu_answer_create, omumu_bucket_create, omumu_door_create, omumu_note_create, omumu_option_point_create) when authoring a quiz from scratch; those granular tools remain the way to edit an existing quiz afterwards — this tool only creates. definition.quiz mirrors omumu_quiz_create's fields: {label (required), scoring: OUTCOME|SCORE, completionTag?, captureTag?, askFirstName?, noClearLowestText?, noClearLowestTextEncoding?, preQuizOptinFormId?, postQuizOptinFormId?, destinations? (e.g. ["HIGHLEVEL"]), highLevelResultLinkField?}. destinations lists where, besides Omumu, the quiz delivers each identified respondent once their result is saved; the only destination is HIGHLEVEL, which upserts the respondent as a contact (first name, email, phone) in the site's HighLevel sub-account with the tags this quiz applied to them. Adding HIGHLEVEL is refused unless the site's plan includes CRM_DESTINATIONS and a Headmaster has connected HighLevel (omumu_site_context reports highLevelConnected); no tool reads or writes the connection itself. highLevelResultLinkField names a custom contact field in that sub-account (its field key, e.g. contact.quiz_result_link; the contact. prefix is dropped) that receives the respondent's result link at completion, written before the tags so a workflow fired by the completion tag can use it; it needs the HIGHLEVEL destination, and null removes it. definition.scales (SCORE quizzes only) is [{key (required,

your own id — any string unique among scales), name, sortOrder?, lowestScaleText?, lowestScaleTextEncoding?, levels?: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]}]. definition.questions is [{key (required, unique among questions), header, type: RADIO|CHECKBOX|TEXTAREA|RADIO_OTHER|CHECKBOX_OTHER, sortOrder?, scaleKey? (must name a scale in this same definition), answers: [{key (required, unique among every answer in this definition), label, value?, points? (SCORE quizzes), sortOrder?, tag? (applied at completion to an identified respondent who chose this answer)}]}]. definition.buckets is [{key (required, unique among buckets), name, description?, descriptionEncoding?, minPercent? (SCORE quizzes, 0-100), tag?, redirectUrl?, successMessage?, emailSequenceId?, courseId?, videoResourceId?}] — emailSequenceId, courseId and videoResourceId must already exist on this site. definition.doors is [{name, heading?, text?, textEncoding?, buttonLabel?, buttonUrl?, checkoutEmbedUrl?, tag?, sortOrder?, clauses?: [{kind: ANSWER_CHOSEN|QUESTION_ANSWERED, questionKey, answerKey?}]}] — doors have no key, nothing references one; tried in sortOrder, first matching condition wins, empty clauses is the default door. definition.notes is [{text, textEncoding?, sortOrder?, clauses?: [...]}] — same clause shape as doors, but every matching note is shown, not just the first. A clause's questionKey/answerKey name a question/answer in this same definition, never a real id; an ANSWER_CHOSEN clause's answerKey must belong to its questionKey's question. All clauses in a condition must hold (AND only). definition.optionPoints (OUTCOME quizzes only) is [{answerKey, bucketKey, weight?}], both keys naming entries in this same definition. scales is only valid on a SCORE quiz; optionPoints is only valid on an OUTCOME quiz. The response returns quizId, publicUrl (the quiz's result/take link), and the real ids created for every key you chose: scaleIds, questionIds, answerIds, bucketIds (each a {yourKey: realId} map), plus doorIds and noteIds (plain lists, since doors and notes have no keys). omumu_quiz_get is this tool's natural counterpart: fetch an existing quiz with includeQuestions, includeScales, includeBuckets, includeDoors and includeNotes all true, and its output — with ids used in place of keys — is already shaped like a definition you could feed back into this tool (minus doors'/notes' clauses, which name questions/answers by id there and must be rekeyed). Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion]

(the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`definition`	object	yes	The whole quiz: {quiz, scales?, questions?, buckets?, doors?, notes?, optionPoints?}. See the tool description for the full shape of each.

`omumu_quiz_create`

Creates a new quiz (question set). The scoring mode decides how it's judged: OUTCOME (default) sorts the respondent into one of several named buckets by plurality; SCORE accumulates points per named scale (see omumu_scale_create) and reports a percent per scale plus a total. Quizzes can optionally link to opt-in forms for email capture: preQuizOptinFormId shows an email capture form before the quiz starts (for non-logged-in users), postQuizOptinFormId shows one after quiz completion — that post-quiz form also submits itself as soon as the email field validates, before the respondent presses the final button (early contact capture), so the contact exists even if they never finish. After creating an outcome quiz, add questions with omumu_question_create, then add answer options with omumu_answer_create, create result buckets with omumu_bucket_create, and finally link answers to buckets using omumu_option_point_create. After creating a score quiz, create its scales with omumu_scale_create, then questions assigned to a scale, then answers carrying points. On a score quiz, noClearLowestText is shown on the result page when the respondent's scales are tied or there are fewer than two scales (no clear lowest); leave unset for no such section. noClearLowestText supports result-copy placeholders (see omumu_note_create for the full set). When askFirstName is true, an anonymous taker sees a first-name field on the start screen before the first question; the name is stored on the lead, copied to the user record on identification (never overwriting an existing name), and used for the [firstName] placeholder and the result page's greeting. completionTag, when set, is applied to an

identified respondent when their result is saved, regardless of which bucket won — so an automation can tell a finisher apart from someone who only left an email (e.g. FORM_SUBMIT on the post-quiz form, then WAIT, then HAS_TAG on this tag to find abandoners). captureTag, when set, is applied to the respondent at early contact capture — the moment the post-quiz form's email is valid, before the final button — and is never removed at completion, so a CRM workflow or automation has a tag to fire on for someone who left their email but may not finish. A respondent who never reaches the form never gets it. Each answer can carry its own tag too (see omumu_answer_create).

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`askFirstName`	boolean	no	Ask an anonymous taker for their first name on the start screen before the first question (default false)
`captureTag`	string	no	Tag applied to the respondent at early contact capture, the moment the post-quiz form's email is valid; never removed at completion
`completionTag`	string	no	Tag applied to an identified respondent when their result is saved, regardless of which bucket won
`label`	string	yes	The quiz name/label
`noClearLowestText`	string	no	Score quizzes only: text shown when there is no clear lowest scale (tie or all-equal)
`noClearLowestTextEncoding`	string	no	How noClearLowestText is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`scoring`	string	no	Scoring mode: OUTCOME (default) or SCORE

`omumu_quiz_get`

Gets a quiz with optional questions, result buckets and scales. Always returns the scoring mode, askFirstName, completionTag, captureTag, destinations, highLevelResultLinkField (when set) and, for a score quiz, noClearLowestText/noClearLowestTextEncoding. Returns preQuizOptinFormId and postQuizOptinFormId if linked opt-in forms are configured for email capture. For an outcome quiz, set includeQuestions=true, includeBuckets=true, and includeOptionPoints=true to see the complete scoring configuration — option points link answers to buckets with weights, and the bucket with the highest total weight from selected answers wins. With includeQuestions=true each answer also returns its tag, if any. For a score quiz, set includeQuestions=true and includeScales=true — each question returns its scaleId and each answer its points; each scale also returns its levels, lowestScaleText and lowestScaleTextEncoding; includeBuckets=true also returns each bucket's minPercent, descriptionEncoding and videoResourceId (its profile configuration). Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

Parameter	Type	Required	Description
`includeBuckets`	boolean	no	Include result buckets
`includeDoors`	boolean	no	Include doors, each with its answer-condition clauses
`includeNotes`	boolean	no	Include notes, each with its answer-condition clauses (score quizzes only)
`includeOptionPoints`	boolean	no	Include option points (answer-to-bucket scoring weights, outcome quizzes only). Requires questions to be fetched.
`includeQuestions`	boolean	no	Include questions and answers
`includeScales`	boolean	no	Include scales, each with its levels and lowest-scale text (score quizzes only)
`quizId`	string	yes	The quiz ID

`omumu_quiz_list`

Lists all quizzes for the current site

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

No parameters.

`omumu_quiz_update`

Updates a quiz's properties including label, scoring mode, linked opt-in forms for email capture, (score quizzes only) the no-clear-lowest text, the ask-first-name flag, the completion tag, the capture tag and the destinations. Pre-quiz opt-in forms appear before the quiz starts for non-logged-in users. Post-quiz opt-in forms appear after completion. Set to null to remove the link. destinations lists where, besides Omumu, the quiz delivers each identified respondent once their result is saved; the only destination is HIGHLEVEL, which upserts the respondent as a contact (first name, email, phone) in the

site's HighLevel sub-account with the tags this quiz applied to them. Adding HIGHLEVEL is refused unless the site's plan includes CRM_DESTINATIONS and a Headmaster has connected HighLevel (omumu_site_context reports highLevelConnected); no tool reads or writes the connection itself. highLevelResultLinkField names a custom contact field in that sub-account (its field key, e.g. contact.quiz_result_link; the contact. prefix is dropped) that receives the respondent's result link at completion, written before the tags so a workflow fired by the completion tag can use it; it needs the HIGHLEVEL destination, and null removes it.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`askFirstName`	boolean	no	Ask an anonymous taker for their first name on the start screen before the first question
`captureTag`	string	no	Tag applied to the respondent at early contact capture, the moment the post-quiz form's email is valid, or null to remove
`completionTag`	string	no	Tag applied to an identified respondent when their result is saved, regardless of which bucket won, or null to remove
`destinations`	array	no	Where, besides Omumu, the quiz delivers each identified respondent: ["HIGHLEVEL"] or [] to stop. The quiz ends up with exactly these
`highLevelResultLinkField`	string	no	HighLevel custom contact field key (e.g. contact.quiz_result_link) that receives each respondent's result link at completion; needs the HIGHLEVEL destination; null to remove
`label`	string	no	New label
`noClearLowestText`	string	no	Score quizzes only: text shown when there is no clear lowest scale, or null to remove
`noClearLowestTextEncoding`	string	no	How noClearLowestText is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`postQuizOptinFormId`	string	no	ID of opt-in form to show after quiz completion (for non-logged-in users), or null to remove
`preQuizOptinFormId`	string	no	ID of opt-in form to show before quiz (for non-logged-in users), or null to remove
`quizTag`	string	yes	Scoring The quiz OUTCOME or SCORE

`omumu_resource_delete`

Permanently deletes a SiteResource. Fails if the resource is currently referenced by any lesson, page, or other entity — remove those references first.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_DELETE` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	The resource ID to delete

`omumu_resource_get`

Returns the full metadata for a single SiteResource, including CDN URI, thumbnail, MIME type, file size, and video processing status (when applicable).

- **Write category:** read-only
- **Access:** requires the `RESOURCE_READ` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	The resource ID

`omumu_resource_list`

Lists uploaded files (videos, images, PDFs, etc.) on the current site. Optionally filter by a MIME type prefix (e.g. ‘video/’ or ‘image/png’) and paginate with limit/offset. Returns total count so callers can implement ‘load more’ paging.

- **Write category:** read-only
- **Access:** requires the `RESOURCE_READ` permission

Parameter	Type	Required	Description
`limit`	number	no	Maximum resources to return (default: 25, max: 100)
`mimeTypePrefix`	string	no	Optional prefix match against the resource's MIME type, e.g. 'video/' to return all videos or 'image/png' to return only PNG images. Omit to list all types.
`offset`	number	no	Number of resources to skip for pagination (default: 0)

`omumu_resource_update`

Updates the title and/or description of a SiteResource. Does not change the underlying file — re-upload for that. At least one of title/description must be provided.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New description (optional)
`resourceId`	string	yes	The resource ID
`title`	string	no	New title (optional)

`omumu_resource_upload_presign`

Prepares the upload of any file type the site accepts (PDF, image, audio, text, office document, archive, video) without the bytes passing through the MCP channel. Check 'uploadProtocol' in the result. 'put': send exactly 'length' bytes of the file as the body of an HTTP PUT to 'uploadUrl' (no Authorization header; the URL is a single-use ticket that expires at 'expiresAt'). The response is 201 with the new resource (id, fileName, fileSize, uri); use its id with omumu_lesson_resource_create or omumu_set_image. A failed PUT needs a new presign. 'tus': a video; follow omumu_resource_upload_video_presign's

instructions with the same fields (resourceId is usable right away). Uploads count against the site's per-file and total storage limits.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`filename`	string	yes	Original file name including extension, e.g. 'worksheet.pdf'. Becomes the resource's file name.
`filetype`	string	yes	MIME type of the file, e.g. application/pdf, image/png, audio/mpeg, application/zip, video/mp4
`length`	number	yes	Exact file size in bytes; the upload is rejected if the body differs

`omumu_resource_upload_video_presign`

Creates a SiteResource for a video (videoProcessingStatus QUEUED) and returns a presigned TUS (tus.io) upload the caller PATCHes bytes to directly — the video never passes through the MCP server or the JSON-RPC channel. Use the returned resourceId right away with omumu_lesson_resource_create or omumu_set_image; poll omumu_resource_get for videoProcessingStatus once the upload finishes and Bunny's webhook reconciles it. To perform the upload: POST to the returned 'endpoint' with headers AuthorizationSignature, AuthorizationExpire, VideoId (the 'guid' field), LibraryId, Upload-Length (the file size), and Tus-Resumable: 1.0.0 — Bunny responds with a Location header, which is the URL to PATCH the file bytes to (Content-Type: application/offset+octet-stream, Upload-Offset: 0, Tus-Resumable: 1.0.0).

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`filename`	string	yes	Original file name of the video, e.g. 'lesson-1.mp4'
`filetype`	string	yes	MIME type of the video: video/mp4, video/webm, video/ogg, video/quicktime, or video/x-matroska
`length`	number	yes	Total file size in bytes

`omumu_scale_create`

Creates a named scale on a score quiz (e.g. "Listening"). Assign questions to it with omumu_question_create/update's scaleId; the scale's maximum is derived from its questions' answer points, never set here. Optionally carries its levels inline and a lowest-scale text. Each level is a percent range with copy only (no tags, sequences or grants): the level with the highest minPercent at or below the respondent's percent on this scale is shown on its result card. lowestScaleText is shown once on the result page when this scale is the respondent's lowest by percent. Level text and lowestScaleText support result-copy placeholders (see omumu_note_create for the full set).

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`levels`	array	no	Inline levels: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]
`lowestScaleText`	string	no	Text shown when this scale is the respondent's lowest
`lowestScaleTextEncoding`	string	no	How lowestScaleText is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`name`	string	yes	The scale's name
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Display order (defaults to last)

`omumu_scale_delete`

Deletes a scale. Questions assigned to it become unscored background questions rather than being deleted.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`scaleId`	string	yes	The scale ID to delete

`omumu_scale_update`

Updates a scale's name, sort order, levels and/or lowest-scale text. A levels array replaces the scale's levels wholesale rather than appending to them — include every level you want kept.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`levels`	array	no	Replaces this scale's levels wholesale: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]
`lowestScaleText`	string	no	New lowest-scale text, or null to remove
`lowestScaleTextEncoding`	string	no	How lowestScaleText is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`name`	string	no	New name
`scaleId`	string	yes	The scale ID
`sortOrder`	number	no	New display order

`omumu_set_image`

Attach an existing image resource to a course, module, or lesson. Use this to reuse an already generated image or to change an entity's image.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	ID of the image resource to attach
`targetId`	string	yes	ID of the entity to set the image on
`targetType`	string	yes	Entity type: 'course', 'module', or 'lesson'

CHAPTER 3

Selling

You have something worth paying for. This chapter is about the distance between that and money arriving: putting a price on it, giving buyers a checkout that works, and getting them into the product the moment they pay — all from your own site, on your own terms.

On Omumu you don't fill in a product listing on someone else's marketplace. You tell your AI assistant what you want to sell and for how much, and it sets up the offer, the checkout, and the delivery. You can pre-sell a course that doesn't exist yet, bolt an extra purchase onto your checkout, give a course away free to grow your list, invoice a company instead of asking for a card, or hand someone access as a gift — and, before any of it, have your assistant research the market and architect the offer from what buyers actually say. Every one of those is a few sentences of conversation. The rest of this chapter shows you the sentences — and, for each, the buttons in the admin UI that do the same thing. A path such as *Sales and marketing → Offers* names a group in the admin's navigation — the rail on the left on a laptop, behind the Menu button on a phone — and the page under it.

Concepts

Selling on Omumu revolves around one object and one guarantee.

The object is the **offer**: a sellable bundle scoped to your site. An offer holds a list of products (courses, skills, and optionally a platform subscription plan), a price, a validity window, and an optional tag that triggers automations when someone buys. The offer — not the course — is what has a checkout URL. One course can be sold through many offers at different prices and framings; an offer can bundle several products into one purchase.

An offer has one **status**, shown everywhere: *not published* until you give it a Valid From date, *scheduled* while that date is in the future, *live* while it is within its window and the checkout can actually take orders, *paused* when it is within its window but the checkout cannot (the status says why — a payment provider not connected, for example), and *ended* after its Valid To.

The guarantee is that **an offer's price is frozen once it is published**. The moment an offer's validity window is set and it can be bought, its price never changes again. To charge something different, you copy the offer into a new one with a new price — the old offer id keeps meaning what every buyer saw when they bought. Your sales history stays honest by construction.

A few more terms carry the rest of the chapter:

A **bump offer** is an offer attached to a parent offer and presented as an optional checkbox on the order form — "add this to my order". Only bumps that are currently live are shown, and a ticked bump fulfils as its own mini-purchase alongside the parent. Since a bump must be live (hence published) to be offered, its price is already frozen too.

A **free offer** is simply an offer priced at zero. The order form detects the zero price and fulfils immediately — no payment step at all. Understand what that means: while a free offer is live, *anyone* who submits the public order form gets the products. It is a live giveaway, not a private test.

An **Invoice Request** is a payment option for buyers who don't pay by card at checkout — typically companies. Choosing it grants the buyer full access immediately, at the offer's real price, and records an invoice request for you to bill against later. Grant now, bill later. Unlike a free offer it sits behind an explicit button at a real price, so it is not open-door access.

Behind all of these sits **offer fulfillment**, and its key property is that it is **decoupled from payment**. Granting access and recording payment are separate machinery: the access side gives the buyer their student role, product access, welcome email, and fires the offer's automation tag; the payment side records the payment and sends the receipt. Card purchases run both; a

free offer runs both with an amount of zero; an Invoice Request runs both without money changing hands yet; a gift grant (below) runs only the access side. Fulfillment is also exactly-once: however many payment signals arrive — buyer returning to the page, provider webhook, a retry — a purchase fulfils once, never twice.

Because fulfillment doesn't wait for a finished product either, Omumu supports the **pre-sale offer**: an offer connected to a real but still-empty course, sold to validate demand before you build. Buyers get normal access immediately — to a product you then fill in. The platform's **validation page** takes this further: a designed, structured sales page sold through a low-ticket offer, built to test whether strangers will pay for an idea. This chapter covers the selling mechanics of pre-sales; the validation page itself and its self-improving machinery have their own chapters.

Hands-on

The walkthroughs below follow Mira, an independent developer turned educator who teaches non-programmers to automate their work with spreadsheets and AI. She runs her course business by talking to her AI assistant, which operates her Omumu site directly. Each walkthrough leads with the conversation, shows what happens underneath, and gives the admin-UI route.

Build the offer from what buyers say

Before Mira prices anything, she wants to know what Norwegian solo consultants actually say about their repeat work. She tells her assistant:

> "Start an offer document for Automate Your Week — for solo consultants in Norway who lose hours to repeat admin. Research what they say, then architect the offer. Build it on my existing course."

The assistant opens an **Offer DNA** document (`omumu_offer_dna_create`) from the interview — the idea, the audience, any price idea, the intended size of the thing (a single video, a few lessons, a multi-module programme, or undecided), and either an existing course or a pasted outline to build on. Then it does the work itself, in two rounds. `omumu_offer_dna_research_brief`

hands it the method rules and the exact shape the research must take; the assistant searches the web, and submits its cited findings with `omumu_offer_dna_research_submit`. `omumu_offer_dna_architect_brief` hands it the research verbatim, the method and the output schema; the assistant drafts the stack, the bonuses, the guarantee, name candidates and a score, and submits them with `omumu_offer_dna_architect_submit`. Degenerate or empty submissions are refused. None of this spends Omumu's AI budget: the thinking is your assistant's, and the platform validates and stores it. `omumu_offer_dna_get` returns the whole document; `omumu_offer_dna_list` lists the site's documents with their completion state.

From there the offer and its page are ordinary work: `omumu_offer_create` with the price the architecture settled on, and a sales page written from the document (the Pages and Sales Pages chapter). The document stays as the evidence behind the offer.

There is no admin-UI path for this: the Offer DNA is your assistant's instrument.

Sell your course from your own website

Mira's flagship course "Automate Your Week" is ready. She tells her assistant:

> "Create an offer for Automate Your Week at 1 490 NOK, card and Vipps payments, and put it on sale from Monday."

The assistant creates the offer (`omumu_offer_create`) with the title, the price in cents, the course, the payment providers, and a `validFrom` date. Only a title and price are strictly required — currency defaults to NOK, and products, payment options, pages, and dates can all be added afterwards (`omumu_offer_add_product`, `omumu_offer_update`). Setting Valid From to now or to a past time publishes the offer; a future date schedules it. The result reports the status honestly — *live* only when the checkout takes orders, otherwise *paused* with the reason — and returns the checkout URL Mira can put behind any buy button. A payment provider the site has not connected is refused at once, so the assistant cannot create an offer that looks sellable and is not (Invoice always works).

One more refusal guards the page rather than the checkout. A new offer's sales page is a starter letter with bracketed prompts where only Mira

knows the substance, and the offer cannot be published while any remain: `omumu_offer_update` with a Valid From fails and lists each one (section, field, excerpt, where to fix it), `omumu_offer_create` with a Valid From drops the date and returns the offer *not published* with the list and a warning, and `omumu_offer_get` shows the list under `salesPagePlaceholders` for a check before trying. The edit page's Valid From refuses the same way, each prompt a link into the editor. Until the offer is published, its sales page answers visitors with a 404 and shows the Headmaster a banner saying so; a scheduled offer's page is public before launch. The Pages and Sales Pages chapter has the page itself.

Mira asks "what am I selling right now?" and the assistant lists every offer with its status, products, and pricing (`omumu_offer_list`), or pulls one offer's full detail (`omumu_offer_get`). If a draft was a mistake, it can be deleted — but only an offer that has never been sold (`omumu_offer_delete`); sold offers are permanent sales history.

Two honesty notes. First, the frozen-price rule from Concepts applies here: once this offer is published, "change the price to 1 990" means the assistant copies it into a new offer at the new price — the update tool changes the price only while the offer is still unpublished. Second, whether the offer can actually take orders also depends on your site's payment setup: card payments need your Stripe account connected and chargeable, Vipps needs your Vipps integration. Until at least one selected payment path is usable, the order form shows a neutral "not taking orders" state, the offer reads *paused*, and it unpauses by itself when you finish the setup — no republish needed.

In the admin UI: the offer wizard at *Sales and marketing → Offers → New offer* walks through the same fields, and existing offers are listed and edited at *Sales and marketing → Offers*.

Get paid for a course idea before you build it

Mira has an idea for a second course — "AI Email Triage" — and wants proof people will pay before she records a single lesson. She says:

> "Create an empty course called AI Email Triage, and sell it at an early-bird 790 NOK. I'll build it in March."

The assistant creates the skeleton course (chapter 2) and an offer around it, exactly as above. This is a pre-sale offer: a purchase fulfils like any other, so early buyers get real access immediately — to a course that is still empty. There is no special pre-order state and no gate; the "opens in March" date is a communicated expectation. Set it and the thank-you page tells buyers *"Available on [date]"* instead of showing an access button. As Mira fills the course in, the content simply appears for everyone who already bought.

Stated plainly: setting that access-opens date is **UI-only** today — it lives on the offer's edit page in the admin UI and is not exposed through the assistant's tools. The assistant can create and sell the pre-sale offer; Mira sets the date at *Sales and marketing → Offers → edit*.

If the idea fails the test, the exit is as honest as the entry: refund the buyers and revoke their access. And if what you want to validate is the *pitch* as much as the product, the validation page (see the Pages and Sales Pages chapter) is this same pre-sale mechanic wrapped in a designed, self-improving sales page.

Add an order bump to your checkout

Every buyer of "Automate Your Week" is a good prospect for Mira's template pack, sold as its own small offer at 290 NOK. Rather than a separate campaign, she puts it on the checkout itself:

> "Offer the template pack as an order bump on the Automate Your Week checkout."

The assistant attaches the bump (`omumu_offer_add_bump`). From then on the order form shows a checkbox — buy the course, tick to add the templates — and a ticked bump fulfils as its own mini-purchase alongside the parent: the buyer gets both products, and your records show both sales. Both offers must belong to the same site, an offer cannot be its own bump, and only a bump that is currently *live* is actually shown to buyers — so if the template-pack offer's validity window ends, the checkbox quietly disappears rather than selling something unbuyable. Removing a bump is the mirror move (`omumu_offer_remove_bump`).

In the admin UI: bumps are managed on the offer's edit page at *Sales and marketing → Offers*.

Run a free course as a giveaway

Mira made a three-lesson mini-course, "Five Formulas That Replace an Hour of Work", to grow her mailing list. She says:

> "Create a free offer for the mini-course."

An offer with a price of zero (`omumu_offer_create` with `totalAmountCents: 0` and no payment providers) skips the payment step entirely: anyone who submits the order form gets the course immediately, with the full delivery — account, student role, access, welcome email, and the offer's automation tag, which is what makes it a list-builder: tag the buyers and let your email automation (see the Audience and Email chapter) take it from there.

The honesty note is the one from Concepts, worth repeating because it bites: a live free offer is a live giveaway — the wizard's review step says so under Valid From, and a free offer's starter letter uses free wording rather than "pay 0 kr once". The public order form fulfils it *for anyone who submits*. Do not use a free offer to preview a course privately — end the offer's validity window when the giveaway should stop, or use a direct access grant (below) for individuals.

In the admin UI: create an offer in the wizard at *Sales and marketing → Offers* with price 0.

Take invoice payments from companies

An office manager wants "Automate Your Week" for her team of five and asks — as B2B buyers do — whether she can pay by invoice instead of card. Mira tells her assistant:

> "Add invoice payment as an option on the Automate Your Week offer."

Invoice Request is a payment provider like card or Vipps, set when creating the offer or added later (`omumu_offer_create` and `omumu_offer_update` both take `paymentProviders`; on update the list replaces the offer's current providers). A buyer who clicks it gets full access immediately at the offer's real price, and Omumu records an invoice request for Mira to bill against. The trade is explicit: the buyer is served now, and collecting the money is Mira's follow-up, tracked

so it isn't forgotten. Because it needs no card machinery, an offer with Invoice Request enabled can take orders even before Stripe or Vipps is connected.

The button that sends the request says "Send invoice request" in the site's language unless Mira words it herself. "Make the invoice button say *Invoice my company*" sets it (`invoiceButtonText` on `omumu_offer_create` or `omumu_offer_update`, at most 255 characters), an empty string on update brings back the default wording, and `omumu_offer_get` returns what she wrote. In the admin UI it is the *Invoice button text* field on the offer's edit page.

Stated plainly: the follow-up — seeing and settling recorded invoice requests — is admin-UI work (*Products → Invoice Requests*), not something the assistant does.

Give someone free access to a paid course

A podcast host who had Mira on as a guest should get "Automate Your Week" as a thank-you. No checkout, no zero-price offer visible to the world — just a direct grant:

> "Give jonas@example.com access to the Automate Your Week offer — note that it's the podcast thank-you."

The assistant grants access (`omumu_offer_grant_access`): Jonas gets everything a paying buyer gets on the access side — student role, access to every product in the offer, welcome email, the offer's automation tag — but no payment is recorded and no receipt is sent, because none exists. The optional message is kept as an audit note, so six months later "why does Jonas have access?" has an answer. Two constraints: the person must already exist as a user on your site, and offers containing a platform subscription plan can only be gift-granted by Omumu.

In the admin UI the equivalent is slightly different in shape: user management at *Customers → Users* grants access per *product* rather than per offer — pick the user, grant the course. Same outcome for a single course; the offer-level grant (all products in one move, offer tag fired) is the assistant's move.

Reference

`omumu_offer_add_bump`

Adds a bump offer to an existing offer. The bump appears as a checkbox on the checkout page. Both offers must belong to the same site. An offer cannot be its own bump.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`bumpOfferId`	string	yes	The bump offer ID to add
`offerId`	string	yes	The parent offer ID

`omumu_offer_add_product`

Adds a product (course or skill) to an existing offer. The offer must not be live or ended. Subscription plans (SAAS) are added via omumu_offer_add_subscription_plan instead.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID
`productId`	string	yes	The product ID to add — a course id or a skill id (the meta_product id, which equals the underlying course/skill id)

`omumu_offer_add_subscription_plan`

Adds a subscription plan (Omumu platform access) to an existing offer. When the offer is purchased, the buyer gets a new Omumu site provisioned on the plan's tier. Use SAAS_BASIC for the Omumu Basic plan.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID
`subscriptionPlanId`	string	yes	The subscription plan ID (e.g., SAAS_BASIC)

`omumu_offer_create`

Creates a purchasable offer with title, price, courses, and payment providers. Setting validFrom to now or a past time publishes the offer. status is 'live' only when checkout takes orders; otherwise 'paused' with pausedReason. A payment provider the site hasn't connected is refused (Invoice always works). If the sales page still holds unrewritten placeholders, a validFrom is dropped rather than failing the creation: the offer comes back not_published with salesPagePlaceholders filled and a matching warnings message — rewrite the page with omumu_page_update, then publish with omumu_offer_update. Returns the offer with its checkout URL.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`buttonText`	string	no	Text for the purchase button (e.g., 'Buy now')
`courseIds`	array	no	Array of course ID strings to include in the offer
`currency`	string	no	Three-letter currency code (default: NOK)
`descriptionPageId`	string	no	Page ID for the offer description shown at checkout
`invoiceButtonText`	string	no	Text for the invoice request button at checkout, in the seller's words (at most 255 characters). Leave it out for the default wording in the site's language
`paymentProviders`	array	no	Array of payment provider names: STRIPE, VIPPS, INVOICE_REQUEST. For a free offer, leave paymentProviders out
`subscriptionPlanId`	string	no	Subscription plan ID to include (e.g., SAAS_BASIC for Omumu Basic). Bundles Omumu platform access into the offer.
`tagline`	string	no	A short tagline or subtitle
`thankYouPageId`	string	no	Page ID for the post-purchase thank-you page
`title`	string	yes	The offer title
`totalAmountCents`	number	yes	Total price in cents (e.g., 4900 for 49.00 NOK)
`validFrom`	string	no	Start date/time in ISO-8601 format (e.g., 2026-04-01T00:00:00)
`validTo`	string	no	End date/time in ISO-8601 format
`vatAmountCents`	number	no	VAT amount in cents (included in totalAmountCents)

`omumu_offer_delete`

Deletes an offer. Only offers that have not been sold can be deleted.

- **Write category:** standard write
- **Access:** requires the `OFFER_DELETE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID to delete

`omumu_offer_dna_architect_brief`

Returns the complete offer-architecture task for a DNA — the method rules, the embedded research verbatim, interview context, the format-intent constraint when declared, the freshly-resolved fenced existing material, and the exact output schema. Pure read: no AI call, no budget interaction. Run your own architecture using the instructions and context, then submit with omumu_offer_dna_architect_submit. Step order: omumu_offer_dna_research_submit → omumu_offer_dna_architect_brief → omumu_offer_dna_architect_submit → omumu_offer_create and omumu_page_create.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_architect_submit`

Validates and stores an offer architecture the calling agent produced itself — no AI budget is spent. The proposal object must have architecture, compression, and score matching the schema returned by omumu_offer_dna_architect_brief. Degenerate, empty (no solutions and no stack), or malformed submissions are rejected with a descriptive error; nothing is stored on failure. Re-submitting replaces all three prior sections. Next step: create the offer with omumu_offer_create and its sales page with omumu_page_create, using the DNA's architecture.

- **Write category:** standard write

- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id
`proposal`	object	yes	The proposal object (architecture, compression, score) matching the schema from omumu_offer_dna_architect_brief

`omumu_offer_dna_create`

Starts a research-based offer document (ADR-0034) from a creator interview. Returns its id; follow with omumu_offer_dna_research_brief.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`audience`	string	yes	Who it is for — the more specific the better
`container`	string	no	The intended size of the thing (format intent, not a price bucket): TINY (a single video or one short lesson), NORMAL (a few lessons up to modest modules), FLAGSHIP (a multi-module program), or UNDECIDED (the architect infers the scale — the default)
`idea`	string	yes	What the creator wants to sell, in a sentence or two
`knownPains`	string	no	Pains the creator already suspects (the research verifies or challenges them)
`priceIdea`	string	no	The creator's price idea, if any
`scopeNotes`	string	no	Free-text medium/cadence/delivery notes, e.g. "one 45-minute screencast" or "6 modules over 6 weeks"
`seedCourseId`	string	no	An existing Course on this site to build on — the architect reads its module and lesson titles as the real deliverables
`seedOutline`	string	no	Pasted outline or lesson material drafted outside Omumu — the offer is architected around it instead of an invented curriculum

`omumu_offer_dna_get`

Returns the full Offer DNA document: interview, cited research, architecture, compression (touchstone + big four), and score.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_list`

Lists the site's Offer DNA documents, newest first, with their completion state.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

No parameters.

`omumu_offer_dna_research_brief`

Returns the complete voice-of-customer research task for a DNA — the method rules, interview context and the exact output schema the submission must match. Pure read: no AI call, no budget interaction. Run your own web research using the instructions and context, then submit with omumu_offer_dna_research_submit. Next step after submitting: omumu_offer_dna_architect_brief.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_research_submit`

Validates and stores research the calling agent ran itself — no AI budget is spent. The research object must match the schema returned by omumu_offer_dna_research_brief. Degenerate, empty-findings, or malformed submissions are rejected with a descriptive error; nothing is stored on failure. Re-submitting replaces the research section while leaving any existing architecture untouched. Next step: omumu_offer_dna_architect_brief.

- **Write category:** standard write

- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id
`research`	object	yes	The research object matching the schema from omumu_offer_dna_research_brie

`omumu_offer_get`

Gets an offer by ID with full details including checkout URL, status (with pausedReason when paused), products, pricing, payment providers, and salesPagePlaceholders (unrewritten starter prompts still on the sales page, each with section, field, and excerpt; empty when none remain or the offer has no sales page)

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID

`omumu_offer_grant_access`

Grants a user access to all products in an offer WITHOUT requiring payment. This bypasses the normal checkout/payment flow entirely. The user receives the same access as if they had paid: STUDENT role, product access entries, welcome email, and offer-tag execution. An offer containing a SAAS plan can only be granted by Omumu. The user must already exist on the same site as the offer.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`email`	string	yes	Email of the user to grant access to. Must already exist on the same site as the offer
`message`	string	no	Optional free-text reason for the grant (e.g. 'Demo account for Ole-Arvid'). Recorded in the audit log only, not stored on the access entry. Max 500 characters
`offerId`	string	yes	The offer to grant access to

`omumu_offer_list`

Lists all offers for the current site with their status (not_published/scheduled/live/paused/ended, with pausedReason when paused), products, and pricing

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

No parameters.

`omumu_offer_remove_bump`

Removes a bump offer from an existing offer.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`bumpOfferId`	string	yes	The bump offer ID to remove
`offerId`	string	yes	The parent offer ID

`omumu_offer_update`

Updates an offer's title, tagline, buttonText, invoiceButtonText, price, payment providers, or validity dates. Only provided fields are updated; others are preserved. Setting validFrom to now or a past time publishes the offer. status is

'live' only when checkout takes orders; otherwise 'paused' with pausedReason. A payment provider the site hasn't connected is refused (Invoice always works). Setting validFrom while the sales page still holds unrewritten placeholders fails, listing each one with where to fix it — rewrite the page with omumu_page_update, then retry.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`buttonText`	string	no	New button text
`currency`	string	no	New currency code
`invoiceButtonText`	string	no	New text for the invoice request button at checkout (at most 255 characters). An empty string brings back the default wording
`offerId`	string	yes	The offer ID
`paymentProviders`	array	no	Array of payment provider names: STRIPE, VIPPS, INVOICE_REQUEST. Replaces the offer's current providers
`tagline`	string	no	New tagline
`title`	string	no	New title
`totalAmountCents`	number	no	New total price in cents
`validFrom`	string	no	New start date/time in ISO-8601 format
`validTo`	string	no	New end date/time in ISO-8601 format
`vatAmountCents`	number	no	New VAT amount in cents

CHAPTER 4

Pages & Sales Pages

Your site needs two kinds of pages, and they are built differently. There are the pages a visitor expects to find — a front page, something about you, the terms and privacy pages, maybe a blog or a section of notes — which you write the way you write anything. And there is the sales page: one per offer, built on a designed template, where the words matter more than anywhere else on the site and where your assistant does the first draft.

This chapter is both. It shows the pages a new site starts with and how to make more, the sales page from the starter letter to live, and the rule the platform enforces so a page with "[rewrite me]" prompts on it never reaches a buyer. A path such as *Sales and marketing → My Pages* names a group in the admin's navigation and the page under it.

Concepts

A page has a kind, a format and an address. The kind says what the page is for: a general page, a blog post, the front page, terms, privacy policy, cookie policy, or a thank-you page. The format says how its content is stored: *Rich Text* for anything you write in the editor, *HTML* for a page you or your assistant author as markup (a plan feature), and the *designed* format of sales pages, where content lives in named slots on a template. Every page is served at `/page/<slug>`; *Serve at site root* makes it answer at `/<slug>` as well, unless the

slug is a name the platform reserves. A page is public the moment it exists — there is no draft step for ordinary pages — and it appears in the site's sitemap and in its `/llms.txt`, the page agents read to learn what a site offers.

A new site starts with four pages. A front page at `/` titled with the site's name, carrying a welcome line and a *site contents* block that lists what the site shows the public — its pages, its live offers, its podcast — plus terms, privacy and cookie-policy pages at `/terms`, `/privacy` and `/cookies`, all written as starting points for you to edit. These four kinds are *standard* pages: a site has one active page of each, and if you write a second front page or a new terms page you make it the active one from the list. The site header and footer wrap every page: the logo on the left (the logged-out menu is only *Login* and *Forgot your password?*; there is no site menu to edit), and the Terms, Privacy and Cookies links underneath.

A sales page belongs to an offer. It is the page a buyer reads before the checkout, and it points at the offer it sells — the price on the page and the buy button are wired to the offer and change with it. An offer gets its sales page when it is created (the Selling chapter), and several pages may sell one offer, with one of them the offer's own. Sales pages are not listed under *My Pages*; you reach one from its offer.

A sales page is slots on a template. The template — today there is one for selling, *Sales Page*, a full-length letter for an audience that has never heard of you — fixes which sections exist and in what order: the hero with its headline, subheadline and badges; the problem; a two-column comparison; how it works, in steps; your approach; proof; what you get; why the price; the call to action; and the FAQ. The hero and the call to action are always there; every other section can be shown or hidden, and sections with lists (badges, rows, steps, deliverables, questions) take as many items as you like. What you cannot do is reorder sections or invent new ones; the freedom is in the words. Rich slots take bold, italic and lists, and body copy takes no links, so the only click on the page is the buy button. The look is a theme that draws its colours and fonts from your brand (Getting Started), not from the page.

Placeholders never go public. A new sales page is a *starter letter*: every section filled from what the offer knows, with bracketed prompts where only you know the substance — *"[What staying stuck costs them]"*. Publishing the offer is refused while any of those remain, from the offer's edit page, from the wizard and from your assistant alike, and the refusal lists each one with a link to the spot in the editor. Until the offer is published its sales page answers

visitors with a 404; you, signed in as the Headmaster, see it with a banner saying so, exactly as buyers will once it is live. A scheduled offer's page is public before the launch, so you can send traffic to it.

HTML pages are for the assistant to write and you to touch up. On plans with custom HTML pages, a page can be stored as markup — the natural output of an assistant that designs a page for you. You can edit the source, but the everyday path is *touch-up*: open the live page as the Headmaster, click the pencil, and the text that is safe to change is highlighted; you fix a word, bold a phrase, save, and the page is live. Structure and styling stay with the source.

Evergreen notes grow instead of being dated. Beside the blog, every site has *Notes* at `/notes`: short public writing on one idea each, published the moment it is saved and meant to be revised and renamed as the idea sharpens, rather than finished on a date. A note shows no date anywhere, and the index lists the most recently grown first. Notes link to each other by title: `[[Title]]` in a note's body is a *Note link* to the note with that title, matched ignoring case, and it keeps working when that note is renamed. A title no note has yet is a *Stub*, an idea named before it is written: visitors read it as plain text, you read it as an invitation to write that note, and it becomes a link the moment a note with that title is saved. Each note lists *Links to this note*, the other notes that link to it. Because a title is what a link names, no two of a site's notes share one, and a title holds no square brackets, since a Stub could not name it. A note's address is `/notes/<id>/<title>`: the id finds it, so after a rename an old address answers with a permanent redirect to the current one. Notes are on every plan, and they are not the answer-conditioned notes on a quiz's result page (Courses & Content).

Hands-on

The pages you start with

Mira's site came with a front page, terms, privacy and cookie pages. *Sales and marketing → My Pages* lists them with their kind, address and format, filtered by chips (*Front Pages*, *Blog Posts*, *Terms*, …) or searched by title. *Edit* opens the editor, *Preview* the public page in a new tab, *Copy* duplicates one, and the lightning button on an inactive standard page makes it the active one. The *Set*

up your site checklist on the admin home keeps *Edit your front page* unticked until the front page has been saved once.

Mira says: *"Rewrite my front page: who I am, what the course does, and keep the list of what is on the site."* The assistant reads it with `omumu_page_get` (by slug or id) and writes it back with `omumu_page_update` — title, tagline, search description, contents, image or slug — keeping the site contents block, which is an embed in the content rather than text. In the admin, the editor is the page's content with a *Page Settings & SEO* panel: title, slug, *Serve at site root*, search description and kind; *Save* publishes at once.

Add a page

"Make an About page at /about-mira: I was a developer for fifteen years, now I teach people who don't code to automate the boring parts of their week."

`omumu_page_create` takes the slug, title, kind and contents — Markdown is fine for a page written in conversation; it is stored as rich text — and returns the public address. `omumu_page_list` shows every page, filtered by kind. In the admin: *New Page* asks for title, slug, kind and format (*Rich Text Editor* or *HTML Editor*, the latter only on plans that include it) and opens the editor. To put an opt-in form on a page, write `[optin-form id="…"]` on a line of its own; the Audience & Email chapter has the forms.

Write a blog

A blog post is a page of the *Blog Post* kind. The public blog lives at `/blog`, each post at `/blog/<slug>`, with RSS and Atom feeds at `/blog/feed.rss` and `/blog/feed.atom`. Create posts as any other page (`omumu_page_create` with type `blog`, or *New Page* with the kind set to Blog Post); the *Blog Posts* chip under *My Pages* lists them. There is a separate *Blog Management* page at `/admin/blog` with the same list, but nothing in the navigation links to it.

Publish Evergreen notes

Mira keeps explaining the same idea to every client. She says: *"Write a note called 'Automate the report, not the job': most people try to automate their whole role and give up; start with the one report you rebuild every Monday. Link it to [[The Monday report]]."*

The assistant calls `omumu_evergreen_note_create` with the title and a Markdown body; the `[[The Monday report]]` in it pins to that note if Mira has one, and otherwise stays a Stub. The answer carries the note's id and its public address, live at once. `omumu_evergreen_note_update` grows or renames a note, keeping whatever it is not given (title, body, meta description, sharing image); `omumu_evergreen_note_get` gives the body back as Markdown with every Note link written `[[Its current title]]`, ready to edit and send back; `omumu_evergreen_note_list` lists the notes, the most recently grown first, and the Stubs with how many notes name each, so *"what should I write next?"* has an answer; `omumu_evergreen_note_delete` takes one down. A title another note already has is refused, naming that note, and so is a title holding a square bracket; either way nothing is written. These tools take the same permissions as the page tools, and the page tools refuse a note, naming these instead.

In the admin, *Sales and marketing → Notes* lists the notes and, beside them, the *Stubs*, the most-named first; choosing a Stub opens *New note* with its title filled in. The editor takes a title, a body in the same editor pages use, and an optional meta description and sharing image URL; typing `[[` in the body suggests her note titles as she types, Up, Down and Enter (or a click) pick one into a Note link, and Escape keeps typing. *Save and publish* makes it public. Notes are not listed under *My Pages*, the page pickers on offers, quizzes, opt-in forms and funnel steps never offer one, and a note's address under `/page/` answers 404; the pencil on a note she is reading opens its editor.

Visitors find the index at `/notes` and each note with its *Links to this note*. The section has its own RSS and Atom feeds at `/notes/feed.rss` and `/notes/feed.atom`, separate from the blog's: each note is announced once, dated when it was first saved, so growing or renaming it never puts it back in a reader's feed. The sitemap lists the index and each note under its current address.

Deleting is on the note's editor, behind a confirmation page: the note's addresses answer 404, it leaves the index, the feeds and the sitemap, and every link to it in her other notes turns back into a Stub of its last title, so writing a note with that title again links them once more.

Write the sales page

Mira's offer *Automate Your Week* has a sales page full of prompts. She says: *"Rewrite my sales page from these notes: the problem is Sunday evenings lost to*

spreadsheets, the first win is a report that builds itself, refunds within 14 days, no questions."

The assistant calls `omumu_template_list` once to see the template's sections and the fields each slot takes, reads the page with `omumu_page_get`, and writes the whole slot document back with `omumu_page_update` — the full page every time, not one section; the template, the offer and the price are fixed facts the document cannot change. A field the template does not have is dropped without a word, so it checks the result. When the page is clean, publishing the offer with `omumu_offer_update` goes through; while prompts remain, it fails with the list (section, field, excerpt, editor path), and `omumu_offer_get` shows the same list under `salesPagePlaceholders` for a check before trying.

In the admin, *Sales and marketing → Offers*, the offer, *Edit sales page* opens the page on its real look: click any text and type; *Show or hide this section* on the optional sections; *Add badge*, *Add row*, *Add step*, *Add testimonial*, *Add item*, *Add question* for the lists; the buy button's label is editable, its destination is not. *Save page* makes it live for anyone who can see the page; *View live page* shows it. From the public page, the pencil in the corner brings the Headmaster back to the editor.

Publish, and what buyers see

Publishing is the offer's *Valid From* (the Selling chapter). With placeholders left, the edit page refuses and lists them, each a link to its section; fix them, save, and publish again. Once live, the page shows the site header and footer, the price from the offer, and a buy button that goes straight to the checkout. Pausing or ending the offer keeps the page up; unpublishing hides it again.

Author a page in HTML

"Design me a one-page launch announcement in HTML with a countdown-style headline, our colours, and a link to the sales page."

`omumu_page_create` with `encoding: MANAGED_HTML` stores the markup as given and serves it inside the site header and footer. Editing it in the admin — the source editor and the touch-up alike — needs a plan with custom HTML pages; without one the page is shown read-only there. Afterwards Mira opens the live page, clicks the pencil, and lands in *Touch up copy*: highlighted text can be edited and formatted (bold, italic, underline, highlight), dimmed text is

protected, and *Save* says *Your changes are live.* If the page changed under her meanwhile, nothing is saved and she reloads. Changing the structure means *Edit source*, which is the markup itself.

What you cannot do yet

Stated plainly: ordinary pages have no draft state — a page is public the moment it exists, and only sales pages follow their offer's status. There is no site menu to edit; the logged-out header links only to sign-in, so pages reach each other through the front page's contents block and your own links. Sales pages are not listed under *My Pages*, and nothing links the blog's own admin page. Nothing points visitors to `/notes` but your own links: the front page's contents block and the site's `/llms.txt` list general pages, not notes. A note's Markdown from `omumu_evergreen_note_get` leaves out embedded videos, and when sent back gains an empty line between paragraphs that had none and reads Markdown characters in the text as formatting, so a line starting with "-" becomes a list; the stored Quill body it also returns keeps all of it. One selling template, one theme per template, no reordering of sections, no images inside a sales page beyond the brand. And the sales page's only buy button is at the end of the letter; the hero shows the price and the access badge, not a button.

Reference

`omumu_evergreen_note_create`

Writes a new Evergreen note in the site's Notes section (/notes): a short public note on one idea, without a date, that keeps growing and links to other notes. These are not the quiz notes (omumu_note_*) and not pages. Evergreen notes are public the moment they are saved: there is no draft, and the result's publicUrl is where visitors read the note. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved. A title another note on the site already has, ignoring case, is refused and nothing is written.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`body`	string	no	The note's body: Markdown (headings, lists, quotes, code, bold, italic, links, images) or Quill Delta JSON, detected the way the page tools detect it. Use [[Title]] to link to another note.
`image`	string	no	Optional sharing image URL for link previews
`metaDescription`	string	no	Optional description for search results and link previews
`title`	string	yes	The note's title, unique among the site's notes ignoring case; at most 255 characters, with no square brackets

`omumu_evergreen_note_delete`

Deletes an Evergreen note; its URL then answers 404. Every link to it in the site's other notes turns back into the Stub [[Its last title]], so writing a note with that title again links them once more.

- **Write category:** standard write
- **Access:** requires the `PAGE_DELETE` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note's id

`omumu_evergreen_note_get`

Gets an Evergreen note: its title, publicUrl, metaDescription, image, and its body as Markdown in which every link to another note is [[Its current title]] and every Stub is [[Title]], ready to edit and pass back to

omumu_evergreen_note_update. Markdown leaves out embedded videos, and sending it back puts an empty line between paragraphs that had none and reads Markdown characters the text holds as formatting: a line starting with '-', '#', '>' or a number and a dot becomes a list, heading or quote, and text between asterisks or underscores is emphasised; 'quillBody' is the stored Quill JSON, which keeps all of it. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note's id

`omumu_evergreen_note_list`

Lists the site's Evergreen notes, the most recently grown first (id, title, publicUrl), and its Stubs: the [[Title]] titles its notes name that no note has yet, each with how many notes name it, the most-named first. A much-named Stub is a good idea to write next. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

No parameters.

`omumu_evergreen_note_update`

Grows or renames an Evergreen note; any parameter left out stays as it is. Renaming never breaks a link to the note. Evergreen notes are public the moment they are saved: there is no draft, and the result's publicUrl is where visitors read the note. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps

working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved. A title another note on the site already has, ignoring case, is refused and nothing is written.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`body`	string	no	The new body, replacing the old one. The note's body: Markdown (headings, lists, quotes, code, bold, italic, links, images) or Quill Delta JSON, detected the way the page tools detect it. Use [[Title]] to link to another note.
`image`	string	no	New sharing image URL; an empty string removes it
`metaDescription`	string	no	New description for search results; an empty string removes it
`noteId`	string	yes	The note's id
`title`	string	no	The new title, at most 255 characters, with no square brackets

`omumu_page_create`

Creates a new page. The page is live on the public site immediately — there is no draft step — and the result's publicUrl is the address it is served at. Two modes:

Slot page (recommended for sales pages): Pass 'template' (discriminator from omumu_template_list) and 'slotJson' (whole-document slot JSON validated against that template). For SALES templates, also pass 'offerId'. Wiring fields (offerId, purpose) must be tool parameters — never inside slot contents.

Classic page: Pass ‘contents’ with optional ‘encoding’ (QUILL_JSON or MANAGED_HTML). MANAGED_HTML is available but de-emphasised for selling pages — prefer slot pages instead.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`contents`	string	no	Classic page content. Raw HTML when encoding is MANAGED_HTML, Quill JSON or markdown otherwise. Not used when creating slot pages.
`encoding`	string	no	Content encoding for classic pages: QUILL_JSON (default) or MANAGED_HTML. Not used when creating slot pages.
`offerId`	string	no	Offer ID this sales page sells. Required for SALES templates; must be a valid offer belonging to this site. Never pass inside slot contents.
`slotJson`	string	no	Whole-document slot JSON validated against the named template. Required when 'template' is provided.
`slug`	string	yes	The page URL slug (e.g., 'my-course-launch')
`template`	string	no	Template discriminator from omumu_template_list (e.g. 'sales-cold-traffic'). Required for slot page creation; causes slotJson validation against the named template.
`title`	string	no	The page title
`type`	string	no	Page type: blog, frontpage, userpage, terms, privacy, cookies, template, block, offer, thank_you

`omumu_page_delete`

Deletes a page by ID

- **Write category:** standard write
- **Access:** requires the `PAGE_DELETE` permission

Parameter	Type	Required	Description
`pageId`	string	yes	The page ID to delete

`omumu_page_get`

Gets a page by ID or slug. At least one of pageId or slug must be provided. The result's 'public' field is false for a sales page whose offer is not published (visitors get a 404 there until it is), true otherwise.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`pageId`	string	no	The page ID (provide this or slug)
`slug`	string	no	The page slug (provide this or pageId)

`omumu_page_list`

Lists all pages for the current site, optionally filtered by type

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`excludeOfferPages`	boolean	no	Exclude auto-generated offer pages
`type`	string	no	Filter by page type: blog, frontpage, userpage, terms, privacy, cookies, template, block, offer, thank_you

`omumu_page_update`

Updates a page. The change is live on the public site immediately — there is no draft step — and the result's publicUrl is the address the page is served at. For slot pages (encoding SLOT_JSON), pass 'slotJson' with the full updated slot document — wiring (offerId, purpose) is immutable on update and is not re-validated here. For classic pages, pass any combination of 'title', 'tagline', 'metaDescription', 'contents', 'image', or 'slug'.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`contents`	string	no	New content for classic pages. Format depends on page encoding: raw HTML for MANAGED_HTML pages, Quill JSON for QUILL_JSON pages.
`image`	string	no	New header image URL
`metaDescription`	string	no	New meta description for SEO
`pageId`	string	yes	The page ID
`rootPublished`	boolean	no	Serve the page at the site root (/<slug>) in addition to /page/<slug>. Reserved platform names (admin, terms, courses, …) are refused. Both addresses keep working; rendered links, canonical metadata, and the sitemap use the root URL. Omit to leave unchanged.
`slotJson`	string	no	Full updated slot JSON document for slot pages (encoding SLOT_JSON). The template discriminator is read from the existing slot document. Wiring is immutable.
`slug`	string	no	New URL slug (changes the page URL)
`tagline`	string	no	New tagline
`title`	string	no	New title

`omumu_template_list`

Returns all registered page templates with their section layouts and per-section JSON schema. Call this tool first when building a slot page so you know which template discriminator to pass to omumu_page_create and what slot

fields each section expects. Each entry includes: ‘name’ (the discriminator), ‘purpose’ (e.g. SALES), ‘sections’, and ‘schema’.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

No parameters.

CHAPTER 5

Audience & Email

Nobody buys from a site they visited once. The people who buy are the ones who left an email address, got something useful for it, and heard from you a few more times. This chapter is that machinery on Omumu: the opt-in form that trades a gift for an address, the sequence that follows up on its own, the invite that gives a few people a week something free and tells the rest to wait, the quiz that sorts people by what they said, and the inbox that catches what they write back. Most of it your assistant can set up in a conversation; all of it is in the admin under *Sales and marketing*.

One rule sits under everything here, so it comes first: **sequence emails go out only through an email service you connect yourself.** Logins, receipts and access emails always go out through Omumu. Until you connect Resend or SendGrid, every sequence email is held — recorded as not sent, never delivered later — and every page in this chapter says so at the top. Connecting takes a domain you own and three DNS records.

Concepts

A contact is a person on your site. Whoever submits a form, accepts an invite or takes a quiz with their email gets a user record with no password; buying or being granted a course makes them a student. They sign in through

the one-time link in the access email, or *Forgot your password?* — there is no separate "contacts" list; *Customers → Users* is everyone.

An opt-in form asks for the address. It has a title, an optional description and trust line, a submit button, and fields — email always, a name (one field or first and last) and a phone number if you want them. The form carries its own *tag*, and what it does on submission is wiring on the form: start an *email sequence*, grant a *course*, then show a success message or redirect to a page. A form lives on a page: write `[optin-form id="…"]` on a line of its own in any page, or set it as a designed page's form (the Pages chapter). Each form counts impressions, unique visitors and submissions per day.

A sequence is a first email and its follow-ups. The first email goes after an *initial delay* from enrolment; each follow-up has its own *send after* — counted from the first email, not from the previous one, so "day 1, day 3, day 7" means exactly that. Bodies are written in the editor (or as Markdown through your assistant) and may use merge fields: `{{first_name}}`, `{{login_link}}`, `{{site_url}}`, `{{resume_link}}` and `{{email_sequence_cancel_url}}`; square brackets are not merge fields and go out as typed. Every sequence email carries an unsubscribe link. People get into a sequence through a form's setting, an invite, an automation's *Start Sequence* step, or a quiz bucket — and the sequence's page lists those doors, and its members.

An invite is scarcity you can keep honest. It gives away a course to *N* people per *window* — hourly, daily, weekly or monthly, from an opening moment in your time zone — and when the window's invites are taken, the page turns into a waitlist that is emailed when the next window opens. Nothing rolls over, nothing is auto-granted; the counter on the page moves when people act, and there is no countdown clock. Launching makes the whole thing in one go: a page, a form, three sequences (delivery, waitlist welcome, reopen) seeded in your language, and the invite — as a *draft*. Its page is a 404 to visitors and its form takes nothing until you *Go live*, which is refused while any bracketed starter prompt remains on the page or in the emails.

A quiz sorts people by their answers. Questions score towards *buckets*, and a bucket can tag the person, start a sequence, grant a course, redirect or show a message; the quiz can ask for a first name at the start and an email before or after. A score quiz goes further: the email form at the end saves the contact as soon as the address is valid, a *completion tag* marks the ones who finished so an automation can remind the rest, and the sequence a profile starts can place the person's result link with `$quiz_result_link`. It is a page of its own at

an address you share, not part of a course. Building one — questions, answers, points, buckets, and for a score quiz scales, doors and notes — is in the Courses & Content chapter; what it does for your audience is here.

A quiz can hand people on to your CRM. Omumu always keeps each result, with its tags and sequences, first. On plans that include CRM destinations, a quiz can also deliver each identified respondent — a signed-in student, or a visitor who left an email — to HighLevel: the contact (first name, email, phone) arrives in your sub-account with the tags this quiz gave them — the capture, completion, bucket and door tags, and the *answer tag* on any answer they chose — and, once they finish, a dated note holding their answers, the percent per scale, the profile, the door and the result link. The result link can also land in a custom contact field you name, so an email or SMS sent from HighLevel can carry it; a note cannot be used there. Someone who leaves the email and stops arrives once, with the *capture tag*, so HighLevel can chase them; finishing is a second delivery. Only the quiz's own tags go out, and nothing is ever removed in HighLevel. A site has one HighLevel connection, to one sub-account.

The inbox is mail to your domain. With receiving turned on for your Resend domain, everything sent to any address at it lands in one inbox shared by the site's admins, where you read, archive, reply and compose. Senders you don't know can be held until you approve them, and a noisy sender can be bundled into one row.

Some addresses stop getting your email, and Omumu stops sending it. Mail that keeps bouncing or being reported as spam hurts whether the rest of it arrives, so Omumu acts on three things by itself. An *undeliverable address* is one whose mail server said the mailbox does not exist: no site on Omumu sends it anything more, not even a sign-in link or a receipt, and its sequences end on every site as if the person had unsubscribed. A typo is the usual cause, and a changed address starts over, with nothing carried across. A *spam complaint* is someone marking your site's email as spam: your site sends them no more marketing — sequence email — and their sequences on your site end, while sign-in links, receipts, course access and replies you write by hand still go, and other sites are unaffected. A *sending pause* holds all of a site's marketing when too many of its recent emails bounce or are marked as spam; mail people need keeps going, and only Omumu lifts it.

Hands-on

Connect email first

Integrations → Email: *Resend* (recommended) or *SendGrid*. For Resend, Mira enters a domain she owns, picks a region, and gets the DNS records to add at her registrar; once the domain is verified the banner on every email page goes away and sequences send. Until then she can build everything below; it simply waits.

Capture leads with a form

Mira says: *"Make an opt-in form for the free time-audit checklist, start the welcome sequence when someone signs up, and put it on the About page."*

`omumu_optinform_create` makes the form (name, title, description); `omumu_optinform_update` wires it — `emailSequenceId`, optionally `courseId` for a free course, `successMessage` or `redirectUrl`, the button text and a trust line — and `omumu_optinform_add_field` adds a name or phone field. The page gets `[optin-form id="…"]` through `omumu_page_update`. `omumu_optinform_list` and `get` read forms back; `omumu_optinform_submissions_list` reads who submitted, and needs the separate *leads:read* permission because it returns people's data (the Agent Surface chapter).

In the admin, *Sales and marketing → Opt-in Forms → Create New Opt-in Form* walks four steps — *Identifier*, *UI & Fields*, *Automation* (the sequence and the course), *After submission* (message or redirect to a page, or straight to the course) — with a live preview. Each form's row has *Preview*, *Statistics* (impressions, visitors, submissions and conversion by day) and *Submissions* (each entry with its fields).

Write the welcome sequence

"Write a three-email welcome sequence: the checklist today, a story about my first automation on day 2, and an invitation to the course on day 5."

`omumu_email_sequence_create` takes the name, the first email's subject and content (Markdown is fine), an `initialDelay`, and `followUps` with their own delays (`2d`, `5d`, or ISO durations). `omumu_email_sequence_update` and `omumu_email_followup_update` change them; `omumu_email_sequence_get` shows the sequence with its emails and how many people are in it, how many emails were sent and how many are held; `omumu_email_sequence_list` lists them with

their member counts. Remind the assistant of the merge fields; the tools do not list them.

In the admin, *Sales and marketing* → *Email Sequences* → *New Email Sequence*: a name, the initial delay, the first email, and *Save and create next email* for each follow-up with its *Send after*. The sequence page lists the emails with their timing and a preview of each.

See who is in it

The sequence page lists its *members*: name and email, when they were enrolled, how far they are (not started, "email 2 of 3 sent", or finished), how many emails went out and how many were held, and the latest open and click. A member expands to their send log, every email with its status, time and subject. *Remove from sequence* takes a person out of this sequence only — the way to honour "please stop" without blocking them. The page's *How people get in* panel names every door: the forms, invites, automations and quiz buckets that enrol into it, each a link, or says that nothing does.

Why someone stopped getting your email

When a person's address bounces for good, Mira sees *Undeliverable* beside their name under *Customers* → *Users*, on their page (with the date since when) and in a sequence's member list; their send log says *Not sent: address undeliverable* for any step that came due afterwards, and so does the *Emails* tab on their page. If they can still sign in, Omumu tells them their address is bouncing and to contact the site to change it — which only an admin can do, with *Edit user* on the person's page. The new address starts with nothing from the old one, so they opt in again for what they want. Writing to an undeliverable address from the *Inbox* is refused with a notice before anything is sent.

When someone marks one of Mira's emails as spam, the badge reads *Marked as spam*, and steps that come due are logged *Not sent: marked this site's email as spam*. Nothing Mira does lifts it, by design: anyone can type someone else's address into a form. If the person submits one of Mira's opt-in forms again, Omumu emails them to ask *"Do you want email from Mira's site again?"* — at most once a day, with a link that works for seven days — and only pressing the button on that link's page starts her marketing to them again. The form

subscribes them as usual meanwhile, but its emails are not sent before the click; the ones that came due earlier are skipped, and the next one goes.

If too many of the site's recent emails bounce or are marked as spam, a yellow notice on the admin home and the *Email Sequences* pages says marketing email is paused. Sequence steps that fall due are held, not lost, and go at their next check once Omumu lifts the pause; contact Omumu to have it lifted.

Remove someone

Each person under *Customers → Users* has a *Delete*. A delete ends what they have on your site at once: their course access, their place in every sequence, the AIs they connected and their API keys, and they can no longer sign in. Their email address is freed, so if they sign up, opt in or buy again they start over with a fresh account that inherits nothing. Because that is hard to take back, the dialog asks you to type their email first, and there is no bulk delete. Omumu refuses to delete the site's last Headmaster, or the account that owns the site. To stop the email without removing the person, use *Remove from sequence* or block them.

Choose *Deleted users* in the list's role filter to see everyone you have deleted, with their email, when they were deleted and the date they will be erased: thirty days later. Until then, *Undo delete* gives the person their email back and lets them sign in again. Nothing the delete ended comes back with it; grant the course and enrol them again yourself. Undo is refused once someone else has signed up with that email.

On the erasure date the record goes for good, with everything that was the person's: progress, hand-ins, tags, sequence history and form submissions. What is your site's stays: a receipt keeps the buyer's name, email and address, as bookkeeping requires, and a course or email they created stays with its creator left blank.

People can delete their own account too: *Delete my account* on their account page asks for their email, runs the same delete, signs them out everywhere and tells them the date their data will be erased, and that you can restore the account until then if they ask.

Give a few away each week

"Launch an invite for the pilot group: ten places a week, opening Mondays at nine."

`omumu_invite_launch` takes the name, a course (or creates an empty one), the quota, the cadence and opening moment, the zone and the seed language, and returns the draft with its page, the editor path and the list of placeholders still to rewrite. The assistant rewrites the page with `omumu_page_update` and the three emails with `omumu_email_sequence_update`, then `omumu_invite_update` with `active: true` takes it live — refused, with the list, while a placeholder remains. `omumu_invite_get` and `omumu_invite_list` show the stage (draft, live, paused), the window ("3 of 10 taken", next boundary) and the waitlist size; `active: false` pauses it, which also lets its form and sequences be deleted.

In the admin, *Sales and marketing → Invites → Launch Invite* asks for the same things and lands on the invite's page: the links to its public page, page editor, form and three sequences, *Go live* once the copy is yours, *Deactivate* and *Activate* later, and a *window history* of how fast each window sold out.

Sort people with a quiz

Sales and marketing → Quizzes lists them with *View Results*; a quiz's page shows its public address with a copy button, the questions and the result buckets with their actions, and the email capture switches (before the quiz, or after it for visitors who are not signed in). *Results* has the overview, each respondent and their answers, and the free-text answers. Send people to the quiz from a page, an email or a sales page; they land on a page titled with the quiz.

Send quiz respondents to HighLevel

The connection is entered in the admin, never through your assistant. *Integrations → HighLevel* (for a Headmaster, on a plan with CRM destinations) asks for a Private Integration Token — made in the sub-account under Settings, then Private Integrations, with the scopes contacts.readonly and contacts.write — and the location id from Settings, then Business Profile. *Verify and save* checks both against HighLevel before anything is stored, and from then on the page shows the token only by its last four characters. You can also name the HighLevel user the notes are attributed to.

Then turn delivery on per quiz: *Deliver to HighLevel* under *Deliver respondents* on the quiz's page, or ask your assistant — *"Deliver this quiz's respondents to HighLevel"* — which sets the quiz's destinations with `omumu_quiz_update`. `omumu_site_context` tells it whether the site is connected, and the change is refused without the connection or the plan. With delivery on, *HighLevel field for the result link* on the quiz's page takes the key of a custom contact field in your sub-account — `contact.quiz_result_link`, say. Each finisher's result link is written there before their tags arrive, so a HighLevel workflow that starts on the completion tag can put the link in its email or SMS. Your assistant sets it with `highLevelResultLinkField` on `omumu_quiz_update` or `omumu_quiz_build`. The quiz's *Results* show each respondent's delivery: delivered with its time, pending, or failed with the reason; the person's page under *Customers → Users* lists their HighLevel deliveries. A failure HighLevel calls temporary is retried for about two and a half hours, one it refuses outright — a deleted token, say — is marked failed at once, and neither ever reaches the respondent's result page.

Removing the connection first lists the quizzes that deliver to it; confirming turns their delivery off and deletes the token. Each quiz stays off when you connect again, until you turn it back on.

Read and answer mail

Sales and marketing → Inbox (on plans that include it): *All*, *Unread*, *Read*, *Archived*, search, and *Compose* to write from your domain. The *Unknown senders* tab has the *Hold unknown senders* switch; held mail reaches nobody and triggers nothing until you approve the sender, which also releases everything they already sent. Your assistant reads the inbox with `omumu_email_inbox_list` and `omumu_email_inbox_get`, but cannot reply.

What you cannot do yet

Stated plainly: there is no broadcast — to email everyone on a list once, make a one-email sequence and start it from an automation (the Automations chapter), which is the intended pattern. There is no contacts page with segments; people are reached through tags and sequences. Your assistant cannot delete or restore a person, delete a sequence or an invite, or add follow-ups to an existing sequence, and deleting a paused invite's form deletes the invite with it. Sequence timing cannot be relative to the previous email. The inbox cannot

be answered over the connection. Your assistant does not see which addresses bounced or complained, and nobody on your site can lift an undeliverable address, a spam complaint or a sending pause by hand. HighLevel is the only CRM a quiz delivers to, and your assistant can turn a quiz's delivery on or off but never sees or sets the connection itself. And none of it sends until your own email service is connected.

Reference

`omumu_email_followup_update`

Updates a follow-up message in an email sequence. All fields except followUpId are optional.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON
`delay`	string	no	New delay from the first email (absolute, NOT relative to previous). Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`followUpId`	string	yes	The follow-up ID to update
`subject`	string	no	New subject line

`omumu_email_inbox_get`

Retrieves full details of a specific email including headers, body content (text and/or HTML), attachments, and metadata. Use omumu_email_inbox_list first to find email IDs.

- **Write category:** read-only
- **Access:** requires the `EMAIL_INBOX_READ` permission

Parameter	Type	Required	Description
`emailId`	string	yes	The email ID to retrieve

`omumu_email_inbox_list`

Lists received emails in the inbox with optional filtering by status or search query. Returns email summaries (id, sender, subject, status, date) with pagination. Also returns unread count for quick monitoring.

- **Write category:** read-only
- **Access:** requires the `EMAIL_INBOX_READ` permission

Parameter	Type	Required	Description
`limit`	number	no	Maximum emails to return per page (default: 25, max: 100)
`page`	number	no	Page number for pagination (default: 1)
`search`	string	no	Search in sender address, sender name, subject, or recipient address
`status`	string	no	Filter by status: UNREAD, READ, ARCHIVED, HELD, or ALL (default: the inbox, which excludes deleted and held email)

`omumu_email_sequence_create`

Creates a new email sequence with an initial email and optional follow-up messages. Content can be Markdown or QuillDelta JSON (auto-detected). After creating, link it to an opt-in form via omumu_optinform_update(emailSequenceId) to trigger the sequence when users submit the form. To send a single one-off email to a contact from an automation, create a sequence containing just one message with initialDelay PT0S and start it with the automation's start_sequence action. Subject and content may use $first_name

(the recipient's first name) and $quiz_result_link (the recipient's newest quiz result page on this site, for a sequence a quiz's result bucket starts); a variable that cannot be resolved renders as nothing.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	yes	Email content in Markdown or QuillDelta JSON
`followUps`	array	no	Array of follow-up messages with delay (offset from the first email, NOT relative to previous), subject, and content. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`initialDelay`	string	no	Delay before sending the first email. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w). Default: PT0S.
`name`	string	yes	Sequence name (unique identifier)
`subject`	string	yes	Subject line for the initial email

`omumu_email_sequence_get`

Retrieves a specific email sequence with all follow-up messages, plus memberCount, sentCount and heldCount. heldCount counts emails not delivered because the site has no sequence email service configured yet. No member identities are included.

- **Write category:** read-only
- **Access:** requires the `EMAIL_SEQUENCE_READ` permission

Parameter	Type	Required	Description
`sequenceId`	string	yes	The sequence ID to retrieve

`omumu_email_sequence_list`

Lists all email sequences for the current site with their follow-up messages and memberCount (how many people are enrolled). No member identities are included.

- **Write category:** read-only
- **Access:** requires the `EMAIL_SEQUENCE_READ` permission

No parameters.

`omumu_email_sequence_update`

Updates an existing email sequence's properties. All fields except sequenceId are optional.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON
`initialDelay`	string	no	New delay from the subscription trigger. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`name`	string	no	New sequence name
`sequenceId`	string	yes	The sequence ID to update
`subject`	string	no	New subject line for initial email

`omumu_invite_get`

Gets a single Invite by ID with full config, stage (draft, live, or paused), window state (accepted of quota, next boundary), waitlist size, and placeholders (unrewritten starter prompts still on the invite page or in its delivery, waitlist-welcome, or reopen emails, each with section, field, and excerpt).

- **Write category:** read-only
- **Access:** requires the `INVITE_READ` permission

Parameter	Type	Required	Description
`inviteId`	string	yes	The Invite ID

`omumu_invite_launch`

Creates a complete Honest Scarcity Invite in one call: a course (or attaches an existing one), three seeded email sequences (delivery, waitlist-welcome, reopen), an opt-in form wired to the course grant and delivery sequence, the Invite itself, the waitlist tag action on the reopen sequence, and the invite page — all wired, but created as a draft: its page answers visitors with a 404 and its form refuses submissions until you take it live. Returns status 'draft', the page URL, the slot editor path, and placeholders (unrewritten starter prompts still on the page or in its emails, each with section, field, and excerpt) so you can customize copy with omumu_page_update and omumu_email_sequence_update before going live with omumu_invite_update (active: true); going live is refused while any remain. Cadence values: HOUR, DAY, WEEK, MONTH. openingDayOfWeek required for WEEK cadence (e.g. MONDAY). openingDayOfMonth required for MONTH cadence (1-31). openingTime format: HH:MM (24h); defaults to 09:00. zone: IANA timezone id (e.g. 'Europe/Oslo'); defaults to Europe/Oslo. seedLanguage: 'nb' for Norwegian Bokmål, 'en' (or omit) for English.

- **Write category:** standard write
- **Access:** requires the `INVITE_WRITE` permission

Parameter	Type	Required	Description
`cadence`	string	yes	Reset cadence: HOUR, DAY, WEEK, or MONTH
`courseId`	string	no	Existing course ID to attach; omit to create a new empty course
`name`	string	yes	Human-readable name for the Invite (also used as the page headline)
`openingDayOfMonth`	number	no	Day of month for MONTH cadence (1-31)
`openingDayOfWeek`	string	no	Day of week for WEEK cadence (e.g. MONDAY)
`openingTime`	string	no	Opening time in HH:MM (24h); defaults to 09:00
`quota`	number	yes	Slots available per window
`seedLanguage`	string	no	'nb' for Norwegian Bokmål, 'en' or omit for English
`zone`	string	no	IANA timezone id (e.g. 'Europe/Oslo'); defaults to Europe/Oslo

`omumu_invite_list`

Lists all Invites for the current site with their stage (draft, live, or paused), window state (accepted of quota, next boundary), and waitlist size.

- **Write category:** read-only
- **Access:** requires the `INVITE_READ` permission

No parameters.

`omumu_invite_update`

Updates an Invite's name, quota, schedule (cadence, opening moment, zone), or active flag. Changing cadence, openingDayOfWeek/openingDayOfMonth, openingTime, or zone recomputes the next window boundary. Raising or lowering quota takes effect immediately in the current window without

invalidating existing acceptances. active=true takes a draft or paused Invite live (its page and form start working for visitors), but is refused while the invite page or its delivery, waitlist-welcome, or reopen emails still hold unrewritten placeholders — the error lists each one with where to fix it; rewrite with omumu_page_update and omumu_email_sequence_update, then retry. active=false pauses a live Invite (its page 404s for visitors, its form refuses submissions) and lifts any deletion guards on its form and sequences; deactivating is never refused. Cadence values: HOUR, DAY, WEEK, MONTH. Day-of-week values: MONDAY..SUNDAY. openingTime format: HH:MM (24h). zone: IANA timezone id (e.g. 'Europe/Oslo').

- **Write category:** standard write
- **Access:** requires the `INVITE_WRITE` permission

Parameter	Type	Required	Description
`active`	boolean	no	Whether the Invite is active; deactivating lifts deletion guards on its form and sequences
`cadence`	string	no	Reset cadence: HOUR, DAY, WEEK, or MONTH
`inviteId`	string	yes	The Invite ID
`name`	string	no	Human-readable name for the Invite
`openingDayOfMonth`	number	no	Day of month for MONTH cadence (1-31)
`openingDayOfWeek`	string	no	Day of week for WEEK cadence (e.g. MONDAY)
`openingTime`	string	no	Opening time in HH:MM for DAY/WEEK/MONTH cadences
`quota`	number	no	Slots available per window
`zone`	string	no	IANA timezone id (e.g. 'Europe/Oslo')

`omumu_optinform_add_field`

Adds a new field to collect user information

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`fieldLabel`	string	no	Display label (e.g., 'Email Address')
`fieldName`	string	yes	Internal field name (e.g., 'email', 'firstName')
`fieldType`	string	no	Field type: EMAIL, TEXT, PHONE, or HIDDEN
`formId`	string	yes	The form ID
`placeholder`	string	no	Placeholder text for the input
`required`	boolean	no	Whether the field is required
`sortOrder`	integer	no	Position in the form (0 = first)

`omumu_optinform_create`

Creates a new opt-in form for collecting leads. The form comes with default email and name fields. After creating, use omumu_optinform_update to connect it to an email sequence (emailSequenceId) and/or course (courseId). Embed on a page with [optin-form id="FORM_ID"] on its own line (Markdown, Quill or HTML pages). For templated (SLOT_JSON) pages set the page's optinFormId instead. Forms can also be linked to quizzes as pre-quiz or post-quiz email capture.

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Description or subtitle. Supports rich text when descriptionEncoding is QUILL_JSON.
`name`	string	yes	Internal name for the form (admin use)
`title`	string	no	Display title shown to users

`omumu_optinform_delete`

Deletes an opt-in form and all its submissions

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID to delete

`omumu_optinform_get`

Gets an opt-in form by ID with all its fields. Returns descriptionEncoding (PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML) and trustLine if configured.

- **Write category:** read-only
- **Access:** requires the `OPTINFORM_READ` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID

`omumu_optinform_link_sequence`

Connects an opt-in form to an email sequence so submissions trigger subscription

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`emailSequenceId`	string	yes	The email sequence ID to link
`formId`	string	yes	The form ID

`omumu_optinform_list`

Lists all opt-in forms for the current site

- **Write category:** read-only
- **Access:** requires the `OPTINFORM_READ` permission

No parameters.

`omumu_optinform_remove_field`

Removes a field from a form by its field ID

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`fieldId`	string	yes	The field ID to remove
`formId`	string	yes	The form ID

`omumu_optinform_submissions_list`

Lists submissions for an opt-in form, newest first. Each entry contains the submission ID, timestamp, and the full submitted data (all field values). Returns respondent personal data, so it requires the dedicated leads:read scope (not granted by optinform:read).

- **Write category:** read-only
- **Access:** requires the `LEADS_READ` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID
`limit`	number	no	Maximum number of submissions to return (default 50, max 500)

`omumu_optinform_update`

Updates an opt-in form's properties. Use this to connect the form to a course and email sequence, configure the submit button, and set up redirect behavior. Typical funnel setup: create sequence -> create form -> update form with emailSequenceId + courseId + redirectUrl -> embed in page.

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	no	Course to grant access to — creates a user account and enrolls them on form submission
`description`	string	no	Description. Supports rich text when descriptionEncoding is QUILL_JSON.
`descriptionEncoding`	string	no	Encoding for description: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`emailSequenceId`	string	no	Email sequence to subscribe users to — triggers the sequence on form submission
`formId`	string	yes	The form ID
`name`	string	no	Internal name
`redirectUrl`	string	no	URL to redirect after submission
`submitButton`	string	no	Submit button text
`successMessage`	string	no	Message shown after successful submission
`title`	string	no	Display title
`trustLine`	string	no	Small muted text shown below submit button (e.g., 'No spam. Unsubscribe anytime.')

CHAPTER 6

Automations

Most of what a site does for you happens while you are not looking: someone signs up and the welcome sequence starts; a student finishes a module and gets the next one; a hand-in lands and you hear about it. An automation is how you say what should happen when — once — and let the site do it every time after.

On Omumu an automation is a small diagram: a trigger, the steps that follow, and the lines between them. You can draw it in the builder or describe it to your assistant, which writes the same diagram through the automation tools, checks it, and switches it on. This chapter explains the pieces, shows the common automations in both forms, and is plain about the one awkward fact of the current design: some of the wiring on your site lives outside the builder, and the chapter tells you where to look for it. A path such as *Settings → Automations* names a group in the admin's navigation and the page under it.

Concepts

Trigger, steps, connections. Every automation starts with one trigger — the thing that happens — and continues through steps joined by connections. A step either acts (start a sequence, apply a tag, enrol in a course) or decides (*Has Tag* splits the flow into a yes and a no branch). A *Wait* step holds the flow for

a duration before the next step. The diagram has no loops: a contact enters at the trigger and runs to the end.

Triggers you can use. Someone visits a page, submits an opt-in form, buys an offer, gets a tag, watches a given share of a video lesson, completes a module or a course, hands in work on an assignment, has that work rated by the AI or reviewed by your staff, or finishes a quiz with a result (and an email address — anonymous quiz-takers never trigger anything).

Steps you can use. Start or stop an email sequence, apply or remove a tag, grant access to a course, unlock one module of a sequentially gated course, wait, *Notify Site Staff* — an email to one of your site's staff with a digest of what triggered the run — and *Send Webhook*, which posts the run as JSON to an https address you control, signed with a secret if you give one; for a run a quiz started, the post carries the result: profile, total and scale percents, the door and the result link. There is deliberately no "send one email" step: write the email as a one-email sequence and start that. Nothing an automation sends goes to an address outside your staff except through a sequence or a webhook you set up yourself, which is what keeps automations from becoming a mail cannon.

Tags are the glue. A tag is a label on a contact, and it is also the one thing every part of the platform can both set and react to: a form, a quiz result and an automation can all apply one, and *Tag Applied* can start any automation. When two things need to talk to each other, a tag is usually the way.

Two systems, for now. The builder is the automation engine. But some wiring predates it and still runs on its own: the opt-in form editor's *Email Sequence* and *Course Access* settings, and the invite's sequences, are stored as *tag actions* — "when this tag is applied, do these jobs" — and listed on a separate page, also titled Automations (it has no entry in the navigation yet; open `/admin/tag-actions`), by the tag's raw name. Both systems fire. So a form can show "No email sequence" while a builder automation starts one, and the builder knows nothing of the form's own setting. Until the two are joined, check both when something sends twice or not at all. (The engine's own notes call this the two vocabularies; the direction is that new capability lands in the builder only.)

Everything is validated before it runs. Saving a diagram checks every step's settings and every reference — the form, sequence, course, module,

lesson, assignment, quiz or offer it names must exist on your site — and switching it on checks again. Saving an active automation takes effect at once.

Hands-on

When someone signs up, start the sequence

Mira has a "Time audit checklist" form and a welcome sequence. She says: *"When someone submits the time-audit form, start the Welcome sequence and tell me."*

The assistant creates the automation, draws *Form Submitted (Time audit checklist) → Start Sequence (Welcome to Automate Your Week) → Notify Site Staff (Mira)*, saves it, runs a simulation of one sign-up to see the timeline, and switches it on.

Underneath, the authoring loop is five tools. `omumu_automation_options` lists every sequence, course and module, tag, form, offer, quiz and staff recipient on the site, with the settings each step type takes; `omumu_automation_create` makes an inactive automation; `omumu_automation_generate_ids` hands out ids for steps and connections; `omumu_automation_save` writes the whole diagram (steps with their settings, and the connections between them) and returns any warnings; `omumu_automation_simulate` plays a scenario — "this contact submits the form at t0" — against all of the site's automations in memory and returns what would happen and when, without sending anything; `omumu_automation_activate` switches it on or off. `omumu_automation_get` and `omumu_automation_list` read what exists. There is no delete tool; delete in the admin.

In the admin: *Settings → Automations → Create New Automation* asks for a name and opens the builder: a full-screen canvas with a palette of triggers, contact actions and utilities on the left. Click or drag a step onto the canvas, join steps by dragging between their connection points, select a step to fill in its settings, *Save*, then *Activate*. The builder has its own *Back* link; it does not show the admin header or navigation.

Two honesty notes for exactly this automation. The form editor has its own *Email Sequence* setting that does the same thing through a tag action — pick

one path, not both, or the contact is subscribed twice. And every opt-in submission already emails the site's support address, whether or not you add *Notify Site Staff*; there is no switch for that email yet.

Celebrate a finished module

Mira says: *"When someone completes Week 1, tag them 'week-1-done' and start the Week 2 teaser sequence."*

Module Completed (Automate Your Week, Week 1) → *Apply Tag* → *Start Sequence*. The same shape works for *Course Completed*, for *Video Watched* at a chosen percentage of a lesson, and for *Purchase Made* on an offer. On a sequentially gated course the next module opens by itself when the previous one is complete; *Unlock Module* exists for the case where you want to open a module for another reason — a hand-in reviewed, a tag earned.

Get told when something happens

Mira says: *"Email me when a student hands in the Week 1 assignment, and again if the AI rates it below 5."*

Submission Received (Your time audit) → *Notify Site Staff*. The email says what triggered the run. *Submission Rated* and *Submission Reviewed* trigger the same way; the recipient must be a Headmaster or staff member on the site, and the address is checked again at send time. The Teaching & Feedback chapter covers the built-in hand-in emails that need no automation.

Branch on a tag, wait before you act

Mira says: *"Two days after someone gets the checklist, if they haven't bought, start the offer sequence."*

Form Submitted → *Wait (2 days)* → *Has Tag (customer)*, with the *no* branch going to *Start Sequence*. "Customer" here is a tag the purchase automation applies; that is the tag-as-glue pattern. Wait takes a duration; the simulation shows you whether a purchase during the wait changes the outcome, because all of the site's automations run together in it.

Try it before it runs

In the builder, *Test Run* takes the email of an existing contact and a trigger and walks the diagram step by step, saying what would execute without doing it. Over the connection, `omumu_automation_simulate` does the same for several contacts and several events on one timeline. Once live, the *Executions* page under each automation lists every run with its contact, status and timing, and expands to the steps.

What you cannot do yet

Stated plainly: there is no send-one-email step, no branching on email opens or clicks, no scheduled or manual trigger and no inbound-email trigger (those belong to a workflow family not offered to creators), and a webhook can only post to a public https address, never to something on your own network. Your assistant cannot delete an automation; do that in the admin. The tag-actions page lists wiring by raw tag names, and the form editor does not show which builder automations its submissions start. And the built-in email to the site's support address on every form submission cannot be switched off.

Reference

`omumu_automation_activate`

Activates or deactivates an automation. Active automations will trigger when their conditions are met. Before activating, run automation_simulate for each contact journey that matters and resolve any timing-trap warnings from the save/activate response — activation is the last step of the authoring loop (save, check warnings, simulate, activate).

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`active`	boolean	no	true to activate, false to deactivate (default: true)
`automationId`	string	yes	The automation ID

`omumu_automation_create`

Creates a new automation (inactive by default). Recommended authoring loop: 1) automation_save the nodes and connections, 2) read any timing-trap warnings in the save response, 3) automation_simulate the contact journeys that matter (especially 'contact does X, then Y during a delay'), 4) only then automation_activate.

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Description of what this automation does
`name`	string	yes	The automation name

`omumu_automation_generate_ids`

Generates unique IDs for use when creating new nodes and connections in canvas state

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`count`	integer	no	Number of IDs to generate (default: 5, max: 50)

`omumu_automation_get`

Gets an automation with its full canvas state (nodes, connections, configs). Use this to inspect or modify an existing automation.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID

`omumu_automation_list`

Lists all automations for the current site with their status

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

No parameters.

`omumu_automation_options`

Returns all available sequences, courses (with lessons), tags, forms, offers, quizzes, staff recipients, and supported node types. Use this to find valid IDs when building automation configs. Each supported node type carries a settings array describing its config keys: key, required, type, allowed values (when enumerated), and refersTo (which list above an id-valued key must be found in, e.g. “sequences” or “courses[].modules”). quizzes lists the site’s quizzes (id + name) for the quiz_completed trigger. staffRecipients lists who may receive notify_site_staff emails (name + email); defaultRecipient is the authoring user’s email. send_webhook’s settings document its url (https, public address, re-checked at send time) and optional secret (signs the body with X-Omumu-Signature: sha256=); see omumu_automation_save’s description for the posted JSON shape. There is no send_email node: to send a single email to a contact, create an email sequence containing one message with zero initial delay and start it with the start_sequence action.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

No parameters.

`omumu_automation_save`

Saves the full canvas state (nodes + connections) for an automation. Saving takes effect immediately, including on an already-active automation — there is no need to deactivate first. Canvas state format: {“nodes”: [{“id”: “…”, “type”: “tag_applied”, “x”: 0, “y”: 0, “config”: {…}}], “connections”: [{“id”: “…”, “from”: “nodeId”, “to”: “nodeId”, “type”: “default”}]}. Use automation_generate_ids to get IDs for new nodes/connections. Use automation_options to get valid IDs for sequences, courses, tags, forms, offers, quizzes, staff recipients — and, for every node type, its settings (config keys, required or optional, allowed values, and what catalog list an id refers to). Completion triggers (module_completed, course_completed) are side-effects only (celebrate, tag, follow up) — on a

SEQUENTIAL course the next module unlocks by itself, so do not wire module_completed to unlock_module for gating. Pair submission_received with unlock_module to open the next module when a student posts their work. Pair quiz_completed (or form_submit) with notify_site_staff to email a staff member a digest of every answer, the bucket, and the score. send_webhook posts the run data as JSON to a creator-controlled URL ({“event”, “siteId”, “occurredAt”, “runId”, “contact”: {“userId”,“email”,“firstName”,“lastName”,“phone”} when identified, “input”: the trigger’s run input, and for a quiz-triggered run “quiz”: {“quizId”,“label”,“bucketName”,“totalPercent”,“scalePercents”,“doorName”,“resultUrl”,“answers”:[{“question”,“answers”|“text”}]}}); config url must be https and resolve to a public address (checked again at send time), config secret is optional and signs the body as header X-Omumu-Signature: sha256=; a non-2xx response, a timeout or a refused connection is recorded as a FAILED step and the automation continues. There is no send_email node: to send a single email to a contact, create an email sequence containing one message with zero initial delay (omumu_email_sequence_create, initialDelay PT0S) and start it with the start_sequence action. The response includes timing-trap warnings (e.g. stop-sequence-before-delayed-start: a STOP_SEQUENCE may fire before a WAIT-delayed START_SEQUENCE and silently no-op; fix by starting the sequence immediately and moving the delay into the sequence’s initialDelay). Warnings are non-blocking, but do not activate past one without first proving the funnel correct with automation_simulate.

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID
`canvasState`	object	yes	The full canvas state JSON with nodes and connections

`omumu_automation_simulate`

Runs a scenario against the site’s automations in memory (no emails sent, nothing written) and returns the effect timeline plus any timing-trap warnings. All of the site’s automations run together so cross-flow races (e.g. a purchase stopping a delayed abandonment sequence) are visible. ‘events’ is an array of {“at”: duration, “trigger”: triggerType, “contact”: alias}: ‘at’ is a dura-

tion from t0 as ISO-8601 (PT5M) or a friendly form (0s, 5m, 1h, 2d, 1w); 'trigger' is a contact trigger type (form_submit, purchase, tag_applied, page_visit); 'contact' is an optional alias (default 'contact') so several events can describe one contact's journey. Each timeline entry is {atSeconds, at, contact, kind, target, effective}: kind is APPLY_TAG, REMOVE_TAG, START_SEQUENCE, STOP_SEQUENCE, EMAIL_SENT, ENROLL_COURSE, UNLOCK_MODULE, GRANT_ROLE or NODE_REJECTED; 'effective' is false for a no-op (e.g. a stop with no live subscription). Use this to verify a funnel behaves before activating it: don't just look for the expected effects — scan the timeline for effective=false on stops/removals and for EMAIL_SENT entries the contact should NOT receive (an ineffective STOP_SEQUENCE followed by a later EMAIL_SENT is the classic cart-abandonment race). Simulating is step 3 of the authoring loop: save, check warnings, simulate the journeys that matter, then activate.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID to simulate (its site's automations all run together)
`events`	array	yes	Ordered scenario events: [{"at": "5m", "trigger": "purchase", "contact": "buyer"}]

CHAPTER 7

Teaching & Feedback

A course is content; teaching is what happens when the work comes back. This chapter is about that loop: the assignment you attach to a lesson, the hand-in a student sends, the feedback the AI writes within seconds from your own material, the Light the student raises when they are stuck, and the board that tells you who to call. It is the most human chapter in the manual, and the one with the fewest tools: your assistant can set assignments up, but reading a hand-in and answering it is yours.

A path such as *Customers → Mentor Board* names a group in the admin's navigation and the page under it.

Concepts

An assignment asks for work. It is attached to exactly one scope — a lesson, a module, the whole course, or the site itself — and that placement decides where the student meets it and what it counts for: a lesson's assignment sits on the lesson page and is part of finishing that lesson; a module's gets its own page after the module's lessons; a course's is listed on the course home and has its own page; the site's, *All courses* in the admin, has a page of its own outside every course. The assignment carries *rating criteria*: one text that both the AI and your staff read when they judge a hand-in. It also decides whether

every hand-in must carry a Light and whether staff are emailed on each hand-in. New assignments are drafts until you publish them.

A hand-in is text, a file, or both. PDF, image, audio, video, or a plain-text file, up to 500 MB. A student can hand in more than once; each submission is rated on its own. Hand-ins are private to the student and your staff.

Assignments is the student's list of everything asked of them. It shows, for any student who can open at least one course on your site, the published All courses assignments first in the order you set, then one group per course with its course, module and lesson assignments in course order. Each row has the assignment's title, the student's status on it and the date of their latest hand-in, and opens where the work happens: the lesson page, or the assignment's own page. One the student cannot open yet stays on the list with the reason, such as the module that opens it, and no link. The list never jumps straight to an assignment, even when there is only one. An All courses assignment's page has its hand-in form and the student's earlier hand-ins underneath. It is where a participant writes to you about everything they have rather than one lesson — a morning page, an evening reflection — and comes back to see how it is answered. An All courses assignment counts toward no course's completion; it is a place to return to, not a gate.

The rater answers first, from your teaching. When a hand-in lands, the AI rates it at once against the criteria. It reads what the student wrote and attached — text, PDFs and images directly, audio and video through their transcripts — and it reads *your* material in the assignment's scope: lesson descriptions, text blocks, checklist items, uploaded files and the transcripts of your recordings. For an All courses assignment the scope is every course that student can open, so what the rater reads differs from student to student. It also reads the student's participant folder (the next chapter), so feedback remembers them. It never reads your mentor notes and never quotes them. The student sees the AI's feedback immediately, before you have; that is deliberate, so nobody waits on you for a first answer.

Check-ins and Lights. An assignment that *asks for a check-in light* turns into a recurring self-report: with every hand-in the student says how it is going — green (clear and moving), yellow (slowed or unsure), red (stuck, needs help). The Light is the student's own word, never derived from their text or progress; the AI acknowledges it and never grades it. A weekly check-in is a habit your program keeps, not a schedule the platform enforces. When you have read a

Light you *confirm* it — one click that records who saw it and when — and an unconfirmed red is the thing the board exists to show you.

Finishing a lesson includes handing in. Completion is derived from content: a lesson with an assignment is complete when the student has opened it, ticked its checklist, watched its videos, and handed in. On a sequentially gated course that is what opens the next module. Adding an assignment later adds a requirement for students not yet done; it never un-completes anyone.

Hands-on

Ask for work

Mira says: *"Add an assignment to the Week 1 lesson: hand in your time audit. Rate it on whether they listed at least five repeated tasks with a time each and picked one to automate. Ask for a check-in light. Publish it."*

`omumu_assignment_create` attaches it to the lesson with the description and criteria, `omumu_assignment_update` publishes it (the Courses & Content chapter has the tools). In the admin, *Products → Assignments* lists them, and the course overview's *Assignments* card opens that list for the course; *New Assignment* asks first where it is attached, then takes the title, description, rating prompt, the check-in light switch and *Email staff when a student hands in* (on by default), and lands on the list as a draft; *Edit* has the *Status* switch that publishes.

Below the form, *What the rater can read* lists every item in the assignment's scope — lesson descriptions, text, recordings, files, checklist items, links — as readable or not, with the reason: a recording with no transcript, a scanned PDF, a file linked from outside Omumu. Recordings can be transcribed from there; the page says how many of your plan's minutes it will use before it starts. If the rater's feedback misses your method, this list is where to look. An All courses assignment's list is grouped by course and covers every course on the site, with a note that each student is rated against the ones they can open; such assignments live in the *All courses* group of *Products → Assignments*.

What the student does

On the lesson page, under the content and the checklist, the student sees the assignment's brief, the Light choice if asked, a text box and a file field; *Submit*. The file uploads at once and shows its name. After submitting, the AI's feedback appears on the submission page within seconds, with every earlier submission underneath. A student with a Light can hand in again each week and watch the thread grow. All courses assignments work the same way from *Assignments* in the site header, which every student with at least one assignment sees without you doing anything: the student opens one from the list and finds its own form and its own thread of earlier hand-ins on its page. A hand-in the form refuses, an empty one say, comes back to that same page with the reason shown.

What the AI does with a hand-in

Say Mira's rating prompt asks for encouragement and a concrete next step. A student hands in a yellow check-in with a list of five tasks and a note that she is unsure which to automate. The feedback names the tasks, points at the easiest first win, acknowledges the yellow without arguing with it, and suggests a next step. A file-only hand-in of a PDF or a screenshot is read like text; a video is read through its transcript. The rater writes sourced facts into the student's folder as it goes, so the next feedback starts from what it already knows.

Read, confirm, write

Mira opens the hand-in and reads the AI's feedback above the form. She writes her own — *Feedback* and an optional score from 1 to 10 — and *Save feedback*; her words show above the AI's. If the hand-in carries a Light, *Confirm I have seen this Light* records that a human has read it. Two separate actions today: writing feedback does not confirm the Light by itself.

Under the AI's feedback, Mira also sees *What the rater read*: which lessons it opened, what it searched your material for and which passages came back (each search unfolds to show them), whether it hit its reading cap, where your criteria and your material disagree, whether the passages it quoted were found word for word in the hand-in or your teaching (and whether it had to be asked to fix one that wasn't), and what the rating used: the model, its turns and tokens, with the price when it is known. For an All courses assignment it also names the courses that student was rated against. *What the rater can read* on

the assignment shows what was available; this shows what the rater actually used for this hand-in, so it's the first place to look when feedback misses your method. It is for staff only. The student sees the feedback and the score, never the trace or its cost, and an agent reading the hand-in through Omumu's tools doesn't get it either.

Getting to the hand-in: *Customers* → *Users*, the student, *Submissions*, and the hand-in you want — each marked seen or not seen. There is no page yet that lists hand-ins waiting for feedback across students; the staff email on each hand-in is the queue in the meantime.

Run the board

Customers → *Mentor Board* shows every participant with their latest Light and its date, whether it is seen, days since their last hand-in, and progress, filtered by course. An unconfirmed red at the top of the list is who you call this week; a long gap since the last hand-in is who has gone quiet. *Mark seen* confirms a Light from the board.

Who hears what

Staff are emailed on every hand-in, with yellow and red called out in the subject — a per-assignment switch, on by default. The student is emailed when a staff member writes feedback (not for the AI's), a site-wide setting under *Settings* → *Business Details* → *Notifications* that is on by default and can be turned off. For anything else — a tag when work comes in, a sequence after a review — the Submission Received, Rated and Reviewed triggers in the Automations chapter do it.

What you cannot do yet

Stated plainly: there are no tools for hand-ins — your assistant can create and publish assignments but cannot read a submission, write feedback or confirm a Light. There is no review queue across students; no due dates; no quizzes inside a course (quizzes live under Sales and marketing and are for finding out who someone is, not for checking what they learned). *Delete* on the people list disables the account rather than erasing the person's data; permanent erasure

is coming. And the Mentor Board counts you, the Headmaster, as a participant of your own courses.

Reference

No MCP tools cover this chapter's capabilities yet.

CHAPTER 8

The Participant Folder

A mentored programme works because the participant feels known. The mentor remembers what they set out to do, notices when the same thing happens again, and asks about the promise they made last week. AI feedback on its own is the opposite: every hand-in is rated as if it were the first.

The participant folder fixes that. Each participant gets a folder of short, sourced facts about them: their goals, the patterns in their work, their commitments, their context, and your own notes. The AI reads the folder before it rates a hand-in and updates it afterwards. You and your staff curate it on the page you already use for that student, and the participant can read all of it except your own mentor notes. By week nine the feedback can say "you held the silence you couldn't hold in week one", because the folder remembers week one.

Concepts

A **participant folder** is everything the programme knows about one student on one site: their live **folder facts**, plus their raw history of hand-ins, AI ratings, transcripts and material. A folder belongs to one site and one participant. It is never shared with another site. It is meant to be erased with the participant's account; today *Delete* on the people list disables the account, and permanent erasure is coming.

A **folder fact** is one durable statement you can trace back to its source. It has a type, a short name, a one-line description and a body. It also records its **provenance**: the hand-in, material or staff member it came from. There are five types:

- **Goal**: what the participant is trying to achieve.
- **Pattern**: something that keeps happening in their work.
- **Commitment**: something they said they would do.
- **Context**: their situation, such as their job, tools and constraints.
- **Mentor note**: something you or your staff want the programme to keep in mind. Only staff write mentor notes; the AI never does. They are the mentor's own: read by staff only, never shown to the participant, and never quoted by the AI — though the AI reads them before it speaks, so they shape its feedback.

Facts are never deleted. **Editing** a fact writes a new version that replaces the old one. **Retiring** a fact takes it out of the live folder but keeps it. Both old versions and retired facts stay in a collapsed "Retired facts" section, which shows what replaced each one, who replaced or retired it, and when.

Folder material is anything you add to a folder on the participant's behalf: notes from a mentor call, a question they asked in a webinar, a work conversation, their intake answers. You give it a label and either paste text or upload an audio or video recording. A recording is first transcribed from your site's transcription minutes. The AI then reads the material once and proposes facts from it; material is never scored. The difference from a hand-in is who adds it: the participant hands in their own work, and staff add material.

The **rater** is the AI that rates hand-ins for an assignment with a rating prompt. With the folder, it works like this:

1. **Before rating**, it receives the participant's live facts and their three most recent items of history, next to your rating prompt and the hand-in itself. Facts are always included. If the history is too long, the oldest items are dropped until it fits. It can also look up your own teaching in the assignment's scope — lesson text, files and transcripts — and the hand-in's page keeps a staff-only trace of what it read (the Teaching & Feedback chapter).
2. **After rating**, its answer has two parts: the feedback the participant sees, as before, and a list of changes to the folder (add, update or retire a fact). Each change points back to the hand-in it came from.

Changes apply at once, with no approval queue. You curate afterwards rather than approving in advance. If a change can't be understood, it is skipped and logged, and the participant still gets their feedback. Rating a hand-in is still one AI request.

Nothing in a folder is hidden from its participant, except mentor notes. They can read every other live fact — including every fact the AI writes — and where it came from on their own page, which students see as *Growth* (*Utvikling* in Norwegian), but they can't change anything there. Disputes come to you. A mentor note is private in the product, not secret: it is still the participant's personal data, so write nothing you could not stand behind if they read it.

Hands-on

The walkthroughs follow Mira. She has turned her flagship course into a mentored cohort, "Automate Your Week — Coaching Cohort", where participants hand in a short reflection every week. One of them is Ingrid, an office manager at a small accounting firm. Ingrid wants her Friday client-hours report to take thirty minutes instead of three hours.

Mira normally runs her site by talking to her AI assistant, but the folder has **no MCP tools yet**. Curating a folder, adding material and reading a folder all happen in the admin UI, and the walkthroughs below use it. The AI's part runs on its own: the rater and the material reader work in the background whichever way the hand-in or material arrived.

Give a programme a memory

The folder starts working for any assignment that has a **rating prompt**, because that is the assignment the rater runs on. Mira's cohort has one assignment, "Weekly reflection". She sets it up under **Courses → the course → Assignments → New assignment** with this rating prompt:

> You are the coach in "Automate Your Week", a mentored programme where non-programmers learn to automate their weekly work with spreadsheets and AI tools. The participant hands in a weekly reflection. Give warm, specific, honest feedback in at most 180 words. Name one

> thing that went well and one concrete thing to try next week. Build on what you know about the participant from earlier weeks, and hold them to what they said they would do.

Your prompt doesn't need to explain the folder. The platform adds the folder to the request and asks for the list of changes itself. Write the prompt about the coaching you want, and ask it to build on what it knows. Then publish the assignment.

Start a folder from the intake

Before the first hand-in, Mira gives each participant a starting point. She opens **Students → Ingrid Berg**, scrolls to **Folder** and uses **Add a fact** for what Ingrid's intake form told her:

- a **Goal**, "Weekly report in under 30 minutes", with the details of the Friday routine in the body;
- a **Context** fact, "Office manager at a 12-person accounting firm", recording that she uses Google Sheets daily and has no programming background.

Facts Mira types show "Added by Mira Lund" and read "From: your mentor" on Ingrid's page. Adding, editing and retiring facts needs the **student-management** privilege, the same one that reviewing submissions needs.

Add what you learned on a call

After the kick-off call, Mira doesn't turn her notes into facts by hand. Under **Material → Add material** she picks the label **Mentor conversation**, pastes her notes and clicks **Add material**. The material shows "Waiting to be read" for a few seconds, then "Read". From her notes the AI added three facts, each marked "Added by the rater" and showing "From: Mentor conversation" as its source:

- **Context**: the hours come from a Harvest CSV export that Ingrid retypes because the columns are in the wrong order;
- **Pattern**: "Tends to start too big". She wanted to automate invoicing, the report and onboarding all in week one;
- **Commitment**: "Week 1: Build Harvest-importing Google Sheet", working on a copy of the partners' template.

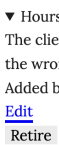

Figure 1: A fact the AI read from Mira's mentor-call notes, with "From: Mentor conversation" as its source

For a recording, upload it under **Upload a recording** instead of pasting text. It is transcribed first ("Being transcribed", then "Transcribed") and read after that. The minutes come from your site's transcription pool. If the recording can't be transcribed, the material says why: the month's minutes are used up, the recording is too long, or transcription isn't part of your plan. Once the reason is fixed, **Retry transcription** runs it again.

Watch a hand-in use the folder

Ingrid hands in her week 1 reflection. The sheet works: Friday now takes 70 minutes instead of three hours. She also admits she lost Wednesday afternoon building an invoice tab "because the data was right there". The rater reads her folder before it answers, and the feedback shows it:

> Ingrid, this is real progress: INDEX/MATCH auto-reordering the columns turned three hours into 70 minutes, and you did it on a safe copy just as planned … On the invoicing detour: you caught yourself and named it honestly … a reminder of why we scoped down to one project. Before adding a client summary next week, I'd like you to first tackle the client-name mismatch … That's the last mile of this goal.

It then updated the folder. It recorded the week 1 commitment as done, updated the "Tends to start too big" pattern with the invoicing detour, and added a new commitment for week 2. In the admin panel each of those facts has a **Source** link back to the hand-in.

Keep the folder right

The AI's changes apply without your approval, so curating afterwards is your job. Mira does three things on Ingrid's folder:

- **Edit.** A fact can be nearly right, or worded in a way Mira wouldn't put it to Ingrid, who reads her whole folder. Mira opens **Edit**, rewrites the body and saves. The edit creates a new version, and the old one moves to **Retired facts** marked "Replaced by Tends to start too big · edited by Mira Lund" with the date and time. When the rater updates a fact after a hand-in or a material read, the old version reads "updated by the rater" instead, so her own edits are easy to tell from the AI's.
- **Retire.** The week 1 commitment is done, so she clicks **Retire**. It leaves the live folder at once and stops shaping feedback. It stays under **Retired facts** as "Retired by Mira Lund" with the date and time.
- **Add a mentor note.** She adds a **Mentor note**, "Motivated by getting her Friday back", so the next rating frames Ingrid's next step in minutes saved.

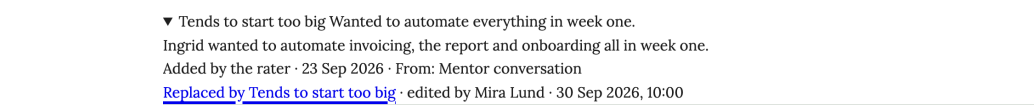

Figure 2: The rater's version of "Tends to start too big" under Retired facts, replaced by Mira's edit

What your participant sees

Ingrid finds **Growth** in the site header, after *Courses* and *Assignments*; a Norwegian site calls it **Utvikling**. Her old bookmark to the folder still lands there. The page shows her live facts grouped as Goals, Patterns, Commitments and Context — never Mentor notes. Each fact shows when it was added and where it came from ("From: your mentor", "From: Mentor conversation", or a link to the hand-in). Her history is listed below: her hand-ins, linked, and the material Mira added, with its text behind **Show text**. The page has no buttons and no forms. If Ingrid disagrees with something, she takes it up with Mira.

Things to know

- **Every fact except a mentor note is visible to the participant**, including those the AI writes. The AI is told this and writes plain statements of what was said, done or agreed, but it is still worth reading its facts and editing any whose wording you wouldn't use yourself.
- **Mentor notes are yours.** Only staff write them, only staff see them, and the rater never quotes one.

- **The folder is not shared across sites.** A participant who takes a programme on another Omumu site starts with an empty folder there.
- **The cost is small and visible.** Each rating or material read is one AI request. Its input and output tokens are recorded in the outcome log with the result.

Reference

No MCP tools cover this chapter's capabilities yet.

CHAPTER 9

Podcasting

A podcast is the cheapest way to be in someone's ears every week, and the people who listen to yours are the people who buy your courses. On Omumu the podcast lives on your own site: one show with a public feed that Apple Podcasts, Spotify and every other app read, episodes you upload as plain audio files and the platform masters, transcribes and serves, and an editor that lets you cut an "um" by selecting the word.

This chapter is the show from nothing to listed. Your assistant can publish episodes for you; the show itself, and the editing, are admin work, and the chapter says which is which. A path such as *Products → Podcast* names a group in the admin's navigation and the page under it.

Concepts

One show, one feed. A site has one podcast show. The show carries what directories need — title, description, language, a category from Apple's list, author and owner, an explicit flag, a show type — and a *slug* that becomes the public address: the show page, each episode's page, and the feed at `/podcast/<slug>/feed.rss`. You submit that feed address once per directory; after that, new episodes reach every app on their own.

Podcast cover art size. Directories require square cover art, at least 1400 and at most 3000 pixels on a side. You can pick an uploaded image, or let the show

page draft covers for you: a template, a short prompt for the background, and the show's title and author composited on top at 3000 × 3000. Publishing an episode while the show has no cover warns you, because the feed would be refused by the directories.

An episode is a file you upload. MP3, M4A (an iPhone voice memo) or WAV. On upload the platform *masters* it — brings it to podcast loudness — and the episode sits in *processing* until that is done, then *ready*. Omumu's own chain does the mastering unless the site has an *Auphonic connection*: then your own Auphonic account masters every new upload (adaptive leveling, automatic EQ, bandwidth extension) on your own Auphonic credits, and the episode page says which of the two mastered it. If Auphonic cannot master a new upload — credits used up, key revoked, Auphonic unreachable — Omumu's chain masters it instead, the episode says why, and its page warns before you publish. *Noise reduction* is a per-episode choice, off by default, for a recording made outdoors or in a noisy room; you make it at upload, when you master the episode for the first time, or when you re-master it. Publishing stamps the publication date, locks the audio file (subscribers' apps may already have fetched it) and puts the episode in the feed. You don't have to wait for mastering to publish: publishing an episode that is still *processing* records that it should go live, and it publishes itself the moment mastering ends, dated then — also when Omumu's chain had to master it instead of Auphonic, and then you get an email saying why, with a link to the episode. A failed mastering publishes nothing, keeps that request for the retry and emails you the reason with a link to retry it, and until the episode is live you can cancel it. A publish that goes as planned sends no email. Mastering can also wait: *Upload without mastering* leaves the episode *unmastered*, so no mastering is spent on a recording you will trim first. It is mastered when you click *Master*, on its page or in the editor, or when you publish it (mastered first, then live). Mastering is deferred, never skipped: nothing reaches the feed unmastered. A failed mastering can be retried, and any episode can be *re-mastered*: the original upload is mastered again with the current chain, and a published episode keeps playing its current audio until the new master is ready. Episodes have a type (full, trailer, bonus), a season and number, an explicit flag that can inherit the show's, and optional artwork of their own.

Transcripts come with the episode. Every episode is transcribed by speech-to-text — at publish, or earlier if you choose *Transcribe episode* in the editor and accept the cost in plan minutes. It can be transcribed before it is mastered. The

transcript ships in the feed in the formats podcast apps read and is shown on the episode's public page, with speaker labels. One transcription serves every round of editing, and a re-master keeps it: the new master is lined up with the audio the transcript was made from, and the word timings move with it. Only where the two cannot be lined up is the episode transcribed again, on plan minutes.

Editing is editing the words. The episode editor opens on any episode that has audio, before it is mastered and before it is transcribed: the waveform and its cuts work at once, and the words wait for a transcript. It plays the audio with the transcript highlighted word by word, under a waveform of the whole recording, the audio no word covers included, with the playback position and every cut shown in both. Select a passage and delete it, or drag across the waveform to remove audio no word covers, such as music before the first word or a jingle after the last (a cut on the waveform removes exactly the audio you marked, and a word leaves the transcript only when the cut covers all of it); tighten long pauses, one or all at once; correct a misheard word (text only; the audio is untouched). Edits are saved as you go, and publishing renders them: the episode goes live with the edited audio as soon as it is ready. On an episode that is already published the editor says what your edits would change for listeners, and *Update published episode* puts them live. The original recording is kept, so a cut can be undone in a later session and renders never stack. Editing is a plan feature of its own.

One recording, several episodes. A long conversation can be split into several episodes at points you choose. Each part is a draft of its own with its slice of the master and of the transcript, so nothing is mastered or transcribed again, and the cuts you already made carry into the parts. The original stays as a locked source that links to its parts, so the same audio never reaches the feed twice. Only an unpublished episode that is mastered and transcribed can be split, and the split can be undone until a part is published.

Hands-on

Create the show

Products → Podcast starts with the show. Mira fills in the title, a description, the language, a category (*Education → How To*), her name as author and owner, and a slug. Then the cover: she types a background prompt, picks a template, generates a draft, and accepts the one she likes — it becomes a 3000 × 3000 cover with the title and author set on it. The page shows the feed address and links to the distribution checklist and analytics.

There are no agent tools for creating or editing the show; `omumu_podcast_show_get` reads it — title, slug, description, language and feed address — once it exists.

Publish an episode

Mira records a 20-minute episode on her phone and says: *"Publish this as episode 2 of my podcast, 'Automate one task', with these show notes."*

The assistant uploads the file straight to storage with `omumu_resource_upload_presign`, creates the episode with `omumu_podcast_episode_create` (the audio resource, title, show notes, season and number, type, explicit flag) and publishes it at once with `omumu_podcast_episode_publish`. The episode is still *processing*, so the call records that it should go live, and `omumu_podcast_episode_get` reports `publishIntent` until mastering ends and the episode publishes itself; `omumu_podcast_episode_publish_cancel` takes it back until then. `omumu_podcast_episode_list` shows every episode and its status; `omumu_podcast_episode_retry` re-runs a failed mastering. For a recording she will trim first, `deferMastering` on `omumu_podcast_episode_create` leaves the episode *unmastered*; `omumu_podcast_episode_master` starts its mastering when she is ready, with noise reduction if she asks for it, and publishing it masters it first.

In the admin: the show page's *New episode* takes the audio file, title, show notes, type, season and number, rating and artwork, and *Upload without mastering* holds the mastering back; *Create episode* lands you back on the show with the episode in its list, where an unmastered one has *Master* beside it. The episode's own page shows its status, lets you edit the details, generate episode artwork (the show's background with this episode's title, no AI involved), see the transcript's state, and *Publish* in a dialog that reminds you the date is stamped and the audio locked. An episode your assistant asked to publish while it was still being mastered says so — *This episode will be published as soon*

as it is mastered — with *Cancel publishing* beside it. An unmastered episode's page offers *Master* and *Publish*, which masters it first and then publishes it.

Clean up the audio

Mira listens back and hears three "um"s and a long pause. On the episode page she opens the *Episode editor*. A strip at the top says where the episode stands: mastered or not, transcribed or not, published or not. If the episode is not transcribed yet, the audio and the waveform are there to cut already, and *Transcribe episode* says how many plan minutes the transcript will cost and asks before it starts. If it is not mastered yet either, *Master it* masters it from the editor, with the cost first: Auphonic credits for each minute of the recording when Auphonic masters for the site, included otherwise. She can do the two in either order, or cut first. The first time she opens it, the waveform above the transcript says it is being drawn, and shows the whole recording a moment later. Then she clicks a word to listen from there, or clicks the waveform to jump to that moment (the arrow keys do the same on it), and *Zoom in* until a tenth of a second is easy to see. The controls sit in one row above the waveform, each with a key: *Play*, *Split*, *Delete*, *Undo* and *Redo*. She drags across the words of a passage (or shift-clicks its first and last word) and presses *Delete*, uses *Tighten all pauses* with a threshold in seconds, and each cut and tightened pause shows hatched on the waveform at once. Her Auphonic free plan put a jingle after her last word, so she drags across the waveform from the end of that word past the end of the recording and presses *Delete*: the jingle is hatched, listed under the waveform as an audio cut with its own *Remove*, and the duration drops to where she stopped talking. The cut is exactly the stretch she dragged across: a word is struck in the transcript only when the cut covers all of it. On a phone, or without a mouse, *Split* at one moment and *Split* at another marks off the stretch between them; selecting that segment and *Delete* does the same. She double-clicks a misheard word to correct it. There is nothing to apply: the edits are saved as she makes them, and when she publishes, the episode is rendered and goes live with the edited audio and the corrected transcript. Had it been published already, *Update published episode* would put the edits live. The original stays behind the scenes for the next round.

Editing itself has no agent tools.

Mira's early episodes were recorded in a café and sound thin next to the new ones. Under *Integrations → Auphonic* she pastes the API key from her Auphonic account; Omumu checks it on save, and from then on every new upload is mastered by Auphonic on her credits. The page also says that Auphonic's free plan adds a short jingle to the start and end of every episode it masters, and that paid Auphonic credits remove it. Omumu cannot see which plan she is on, so everyone gets the same notice, and it never removes the jingle itself: an episode published before mastering ends goes live with it, and one still unpublished she can cut on the waveform, as above. For the old episodes she opens one and clicks *Re-master*: the original upload is mastered again with the current sound, listeners hear the current audio until the new master lands, and then the feed switches over, with the transcript and her edits still in place. For the whole back catalogue the show page's *Re-master all* lists how many episodes it covers, which ones it leaves out and why (published with edits, which a new master would undo, a re-master already running, already mastered with the current sound, not mastered yet), how many Auphonic minutes it will use, and how many transcription minutes it will take from the plan's pool for the episodes whose transcript cannot be carried over to the new master. Re-masters run one at a time. The café episodes also get *Remove background noise* switched on, which is an episode's own setting, chosen at upload or when re-mastering; it helps a noisy recording and can make a clean one sound harsh.

Her assistant can do the re-mastering: *"Re-master episode 2 with noise reduction"* is `omumu_podcast_episode_remaster`, and `omumu_podcast_episode_get` and `_list` report the re-master's state, what mastered the current audio, why Auphonic did not, and whether noise reduction is on. Connecting Auphonic is admin-only, never over MCP.

Split a long conversation

Mira recorded a 50-minute conversation with two guests, and it is really three episodes. In the editor, under the waveform, *Split into episodes* takes the points to split at. She clicks the word where the second topic starts and presses *Add the playback position*, does the same where the third starts, and the field reads `16:42.3, 33:10.8`; she could type the times as well. The audio is split exactly at each point, and a word a point falls inside goes to the part that holds more of it. *Split* cuts the parts while the editor follows along, then lists them: *(1/3)*, *(2/3)* and *(3/3)* after the episode's title, each a draft with its slice of the master

and of the transcript, the show notes and the artwork. Nothing is mastered or transcribed again, so the split uses no Auphonic credits and no plan minutes. The cuts she had already made carry into the parts, a cut across a split point becoming one cut in each, and she edits and publishes each part on its own, days apart. The original stays as a locked source: it cannot be published, edited or re-mastered, the episode list leaves it out, and its page links to its parts. A part cannot be re-mastered either, because its timings belong to the original's master.

If a cut point turns out wrong, the original's page offers *Undo split* until one of the parts is published. Undoing deletes the parts, and any edits made in them are lost, which the confirmation says before anything happens; the original is a draft again with the cuts it had before, ready to be split at other points. Undoing costs nothing either. It is not available while a part is being mastered or having its edits rendered, or once a part is published or has been split in turn; the page then says why the split cannot be undone.

Splitting has no agent tools.

Get listed

Products → Podcast → Distribution checklist lists the directories in the order to submit to — Apple Podcasts, Spotify, Podcast Index, Pocket Casts, YouTube Music, Amazon Music — each with a link to its submission page, a note on how long it takes, and a status you set yourself (not submitted, submitted, live). Apple may ask you to verify ownership; paste the token it shows into the checklist and it appears in your feed. Spotify mails its verification code to the show's owner address.

See who listens

Products → Podcast → Analytics counts downloads (audio requests from apps and browsers, de-duplicated per listener per day) and plays in the web player, per episode and for a date range. The counts are estimates aligned with the industry's measurement rules, not certified figures.

What you cannot do yet

Stated plainly: one show per site; episodes publish now or as soon as they are mastered, never at a time you choose; an episode's audio cannot be replaced after publishing; your assistant cannot create or change the show, edit an episode's details, unpublish or delete one, or run the editor; once an audio cut is made its edges cannot be dragged, only the whole cut removed; a split cannot be undone once a part is published, and a published episode cannot be split; and the episode's admin page has no player — listen in the editor or on the public page.

Reference

`omumu_podcast_episode_create`

Creates a draft episode on the site's show from an already-uploaded audio resource (upload the file first with omumu_resource_upload_presign; mp3, m4a, and wav are accepted). Mastering starts automatically and the episode is returned in PROCESSING status; publish it right away and it goes live when mastering ends, or poll omumu_podcast_episode_get until it reaches READY, then publish. With deferMastering the episode is returned UNMASTERED and nothing is mastered yet: start mastering with omumu_podcast_episode_master, or publish it and it is mastered first, then goes live.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`audioResourceId`	string	yes	ID of the uploaded audio resource to master into this episode
`deferMastering`	boolean	no	Leave the episode UNMASTERED instead of mastering it now (default false), so no mastering credits are spent on a recording that will be trimmed or split first. Mastering is deferred, never skipped: an UNMASTERED episode is mastered before it can go live
`episodeNumber`	number	no	Optional episode number within the season
`episodeType`	string	no	Episode type: 'full' (default), 'trailer', or 'bonus'
`explicit`	boolean	no	Per-episode explicit override; omit to inherit the show's flag
`noiseReduction`	boolean	no	Remove background noise when mastering (default false). Helps a recording made outdoors or in a noisy room, but can make a clean recording sound harsh. Applied only when the site masters with Auphonic
`season`	number	no	Optional season number
`showNotes`	string	no	Show notes describing the episode
`title`	string	yes	The episode title

`omumu_podcast_episode_get`

Gets one episode by id with its processing status (UNMASTERED, PROCESSING, READY, FAILED, PUBLISHED, SPLITTING while being split into other episodes, SPLIT once it has been), duration, error message when failed, pubDate when published, the latest re-master's status (NONE, PENDING, FAILED) with its error when failed, masteringMethod: what made the current master (auphonic, omumu_chain, or null when not mastered yet or mastered by an older Omumu chain), masteringFallback: why Auphonic could not master it when Omumu's chain made it instead (out_of_credits, key_rejected, production_failed, unreachable, timed_out), null when it did not fall back, noiseReduction: whether the episode is mastered with noise reduction, and publishIntent: whether it publishes itself when mastering ends.

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_list`

Lists the show's episodes with their processing status (UNMASTERED, PROCESSING, READY, FAILED, PUBLISHED, SPLITTING while being split into other episodes, SPLIT once it has been), duration, error message when failed, pubDate when published, the latest re-master's status (NONE, PENDING, FAILED) with its error when failed, masteringMethod: what made the current master (auphonic for the site's connected Auphonic account, omumu_chain for Omumu's own chain, null when not mastered yet or mastered by an older Omumu chain), masteringFallback: why Auphonic could not master it when Omumu's chain made it instead (out_of_credits, key_rejected, production_failed, unreachable, timed_out), null when it did not fall back, noiseReduction: whether the episode is mastered with noise reduction, and publishIntent: whether it publishes itself when mastering ends. The source of a split is listed too; it cannot be edited, published or re-mastered, and its parts are the episodes now.

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

No parameters.

`omumu_podcast_episode_master`

Starts mastering an UNMASTERED episode (one created with deferMastering), returning it in PROCESSING status; it reaches READY when mastering ends. Episodes in any other status are refused with their current status: their mastering has already started.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id
`noiseReduction`	boolean	no	Whether to master with noise reduction, for a recording made outdoors or in a noisy room. Omit to keep the episode's choice.

`omumu_podcast_episode_publish`

Publishes a READY episode: the pubDate is stamped now and the episode appears in the RSS feed. A PROCESSING episode records a publish intent instead and is returned unpublished with publishIntent true: it publishes itself when mastering ends, with the pubDate stamped then, and the intent survives a failed mastering and its retry. An UNMASTERED episode starts mastering with a publish intent the same way, so it goes live mastered. Cancel it with omumu_podcast_episode_publish_cancel until then. A FAILED episode is refused with its status.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_publish_cancel`

Cancels the publish intent of an episode that is still being mastered, or whose mastering failed: it stays a draft when mastering ends. Refused once the episode is published.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_remaster`

Re-masters an episode's original upload with the current mastering chain. A READY episode returns to PROCESSING. A PUBLISHED episode stays in the feed with its current audio while remasterStatus is PENDING; when the new master lands it replaces the audio, and a failure sets remasterStatus FAILED with remasterError and leaves the published audio unchanged. The episode's transcript and edits are kept. Refused, with the reason, for published episodes whose listeners get an edited version, UNMASTERED, PROCESSING or FAILED episodes, episodes with a re-master already pending, the source of a split (SPLITTING or SPLIT), and the parts of a split.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id
`noiseReduction`	boolean	no	Re-master with or without background-noise removal, and keep that choice for the episode; omit to keep the episode's current choice

`omumu_podcast_episode_retry`

Re-runs mastering on a FAILED episode, returning it to PROCESSING. Episodes in any other status are refused with their current status.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_show_get`

Gets the site's podcast show: title, slug, description, language, and the public RSS feed URL. Errors if the site has no show yet (shows are created in the podcast admin).

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

No parameters.

CHAPTER 10

Analytics

You put a page up, sent people to it, and now you want to know: how many came, from where, and what they did. Omumu measures that itself, for every visitor, on your own site — no tracker needed, nothing sent anywhere. When you want more — the Meta Pixel to feed your ads, Google Analytics because you know it, Microsoft Clarity to watch where people stall on a sales page — you connect them, and Omumu puts a consent banner in front of them for the visitors whose law requires one. This chapter is the dashboard, the three trackers, and the rule that keeps the trackers from loading before anyone has said yes. A path such as *Sales and marketing → Analytics* names a group in the admin's navigation and the page under it.

Concepts

Your site counts every visitor. A page view is one person seeing one of your pages: the front page, a sales page, a lesson, a podcast episode. The site records it server-side for everyone, consented or not, and never counts what is not an audience — your own admin pages, sign-in, the platform's addresses, bots, and you and your staff. Visitors are told apart by a *visitor key*: a first-party cookie where the visitor has agreed to analytics (or lives where the law allows it by default), otherwise a key made from their address and browser that changes every day, so a visitor without consent is a new visitor tomorrow. A *visit* is one visitor's page views in a day, with one landing page and one *traffic source*: the

`utm_source` on the link they clicked if there was one, otherwise the referring site, otherwise *direct*.

The dashboard is the last 30 days. Page views, visitors and visits; the top pages with their views and visitors; traffic sources; desktop, mobile and tablet; the most recent page views; and, on plans with advanced analytics, the most recent *events* — form impressions and submissions, emails sent, opened and clicked, enrolments, lessons viewed and completed, quiz starts and completions, add-to-cart, checkout started, purchases, episode downloads and plays. Events belong to the thing they happened to, so a purchase is a purchase of an offer.

Three trackers, each a setting. The *Meta Pixel* loads Meta's script on every page and, with a Conversions API token, also reports from the server — a lead when someone opts in, an add-to-cart when a buy button is clicked, a purchase when payment lands — with one shared event id so Meta does not count twice. *Google Analytics 4* takes a measurement id. *Microsoft Clarity* takes a project id and gives you heatmaps and recordings of consented sessions: where people scrolled to, what they clicked, where they rage-clicked. All three are under *Integrations*; none is required, and a site with none configured shows no banner at all.

The banner loads nothing before yes. Once a tracker is configured, visitors in the EU and EEA (Norway included), the UK, and anywhere the site cannot place see the *Cookie Settings* banner: *Accept All*, *Essential Only*, or choose. Nothing from Meta, Google or Microsoft is fetched until the visitor accepts the matching category — *Marketing* for the Pixel, *Analytics* for GA4 and Clarity — and accepting loads it at once, without a reload. The answer is kept in a cookie for a year, so a visitor who chose *Essential Only* is not asked again on the next visit. Visitors in California see no banner: the trackers are on, and a *Do Not Sell My Personal Information* link in the footer turns Meta's limited data use on. Visitors elsewhere get a dismissible notice with the trackers already on. *Cookie Settings* in the footer reopens the choice everywhere, and taking *Marketing* back tells a Pixel that has already loaded to stop. Your own page views are counted either way; only how long the visitor is remembered depends on the answer.

Hands-on

Read the dashboard

Sales and marketing → *Analytics*. Mira sees 30 days: visitors and visits above page views, her sales page and the opt-in form page at the top of *Top Pages*, *Traffic Sources* split between the newsletter she tagged and direct, and under *Recent Events* the day's form submissions and one purchase. The dashboard has no agent tools; the numbers are read here.

Tag the links you send out

Omumu's traffic sources read the standard campaign parameters, so Mira puts `?utm_source=newsletter&utm_campaign=launch` on the link in her email and `utm_source=facebook&utm_content=ad-a` on each ad. The dashboard then shows the newsletter and the ads as sources rather than *direct*, and no tracker was needed for it.

Connect the Pixel, GA4 or Clarity

Integrations → *Facebook Pixel*: the pixel id from Events Manager and, optionally, a Conversions API access token for the server-side events. *Integrations* → *Google Analytics*: the GA4 measurement id (`G-…`). *Integrations* → *Microsoft Clarity*: the project id. Each page says whether the tracker is configured, and *Clear Settings* removes it. From the moment one is saved, the public pages carry the banner for the visitors who need it.

Check what a visitor sees

Open the site in a private window. In Norway the banner appears before anything loads; the browser's network panel shows no request to Meta, Google or Microsoft until *Accept All*, and the requests appear right after it without reloading. *Essential Only* keeps them away, on this visit and the next.

What you cannot do yet

Stated plainly: the dashboard's range is fixed at the last 30 days, with no date picker and no export. It does not yet credit sales and opt-ins to the source and landing page that brought them — the measurement is recorded for it, the view is not built. Events are named by type and id, not by the offer's or form's

name. There are no analytics tools for your assistant. And Clarity sees only the visitors who consented, so its numbers are a sample, never the denominator; the site's own counts are.

Reference

No MCP tools cover this chapter's capabilities yet.

CHAPTER 11

The Agent Surface

The rest of this manual keeps saying "tell your assistant". This chapter is about what makes that sentence true: the single connection point your AI talks to, the credentials that let it in, and the controls that decide — precisely — what it may read, what it may change, and how you take that access back. You don't need to be technical to use any of it; connecting your AI to your site takes a few taps on one page.

If you *are* technical, this chapter is also the receipts. Omumu's agent surface is an MCP server, and every claim below maps to a scope, a permission, or a tool you can check against the generated catalog at the end of this chapter. That catalog — every tool, its parameters, its write category, its access restrictions, and the full permission list — is regenerated from the platform's own capability inventory on every build of this manual, and the build fails if that inventory ever drifts from the code — the catalog below cannot quietly go out of date.

Concepts

The agent surface is one door. Every AI client — Claude on the web, ChatGPT, Claude Code in a terminal, a script you wrote — talks to your site through the same endpoint: your site's address plus `/mcp`, speaking the Model Context Protocol (MCP), the open standard AI tools use to operate external systems. There is also one shared address for every site, `https://omumu.com/`

`mcp`. It leads to the same door: it asks for your email first, finds the site you run, and hands you to that site's own consent page. There is no second, softer API for agents. An agent acts as an authenticated identity on your site and passes the same permission checks the admin UI does; the surface is also advertised in machine-readable form — an API catalog at `/.well-known/api-catalog`, an MCP server card at `/.well-known/mcp/server-card.json`, and a plain-text onboarding guide for AI tools at `/llms-onboarding.md` on your site — so an agent pointed at your domain can find the door on its own. MCP access is a plan feature: it must be enabled on your site's subscription tier for the surface to be live.

A **tool** is the unit of capability: one named operation, like `omumu_course_create` or `omumu_offer_list`, with typed parameters. What your assistant "can do" is exactly the set of tools its credential can call — nothing more. Discovery is **fail-closed**: when an agent asks the server what tools exist, it is shown only the ones its credential can actually use. An agent connected with read-only access doesn't see a greyed-out "delete course" button; it doesn't see the tool at all. The full catalog lives in the Reference below, grouped by the chapter of this manual that explains it.

There are **two ways in**, and choosing between them is simple:

An **OAuth connection** is for interactive assistants like Claude. The AI vendor's app registers itself with your site, and you approve the connection on a consent screen *on your own site* — it names the client and your site, lists exactly what it is asking for, and offers Deny and Authorize. Only the people who run the site — its Headmaster and Staff — can approve: a Student who tries is told to ask the site's owner. The check is not a one-off. It runs again on every call, so a connection stops working the moment its person is blocked, removed, or no longer Staff. Under the hood this is OAuth 2.0 with the current hardening expected of it: authorization code flow with PKCE, and rotating refresh tokens — every refresh issues a new token and retires the old one, and if a retired token is ever replayed (the signature of a stolen token), the site revokes that connection's entire token family on the spot.

An **API key** is for headless use: scripts, scheduled jobs, Claude Code on your laptop. A key looks like `omumu_` followed by a random secret and your site id, is stored only as a salted hash (like a password — the platform itself cannot recover it), and is shown to you exactly once, at creation. Each key carries its own permission picklist, so a reporting script can hold a key that reads courses

and nothing else — and the picklist can be changed later without replacing the key.

Scopes are the language of consent. An OAuth client asks in scopes — `course:read`, `offer:write`, `email:read`, and so on — and each scope maps to a named site permission that every tool checks. Three rules keep this honest. A *write scope implies its read*: `course:write` also grants `course:read`, because editing blind is useless. A client that asks for *nothing* gets the minimum: read access to courses, only. And a scope that reads people's personal data is **never implied by another scope**: it has to be asked for by name, and you see it on the consent screen as its own line before it's live. There are two. `leads:read` unlocks names and emails from your opt-in forms; Omumu doesn't offer it to connecting apps, so a client must know to ask for it. `handin:read` unlocks what your students hand in — their text, a text file they upload, the transcript of a recording, the AI's feedback and your staff's — and, because reading hand-ins is something a creator connects Claude *for*, it is offered: an app that asks for everything your site offers will ask for it. Only a Headmaster's connection can hold it. That rule is a recorded architectural decision, not a habit.

Some permissions no agent can hold at all, whichever way it came in: managing user accounts, changing site settings, and managing the API keys themselves are excluded from both the OAuth scope list and the API-key picklist. The last one matters most: an agent cannot mint, widen, or delete credentials — so a misbehaving agent can't promote itself, and revoking it is always a decision only you can make.

Every connection can ask where it is. Two read-only tools need no permission at all. `omumu_site_context` tells the assistant which site this connection works on, your role there, which features the site has, and whether the site is connected to HighLevel, which a quiz needs before it can deliver its respondents there — the assistant is told to call it first. `omumu_profile_get` gives the connection a name made of the person's first name and the site's name, so an AI app that holds several Omumu connections shows each site as its own account.

Writes are categorized and rate-limited. Every tool carries a write category: read-only tools (listing, getting, reporting) have none; the rest are standard writes or skill writes (skills are the platform's installable agent behaviors — their own chapter). Writes made with an API key pass a per-key, per-category rate limit, so a script stuck in a loop gets slowed down before it can flood your site with half-made courses.

Finally, the catalog also holds a few tools Omumu uses to operate the platform itself. They are not part of this manual, and they are not part of your surface: fail-closed discovery means the agents on your site never even see them.

Stated plainly, the gaps: there are no tools today for site settings, funnels, or analytics dashboards — those are admin-UI work. Files are covered: your assistant can upload anything the admin UI accepts — PDFs, images, audio, documents, archives and video — with `omumu_resource_upload_presign`, and the file travels straight to storage instead of through the conversation. `omumu_generate_image` makes new images. The catalog below is the honest inventory: if a tool isn't in it, your assistant can't do it, and this manual won't pretend otherwise.

Hands-on

The walkthroughs continue with Mira from the Selling chapter — she runs her course business by talking to her AI assistant. This is the chapter where that setup actually happens.

Connect your AI from the setup page

Mira opens *Integrations → Set up your AI assistant* and answers one question: which AI does she use — Claude, ChatGPT, or Other. The page has already named the connector after her site; she can shorten the name, and the buttons below follow her edit.

She picks Claude and taps *Add to Claude*. Claude's add-connector dialog opens already filled in with the name and her site's address; she confirms. Claude finds the OAuth setup on its own and sends her to a consent page *on her own site*, which names Claude and her site and lists the scopes it wants. The list is already narrowed to what her account can actually grant, and it follows the rules from Concepts: no buyer PII, and reading her students' hand-ins on a line of its own, nothing a scope didn't name. She clicks Authorize. Back on the setup page, the status turns green — "Connected!" — and suggests a first message to try.

Every Claude plan can add a custom connector; the free plan allows one, and on Claude Team or Enterprise an owner adds it for the organisation. The

consent screen is the grant moment: Deny costs nothing and can be retried; Authorize is what creates the connection.

The other choices work the same way, from the same page. ChatGPT gets numbered steps for making a personal plugin for the site, with the name and address ready to copy. *Other* has one-tap installs for Cursor and VS Code, and the one-line commands that add the site to Claude Code and Codex. The bare MCP address and the API keys sit under *Advanced*.

Connect through omumu.com/mcp

An AI app can also be pointed at the shared address, `https://omumu.com/mcp`, instead of the site's own. Signing in then starts on Omumu: Mira enters the email she uses on her site and either types the 6-digit code Omumu emails her — it works once, for ten minutes — or uses her password. If that email runs several Omumu sites, she chooses the one this connection should work on; with one, she goes straight on. From there it is her site's own consent page, exactly as above. Each site she connects this way is its own connection, and in apps that hold several accounts, its own account.

Give a script its own key

For Claude Code — and anything else headless — Mira creates an API key instead. In the admin UI: *Integrations* → *MCP API Keys* → *create*, where she names the key and ticks the permissions this key should carry — for her weekly sales-summary script, read permissions only. The key is displayed once, on the creation page, with a copy-it-now warning; the same page shows the exact one-line command that registers it with Claude Code:

```
claude mcp add-json omumu '{"type":"http","url":"https://<your
site>/mcp","headers":{"Authorization":"Bearer <your key>"}}'
```

From then on, Claude Code on her laptop operates the site with exactly that key's permissions. Lose the key and there is nothing to recover — the platform kept only the hash — so she creates a new one and deletes the old. One key per purpose is the honest pattern: the key ticked for writes is not the key her reporting script holds.

When the script's job grows — say it should now draft email sequences too — Mira doesn't mint a new key. She opens *Integrations → MCP API Keys*, edits the key, and ticks what it may do now. The change applies from the key's next request; the key itself stays the same, so the script needs no new setup. She can tick only permissions she may grant herself, and a key keeps at least one. If the key carries a permission she can't grant, her edit leaves it in place, and the page names the permissions that stay.

Stated plainly: creating and managing API keys is **UI-only** by design. There is no tool for it — that's the "no agent manages credentials" rule from Concepts doing its job.

Let your AI read what students hand in

Mira's course asks each student to record a practice call and hand it in. Every morning she wants her assistant to read the new ones and tell her what people keep getting stuck on, so she can fix the lesson instead of answering the same question ten times.

> "Which hand-ins came in on the practice-call assignment since yesterday morning, and what are people struggling with?"

The assistant lists the assignment's hand-ins since that time (`omumu_submission_list`): who handed in, when, their Light, and whether the AI or her staff have given feedback yet. Then it reads each one (`omumu_submission_get`): what the student wrote, the transcript of the recording, and the feedback rows. A student who uploads the call's transcript as a .txt or .md file instead of pasting it gets read too: the assistant sees the file's text, up to 1 MB. Mentor notes never come back, because they belong to the mentor. Both tools only read: nothing gets marked as seen and nothing is changed. Asking for hand-ins since the time of the last one it saw is how a daily check picks up only the new ones.

This needs `handin:read` on the connection. A connection made before the scope existed doesn't have it, and refreshing the connection won't add it, because a refresh has no consent screen and a personal-data scope is only ever granted on one. To add it, remove the connector in the AI app and add it again. The new consent screen names the hand-ins line, and Mira authorizes it. Only

a Headmaster can connect this way; a Staff member's connection never gets the scope.

Take an agent's access back

Revoking is the control that makes granting safe, so here is exactly where it stands.

For API keys it is immediate and self-serve: *Integrations* → *MCP API Keys*, delete the key, confirm. The hash is gone and every request bearing that key fails from that moment. Nothing the agent did survives as access — only as ordinary site data you can still edit.

For OAuth connections it is self-serve too. The setup page lists *Connected AIs*: each one with the AI's name, who connected it, when, and when it was last active. *Disconnect* ends that connection's access and refresh tokens, so the AI's next call fails; using it again means connecting it anew. A Headmaster sees every connection on the site; Staff see only their own. Connecting the same AI again replaces its earlier entry instead of adding a second one. On top of that, the token machinery limits the damage of a stolen token by itself — refresh tokens rotate on every use, and a replayed old token kills the whole family.

A first real conversation

Connected, Mira starts the way anyone should:

> "What can you actually do on my site?"

The assistant calls `omumu_site_context` first, so it knows which site it is on and what Mira may do there, then answers from its live tool list — which, fail-closed, is exactly what her grant allows. Then she gives it the real test, the same pre-sale move from the Selling chapter, end to end:

> "Create a course called AI Email Triage — one module, five lesson outlines — and put up a 790 NOK early-bird offer for it."

The assistant creates the course (`omumu_course_create`), a module inside it (`omumu_module_create`), five draft lessons (`omumu_lesson_create`), then the offer around it (`omumu_offer_create`, `omumu_offer_add_product`) and reports back

with the checkout URL. Mira opens *Products → Courses* and *Sales and marketing → Offers* and finds everything the conversation claimed — the same objects, editable in the same UI, indistinguishable from work done by hand. That is the whole design: the conversation is not a veneer over a separate system. It is your site, through one door, with the keys you chose to hand over.

Reference

Getting Started

`omumu_brand_get`

Gets the site's Brand: the published choices and revision the site wears, the draft choices and whether they differ from published, the draft's derived light and dark color roles and any contrast adjustments, the font catalog identifiers, roundness steps and CTA colors to choose from, and the admin editor URL. Publishing is a human step: the creator publishes the draft in admin at editorUrl.

- **Write category:** read-only
- **Access:** requires the `BRAND_READ` permission

No parameters.

`omumu_brand_update`

Merges any of the given choices into the site's Brand draft; choices left out stay as they are, and an invalid color, unknown font or logo that is not this site's refuses the whole update. Returns the Brand as omumu_brand_get does. The draft is not live: the Site's pages and its public Brand JSON keep the published Brand until the creator reviews the draft and publishes it themselves in admin, at the returned editorUrl. There is no tool to publish.

- **Write category:** standard write
- **Access:** requires the `BRAND_WRITE` permission

Parameter	Type	Required	Description
`accentColor`	string	no	The accent color as #rrggbb. Becomes the tertiary role.
`ctaColor`	string	no	Which color drives calls to action: primary (default) or accent. Filled buttons (site, quiz, checkout, opt-in) follow this choice; headings, links, progress bars and meters stay on the primary regardless.
`font`	string	no	A font identifier from the catalog that omumu_brand_get lists under fonts, e.g. raleway.
`headingFont`	string	no	A font identifier from the catalog that omumu_brand_get lists under fonts, for headings only; body text keeps font. Give "text" to clear it back to the text font.
`markDarkResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo mark for dark surfaces (square, at least 1024 px). It is checked like an upload in the Brand editor.
`markResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo mark (square, at least 1024 px). It is checked like an upload in the Brand editor.
`primaryColor`	string	no	The primary color as #rrggbb, e.g. #215fa6. Kept exactly wherever it can carry readable text; otherwise nudged just enough to pass contrast, reported under adjustments.
`typeDarkResourceId`	string	no	Id of an image already uploaded to this site (omumu_resource_upload_presign) to use as the logo type for dark surfaces (400 px, at least 400 px wide). It is

Courses & Content

`omumu_answer_create`

Creates an answer option for a question. On an outcome quiz, link this answer to one or more buckets using omumu_option_point_create after creating the answer. On a score quiz, set points so choosing this answer adds (or, for a reverse-worded statement, subtracts) that many points from its question's scale. tag, when set, is applied at completion to an identified respondent who chose this answer, alongside the bucket, door and completion tags — the way a single background answer ("do you have the book?") becomes a fact on the contact, usable by HAS_TAG.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`label`	string	yes	The answer label shown to users
`points`	number	no	Points this answer carries toward its question's scale (score quizzes only; default 0)
`questionId`	string	yes	The question ID
`tag`	string	no	Tag applied at completion to an identified respondent who chose this answer
`value`	string	no	The answer value (defaults to label)

`omumu_answer_update`

Updates an answer option's label, value, points (score quizzes only) and/or tag (applied at completion to an identified respondent who chose this answer)

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`answerId`	string	yes	The answer ID
`label`	string	no	New answer label
`points`	number	no	New points value (score quizzes only)
`tag`	string	no	Tag applied at completion to an identified respondent who chose this answer, or null to remove
`value`	string	no	New answer value

`omumu_assignment_create`

Creates an assignment attached to exactly one scope: a course, a module, a lesson, or the site itself. Provide at most one of courseId, moduleId, or lessonId — more than one is rejected. Omitting all three creates a SITE assignment, shown to creators as All courses: attached to the whole site and handed in on its own page /assignments/{aid}, where students write about everything they have rather than one course. Students find it first on their Assignments page (/assignments); creators find it in the All courses group of /admin/assignments. A new SITE assignment goes last in the All courses order; move it with omumu_assignment_reorder. It counts toward no course's completion and is rated against each student's own accessible courses. COURSE scope: the assignment appears on the course homepage and is visible across all its modules and lessons. MODULE scope: the assignment is shown within a specific module and reflects that module's work. LESSON scope: the assignment is tied to a single lesson and reflects its content. New assignments default to DRAFT status; pass status=PUBLISHED to make the assignment immediately visible to students. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`asksForLight`	boolean	no	Check-in: every hand-in must include the student's own Light — green, yellow or red. Defaults to false.
`courseId`	string	no	Attach the assignment to this course (COURSE scope). Must not be combined with moduleId or lessonId.
`description`	string	no	The assignment instructions or description (Markdown)
`lessonId`	string	no	Attach the assignment to this lesson (LESSON scope). The owning course is derived automatically. Must not be combined with courseId or moduleId.
`moduleId`	string	no	Attach the assignment to this module (MODULE scope). The owning course is derived automatically. Must not be combined with courseId or lessonId.
`notifyStaffOnHandIn`	boolean	no	Email every Headmaster and Staff member on the site when a student hands in. Defaults to true.
`ratingPrompt`	string	no	Prompt shown to instructors when rating a submission (optional)
`status`	string	no	Initial status: DRAFT (default, not visible to students) or PUBLISHED (visible)
`title`	string	yes	The assignment title

`omumu_assignment_list`

Returns all assignments (DRAFT and PUBLISHED) for a course, with each assignment's scope kind (COURSE, MODULE, LESSON, or SITE) and the scope's id stated explicitly. Without a courseId it returns the site's SITE assignments only — the All courses assignments, attached to the whole site rather than to any course, each handed in on its own page /assignments/{aid} — in the All courses order, with each one's 0-based sortOrder; order them with omumu_assignment_reorder. Course, module and lesson assignments report no sortOrder. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	no	The course whose assignments to list; omit to list the site's SITE assignments (All courses)

`omumu_assignment_reorder`

Moves a SITE assignment (an All courses assignment) to a 0-based position in the site's All courses order, the order students see first on /assignments and the admin sees in the All courses group of /admin/assignments. The other SITE assignments shift to make room; a position past the end places the assignment last. Draft ones hold their place in the order too. Only SITE assignments have an order: a course, module or lesson assignment is refused, as is an assignment of another site. Read the current order with omumu_assignment_list without a courseId. Returns the moved assignment with its new sortOrder. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`assignmentId`	string	yes	The SITE assignment to move
`newSortOrder`	integer	yes	The position to move it to, 0-based (0 is first)

`omumu_assignment_update`

Updates an existing assignment's title, description, ratingPrompt, status, and/or scope. Provide at most one of courseId, moduleId, or lessonId to set the new scope; omitting all three means the SITE scope, so omit them exactly when updating a SITE assignment. The new scope must belong to the same course as the assignment — a scope change across the course boundary (to or from the SITE scope) is rejected: that is a different assignment. A SITE assignment is an All courses assignment, handed in on its own page /assignments/{aid}; its position in the All courses order is changed with omumu_assignment_reorder, not here. Fields not provided are left unchanged. Requires the MCP_COURSES feature (Starter tier or above).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`asksForLight`	boolean	no	Check-in: every hand-in must include the student's own Light — green, yellow or red. Omit to leave unchanged.
`assignmentId`	string	yes	The assignment ID to update
`courseId`	string	no	New COURSE scope. Must not be combined with moduleId or lessonId.
`description`	string	no	New assignment instructions (Markdown)
`lessonId`	string	no	New LESSON scope. Must not be combined with courseId or moduleId.
`moduleId`	string	no	New MODULE scope. Must not be combined with courseId or lessonId.
`notifyStaffOnHandIn`	boolean	no	Email every Headmaster and Staff member on the site when a student hands in. Omit to leave unchanged.
`ratingPrompt`	string	no	New rating prompt for instructors
`status`	string	no	New status: DRAFT (hidden from students) or PUBLISHED (visible)
`title`	string	no	New assignment title

`omumu_bucket_create`

Creates a result bucket for the quiz. On an outcome quiz it's a category users are sorted into by plurality — after creating buckets, link answers to them using omumu_option_point_create. On a score quiz it's a profile: set minPercent (0-100) and the bucket with the highest minPercent at or below the respondent's total percent wins; a bucket with no minPercent never wins. Each bucket can trigger actions: send email sequence, grant course access, redirect to URL. description may be Quill delta JSON when descriptionEncoding is

QUILL_JSON. On a score quiz, description supports result-copy placeholders (see omumu_note_create for the full set). videoResourceId attaches one video, shown after the description on the result page — the resource must already exist on this site (via the presigned video upload) and be a video.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Bucket description
`descriptionEncoding`	string	no	How description is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`minPercent`	number	no	Score quizzes only: lower percent bound (0-100) of the quiz's total this bucket wins at or above
`name`	string	yes	The bucket name (e.g., 'Beginner', 'Advanced')
`quizId`	string	yes	The quiz ID
`videoResourceId`	string	no	Id of an existing video site resource on this site, shown after the description on the result page

`omumu_bucket_update`

Updates a bucket's properties including actions (email sequence, course, redirect, tag) and, for a score quiz, its percent-range profile fields (minPercent, descriptionEncoding). Pass null for an action to remove it, or leave the key out to keep it unchanged. emailSequenceId and courseId must be a sequence or course on this site. videoResourceId attaches one video (must belong to this site and be a video); pass null to remove it.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`bucketId`	string	yes	The bucket ID
`courseId`	string	no	Course on this site to grant access to; null removes the course
`description`	string	no	New description
`descriptionEncoding`	string	no	How description is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`emailSequenceId`	string	no	Email sequence on this site to subscribe user to; null removes the sequence
`minPercent`	number	no	Score quizzes only: lower percent bound (0-100) of the quiz's total this bucket wins at or above
`name`	string	no	New bucket name
`redirectUrl`	string	no	URL to redirect to after quiz; null removes the redirect
`successMessage`	string	no	Message shown after quiz completion
`tagName`	string	no	Tag to apply to user; null removes the tag
`videoResourceId`	string	no	Id of an existing video site resource on this site; null removes the current video

`omumu_checklist_batch_create`

Creates multiple checklist items for a lesson in one call (max 100). Atomic — all items are created or none. Useful for AI workflows that generate complete checklists at once.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`items`	array	yes	Array of objects with 'description' (required, max 1000 chars) and 'xp' (optional, 0-10000, default 0)
`lessonId`	string	yes	The lesson ID

`omumu_checklist_item_create`

Adds a new checklist item to a lesson. Checklist items are actionable tasks that students complete to earn XP. Items are auto-appended at the end of the list.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	yes	The checklist item text, max 1000 chars (what the student should do)
`lessonId`	string	yes	The lesson ID to add the checklist item to
`xp`	integer	no	XP points awarded when completed, 0-10000 (default 0)

`omumu_checklist_item_delete`

Permanently deletes a checklist item and any associated user progress. Remaining items are re-sorted automatically.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID to delete

`omumu_checklist_item_list`

Lists all checklist items for a specific lesson, ordered by sort order.

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`lessonId`	string	yes	The lesson ID

`omumu_checklist_item_reorder`

Changes the sort order of a checklist item within its lesson.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID
`newSortOrder`	integer	yes	The new sort order position (0-based)

`omumu_checklist_item_update`

Updates the description and/or XP value of an existing checklist item.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`checklistItemId`	string	yes	The checklist item ID
`description`	string	no	New description text, max 1000 chars (optional)
`xp`	integer	no	New XP value, 0-10000 (optional)

`omumu_course_create`

Creates a new course in the user's site with the specified title and tagline. The course is live immediately for learners with access — there is no draft step — and the result's publicUrl is the address of its learner-facing course page, which requires sign-in and access to the course. After creating, add modules (omumu_module_create) and lessons (omumu_lesson_create) to build the course structure. To grant access via an opt-in form, pass the course ID to omumu_optinform_update(courseId).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`tagline`	string	no	The course tagline
`title`	string	yes	The course title

`omumu_course_delete`

Permanently deletes a course and all its content

- **Write category:** standard write
- **Access:** requires the `COURSE_DELETE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID to delete

`omumu_course_duplicate`

Creates a complete copy of an existing course including all modules and lessons

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID to duplicate

`omumu_course_get`

Retrieves detailed information about a specific course

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`includeModules`	boolean	no	Whether to include modules and lessons

`omumu_course_list`

Lists all courses in the user's site with optional module inclusion

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`includeModules`	boolean	no	Whether to include modules in the response

`omumu_course_update`

Updates the basic information of a course. The change is live immediately for learners with access — there is no draft step — and the result's publicUrl is the address of its learner-facing course page, which requires sign-in and access to the course.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`description`	string	no	New course description
`gatingMode`	string	no	Course gating mode: OPEN (default — course access opens every module) or SEQUENTIAL (completing a module unlocks the next; the unlocking is built in)
`tagline`	string	no	New course tagline
`title`	string	no	New course title
`watchThresholdPercent`	integer	no	How much of a video counts as watched, in percent. One of 25, 50, 75, 90, 100 (default 75)

`omumu_door_create`

Creates a door on a quiz: the next step (heading, text, button label/URL or an embedded checkout, optional tag) offered on the result page, chosen from the respondent's answers rather than the score. Doors are tried in sortOrder and the first whose condition holds wins; a door with empty clauses always holds and is the default — put it last. A clause is {kind: "ANSWER_CHOSEN", questionId, answerId} (a specific answer was chosen) or {kind: "QUESTION_ANSWERED", questionId} (the question has any answer, non-blank for free text). Every clause in a door's condition must hold (AND

only — no OR, no NOT); express "either of these answers" as two doors with the same copy. Every clause's question (and answer, for ANSWER_CHOSEN) must belong to this quiz. Works on both outcome and score quizzes. On a score quiz, heading and text support placeholders: Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`buttonLabel`	string	no	The call-to-action button's label
`buttonUrl`	string	no	The button's URL: absolute http(s) or a site-relative path
`checkoutEmbedUrl`	string	no	An embedded checkout shown in place of the button: an https URL on https://app.edpay.no (the provider's embed link for the offer, e.g. https://app.edpay.no/offer/518/10998/book?embed=true); framed as a sandboxed iframe on the result page
`clauses`	array	no	Inline answer-condition clauses: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWERED" questionId, answerId?}]. Empty or omitted makes this the default door.
`heading`	string	no	Heading shown on the result page
`name`	string	yes	Internal name for this door (e.g. "Book a call")
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Evaluation order, first match wins (defaults to last)
`tag`	string	no	Tag applied to the respondent when this door wins, so automations can start the right follow-up
`text`	string	no	Body copy shown under the heading
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML

`omumu_door_delete`

Deletes a door and its clauses.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`doorId`	string	yes	The door ID to delete

`omumu_door_update`

Updates a door's copy, checkout embed, sort order, tag and/or clauses. A clauses array replaces this door's condition wholesale rather than appending to it — include every clause you want kept. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`buttonLabel`	string	no	New button label
`buttonUrl`	string	no	New button URL
`checkoutEmbedUrl`	string	no	An embedded checkout shown in place of the button: an https URL on https://app.edpay.no (the provider's embed link for the offer, e.g. https://app.edpay.no/offer/518/10998/book?embed=true); framed as a sandboxed iframe on the result page; null removes it and brings the button back
`clauses`	array	no	Replaces this door's clauses wholesale: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWE questionId, answerId?}]
`doorId`	string	yes	The door ID
`heading`	string	no	New heading
`name`	string	no	New internal name
`sortOrder`	number	no	New evaluation order
`tag`	string	no	New tag applied when this door wins
`text`	string	no	New body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML

`omumu_generate_image`

Generates an image from a text prompt using AI and stores it as a resource. Optionally attach the generated image directly to a course, module, or lesson. When targeting an entity, aspect ratio defaults to 16:9 (ideal for course illustrations).

- **Write category:** standard write
- **Access:** requires the `MEDIA_GENERATE` permission

Parameter	Type	Required	Description
`aspectRatio`	string	no	Aspect ratio: 1:1, 16:9, 9:16, 4:3, 3:4. Default: 16:9 when targeting, 1:1 otherwise.
`highQuality`	boolean	no	Use high quality model (slower, requires Premium/Enterprise tier). Default: false
`prompt`	string	yes	Text description of the image to generate. Be descriptive for best results.
`targetId`	string	no	ID of the entity to attach the image to. Required if targetType is provided.
`targetType`	string	no	Entity type to attach image to: 'course', 'module', or 'lesson'. If provided, targetId is also required.

`omumu_lesson_create`

Creates a new lesson within a module. Returns a compact summary {id, title, sortOrder} only — the submitted description is not echoed back, to keep responses small when authoring long-form content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	The lesson description
`moduleId`	string	yes	The module ID
`title`	string	yes	The lesson title

`omumu_lesson_resource_create`

Creates a new content block within a lesson. Content can be Markdown or QuillDelta JSON (auto-detected). Optionally attaches an uploaded SiteResource (e.g. a video) via resourceId. Use this to add rich text content to lessons - the 'meat' of course content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	yes	The content in Markdown or QuillDelta JSON format. Markdown supports: # headers, **bold**, *italic*, links, - bullets, 1. numbered lists, > blockquotes, `code`
`lessonId`	string	yes	The lesson ID to add content to
`resourceId`	string	no	Optional SiteResource ID (e.g. an uploaded video) to attach to this lesson part. Must belong to the same site as the lesson.
`title`	string	yes	Title of the content block

`omumu_lesson_resource_delete`

Permanently deletes a content block from a lesson.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`lessonResourceId`	string	yes	The lesson resource ID to delete

`omumu_lesson_resource_list`

Lists all content blocks for a specific lesson.

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`lessonId`	string	yes	The lesson ID

`omumu_lesson_resource_update`

Updates an existing content block. Content can be Markdown or QuillDelta JSON (auto-detected). Optionally replaces the attached SiteResource. Pass an empty string for resourceId to clear the attachment.

- **Write category:** standard write

- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON (optional)
`lessonResourceId`	string	yes	The lesson resource ID
`resourceId`	string	no	New SiteResource ID to attach (optional). Pass empty string to clear the current attachment. Must belong to the same site as the lesson.
`title`	string	no	New title (optional)

`omumu_lesson_update`

Updates a lesson's title and/or description. Description can be Markdown or QuillDelta JSON (auto-detected). Returns a compact summary {id, title, sortOrder} only — the submitted description is not echoed back, to keep responses small when authoring long-form content.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New lesson description (Markdown or QuillDelta JSON)
`lessonId`	string	yes	The lesson ID
`title`	string	no	New lesson title

`omumu_module_copy`

Copies a module (including all its lessons and resources) from its current course into a different target course. Both the source module's course and the target course must belong to the user's site.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`moduleId`	string	yes	The ID of the module to copy
`targetCourseId`	string	yes	The ID of the course to copy the module into

`omumu_module_create`

Creates a new module within a course

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`description`	string	no	The module description
`title`	string	yes	The module title

`omumu_module_list`

Lists all modules in a specific course with optional lesson inclusion

- **Write category:** read-only
- **Access:** requires the `COURSE_READ` permission

Parameter	Type	Required	Description
`courseId`	string	yes	The course ID
`includeLessons`	boolean	no	Whether to include lessons in each module

`omumu_module_update`

Updates a module's title and/or description. Description can be Markdown or QuillDelta JSON (auto-detected).

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New module description (Markdown or QuillDelta JSON)
`moduleId`	string	yes	The module ID
`title`	string	no	New module title

`omumu_note_create`

Creates a note on a quiz: answer-conditioned copy shown on the result page between the lowest-scale section and the door, whenever its condition holds. Unlike a door, every matching note is shown, in sortOrder (notes compose rather than compete) — there is no first-match-wins and no default note. A clause is {kind: "ANSWER_CHOSEN", questionId, answerId} (a specific answer was chosen) or {kind: "QUESTION_ANSWERED", questionId} (the question has any answer, non-blank for free text; use this to quote a free-text answer back with [answer:]). Every clause in a note's condition must hold (AND only — no OR, no NOT). Empty or omitted clauses makes the note always show. Every clause's question (and answer, for ANSWER_CHOSEN) must belong to this quiz. A note has no button and no tag. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`clauses`	array	no	Inline answer-condition clauses: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSWERED" questionId, answerId?}]. Empty or omitted makes this note always show.
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Display order among notes (defaults to last)
`text`	string	yes	The note's body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML

`omumu_note_delete`

Deletes a note and its clauses.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note ID to delete

`omumu_note_update`

Updates a note's text, sort order and/or clauses. A clauses array replaces this note's condition wholesale rather than appending to it — include every clause you want kept. Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`clauses`	array	no	Replaces this note's clauses wholesale: [{kind: "ANSWER_CHOSEN"|"QUESTION_ANSW questionId, answerId?}]
`noteId`	string	yes	The note ID
`sortOrder`	number	no	New display order among notes
`text`	string	no	New body copy
`textEncoding`	string	no	How text is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML

`omumu_option_point_batch`

Outcome quizzes only. Efficiently set up quiz scoring by creating or updating multiple option points in a single call. If an answer-bucket pair already exists, updates its weight. Use this after creating questions, answers, and buckets to configure the scoring matrix. Example: [{answerId: 'a1', bucketId: 'b1', weight: 2.0}, {answerId: 'a1', bucketId: 'b2', weight: 1.0}]

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPoints`	array	yes	Array of option points, each with: answerId (required), bucketId (required), weight (optional, default 1.0)

`omumu_option_point_create`

Outcome quizzes only. Creates an option point linking an answer to a bucket with a weight. When a user selects this answer, the weight is added to the bucket's score. After all questions, the bucket with the highest total score becomes the user's result. Example: Answer 'I exercise daily' → Bucket 'Fitness Enthusiast' with weight 2.0. A score quiz does not use option points — give its answers points instead.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`answerId`	string	yes	The answer ID to link
`bucketId`	string	yes	The bucket ID to assign points to
`weight`	number	no	Point weight (default: 1.0). Higher weights have more influence on final bucket selection.

`omumu_option_point_delete`

Outcome quizzes only. Removes the link between an answer and a bucket, so selecting this answer no longer contributes to that bucket's score.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPointId`	string	yes	The option point ID to delete

`omumu_option_point_list`

Outcome quizzes only. Lists all option points for a specific question, showing which answers are linked to which buckets and their weights.

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

Parameter	Type	Required	Description
`questionId`	string	yes	The question ID

`omumu_option_point_update`

Outcome quizzes only. Updates an option point's weight to adjust scoring influence.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`optionPointId`	string	yes	The option point ID
`weight`	number	no	New weight value

`omumu_question_create`

Creates a new question in a quiz. After creating a question, add answer options with omumu_answer_create. RADIO type shows single-choice radio buttons, CHECKBOX allows multiple selections. On a score quiz, assign the question to a scale with scaleId (created with omumu_scale_create) so its answers' points feed that scale; a question with no scaleId is an unscored background question.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`header`	string	yes	The question text
`quizId`	string	yes	The quiz ID
`scaleId`	string	no	Scale this question feeds (score quizzes only); omit for an unscored question
`type`	string	no	Question type: RADIO, CHECKBOX, TEXTAREA, RADIO_OTHER, CHECKBOX_OTHER

`omumu_question_update`

Updates a question's header, type and/or scale assignment (score quizzes only). The scale must belong to the same quiz.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`header`	string	no	New question text
`questionId`	string	yes	The question ID
`scaleId`	string	no	Scale this question feeds, or null to make it unscored (score quizzes only)
`type`	string	no	New question type

`omumu_quiz_build`

Creates a whole quiz — fields, scales, questions with answers, result buckets, doors, notes and (outcome quizzes) option points — from a single definition object, all or nothing: every cross-reference is validated before anything is

written, and a bad reference fails the call naming every problem found, leaving no quiz, scale, question, answer, bucket, door, note or option point behind. Use this instead of the granular tools (omumu_quiz_create, omumu_scale_create, omumu_question_create, omumu_answer_create, omumu_bucket_create, omumu_door_create, omumu_note_create, omumu_option_point_create) when authoring a quiz from scratch; those granular tools remain the way to edit an existing quiz afterwards — this tool only creates. definition.quiz mirrors omumu_quiz_create's fields: {label (required), scoring: OUTCOME|SCORE, completionTag?, captureTag?, askFirstName?, noClearLowestText?, noClearLowestTextEncoding?, preQuizOptinFormId?, postQuizOptinFormId?, destinations? (e.g. ["HIGHLEVEL"]), highLevelResultLinkField?}. destinations lists where, besides Omumu, the quiz delivers each identified respondent once their result is saved; the only destination is HIGHLEVEL, which upserts the respondent as a contact (first name, email, phone) in the site's HighLevel sub-account with the tags this quiz applied to them. Adding HIGHLEVEL is refused unless the site's plan includes CRM_DESTINATIONS and a Headmaster has connected HighLevel (omumu_site_context reports highLevelConnected); no tool reads or writes the connection itself. highLevelResultLinkField names a custom contact field in that sub-account (its field key, e.g. contact.quiz_result_link; the contact. prefix is dropped) that receives the respondent's result link at completion, written before the tags so a workflow fired by the completion tag can use it; it needs the HIGHLEVEL destination, and null removes it. definition.scales (SCORE quizzes only) is [{key (required, your own id — any string unique among scales), name, sortOrder?, lowestScaleText?, lowestScaleTextEncoding?, levels?: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]}]. definition.questions is [{key (required, unique among questions), header, type: RADIO|CHECKBOX|TEXTAREA|RADIO_OTHER|CHECKBOX_OTHER, sortOrder?, scaleKey? (must name a scale in this same definition), answers: [{key (required, unique among every answer in this definition), label, value?, points? (SCORE quizzes), sortOrder?, tag? (applied at completion to an identified respondent who chose this answer)}]}]. definition.buckets is [{key (required, unique among buckets), name, description?, descriptionEncoding?, minPercent? (SCORE quizzes, 0-100), tag?, redirectUrl?, successMessage?, emailSequenceId?, courseId?, videoResourceId?}] — emailSequenceId, courseId and videoResourceId must already exist on this site. definition.doors is [{name, heading?, text?, textEncoding?, buttonLabel?, buttonUrl?, checkoutEmbedUrl?, tag?, sortOrder?, clauses?: [{kind: ANSWER_CHOSEN|QUESTION_ANSWERED, questionKey,

answerKey?}]}] — doors have no key, nothing references one; tried in sortOrder, first matching condition wins, empty clauses is the default door. definition.notes is [{text, textEncoding?, sortOrder?, clauses?: [...]}] — same clause shape as doors, but every matching note is shown, not just the first. A clause's questionKey/answerKey name a question/answer in this same definition, never a real id; an ANSWER_CHOSEN clause's answerKey must belong to its questionKey's question. All clauses in a condition must hold (AND only). definition.optionPoints (OUTCOME quizzes only) is [{answerKey, bucketKey, weight?}], both keys naming entries in this same definition. scales is only valid on a SCORE quiz; optionPoints is only valid on an OUTCOME quiz. The response returns quizId, publicUrl (the quiz's result/take link), and the real ids created for every key you chose: scaleIds, questionIds, answerIds, bucketIds (each a {yourKey: realId} map), plus doorIds and noteIds (plain lists, since doors and notes have no keys). omumu_quiz_get is this tool's natural counterpart: fetch an existing quiz with includeQuestions, includeScales, includeBuckets, includeDoors and includeNotes all true, and its output — with ids used in place of keys — is already shaped like a definition you could feed back into this tool (minus doors'/notes' clauses, which name questions/answers by id there and must be rekeyed). Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`definition`	object	yes	The whole quiz: {quiz, scales?, questions?, buckets?, doors?, notes?, optionPoints?}. See the tool description for the full shape of each.

`omumu_quiz_create`

Creates a new quiz (question set). The scoring mode decides how it's judged: OUTCOME (default) sorts the respondent into one of several named buckets by plurality; SCORE accumulates points per named scale (see

omumu_scale_create) and reports a percent per scale plus a total. Quizzes can optionally link to opt-in forms for email capture: preQuizOptinFormId shows an email capture form before the quiz starts (for non-logged-in users), postQuizOptinFormId shows one after quiz completion — that post-quiz form also submits itself as soon as the email field validates, before the respondent presses the final button (early contact capture), so the contact exists even if they never finish. After creating an outcome quiz, add questions with omumu_question_create, then add answer options with omumu_answer_create, create result buckets with omumu_bucket_create, and finally link answers to buckets using omumu_option_point_create. After creating a score quiz, create its scales with omumu_scale_create, then questions assigned to a scale, then answers carrying points. On a score quiz, noClearLowestText is shown on the result page when the respondent's scales are tied or there are fewer than two scales (no clear lowest); leave unset for no such section. noClearLowestText supports result-copy placeholders (see omumu_note_create for the full set). When askFirstName is true, an anonymous taker sees a first-name field on the start screen before the first question; the name is stored on the lead, copied to the user record on identification (never overwriting an existing name), and used for the [firstName] placeholder and the result page's greeting. completionTag, when set, is applied to an identified respondent when their result is saved, regardless of which bucket won — so an automation can tell a finisher apart from someone who only left an email (e.g. FORM_SUBMIT on the post-quiz form, then WAIT, then HAS_TAG on this tag to find abandoners). captureTag, when set, is applied to the respondent at early contact capture — the moment the post-quiz form's email is valid, before the final button — and is never removed at completion, so a CRM workflow or automation has a tag to fire on for someone who left their email but may not finish. A respondent who never reaches the form never gets it. Each answer can carry its own tag too (see omumu_answer_create).

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`askFirstName`	boolean	no	Ask an anonymous taker for their first name on the start screen before the first question (default false)
`captureTag`	string	no	Tag applied to the respondent at early contact capture, the moment the post-quiz form's email is valid; never removed at completion
`completionTag`	string	no	Tag applied to an identified respondent when their result is saved, regardless of which bucket won
`label`	string	yes	The quiz name/label
`noClearLowestText`	string	no	Score quizzes only: text shown when there is no clear lowest scale (tie or all-equal)
`noClearLowestTextEncoding`	string	no	How noClearLowestText is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`scoring`	string	no	Scoring mode: OUTCOME (default) or SCORE

`omumu_quiz_get`

Gets a quiz with optional questions, result buckets and scales. Always returns the scoring mode, askFirstName, completionTag, captureTag, destinations, highLevelResultLinkField (when set) and, for a score quiz, noClearLowestText/noClearLowestTextEncoding. Returns preQuizOptinFormId and postQuizOptinFormId if linked opt-in forms are configured for email capture. For an outcome quiz, set includeQuestions=true, includeBuckets=true, and includeOptionPoints=true to see the complete scoring configuration — option points link answers to buckets with weights, and the bucket with the highest total weight from selected answers wins. With includeQuestions=true each answer also returns its tag, if any. For a score quiz, set includeQuestions=true and includeScales=true — each question returns its scaleId and each answer

its points; each scale also returns its levels, lowestScaleText and lowestScaleTextEncoding; includeBuckets=true also returns each bucket's minPercent, descriptionEncoding and videoResourceId (its profile configuration). Result-copy placeholders, substituted at render time in this text: [firstName] (the respondent's first name), [answer:] (the chosen answer label(s), or the free text, for that question), [scale:] (that scale's percent, e.g. "72 %"), [highestScale] and [lowestScale] (scale names), [highestQuestion] (the header of the question with the highest-scoring answer). An unknown placeholder or one that resolves empty renders as nothing.

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

Parameter	Type	Required	Description
`includeBuckets`	boolean	no	Include result buckets
`includeDoors`	boolean	no	Include doors, each with its answer-condition clauses
`includeNotes`	boolean	no	Include notes, each with its answer-condition clauses (score quizzes only)
`includeOptionPoints`	boolean	no	Include option points (answer-to-bucket scoring weights, outcome quizzes only). Requires questions to be fetched.
`includeQuestions`	boolean	no	Include questions and answers
`includeScales`	boolean	no	Include scales, each with its levels and lowest-scale text (score quizzes only)
`quizId`	string	yes	The quiz ID

`omumu_quiz_list`

Lists all quizzes for the current site

- **Write category:** read-only
- **Access:** requires the `QUIZ_READ` permission

No parameters.

`omumu_quiz_update`

Updates a quiz's properties including label, scoring mode, linked opt-in forms for email capture, (score quizzes only) the no-clear-lowest text, the ask-first-name flag, the completion tag, the capture tag and the destinations. Pre-quiz opt-in forms appear before the quiz starts for non-logged-in users. Post-quiz opt-in forms appear after completion. Set to null to remove the link. destinations lists where, besides Omumu, the quiz delivers each identified respondent once their result is saved; the only destination is HIGHLEVEL, which upserts the respondent as a contact (first name, email, phone) in the site's HighLevel sub-account with the tags this quiz applied to them. Adding HIGHLEVEL is refused unless the site's plan includes CRM_DESTINATIONS and a Headmaster has connected HighLevel (omumu_site_context reports highLevelConnected); no tool reads or writes the connection itself. highLevelResultLinkField names a custom contact field in that sub-account (its field key, e.g. contact.quiz_result_link; the contact. prefix is dropped) that receives the respondent's result link at completion, written before the tags so a workflow fired by the completion tag can use it; it needs the HIGHLEVEL destination, and null removes it.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`askFirstName`	boolean	no	Ask an anonymous taker for their first name on the start screen before the first question
`captureTag`	string	no	Tag applied to the respondent at early contact capture, the moment the post-quiz form's email is valid, or null to remove
`completionTag`	string	no	Tag applied to an identified respondent when their result is saved, regardless of which bucket won, or null to remove
`destinations`	array	no	Where, besides Omumu, the quiz delivers each identified respondent: ["HIGHLEVEL"] or [] to stop. The quiz ends up with exactly these
`highLevelResultLinkField`	string	no	HighLevel custom contact field key (e.g. contact.quiz_result_link) that receives each respondent's result link at completion; needs the HIGHLEVEL destination; null to remove
`label`	string	no	New label
`noClearLowestText`	string	no	Score quizzes only: text shown when there is no clear lowest scale, or null to remove
`noClearLowestTextEncoding`	string	no	How noClearLowestText is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`postQuizOptinFormId`	string	no	ID of opt-in form to show after quiz completion (for non-logged-in users), or null to remove
`preQuizOptinFormId`	string	no	ID of opt-in form to show before quiz (for non-logged-in users), or null to remove
`quizId`	string	yes	Scoring method: OUTCOME or SCORE

`omumu_resource_delete`

Permanently deletes a SiteResource. Fails if the resource is currently referenced by any lesson, page, or other entity — remove those references first.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_DELETE` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	The resource ID to delete

`omumu_resource_get`

Returns the full metadata for a single SiteResource, including CDN URI, thumbnail, MIME type, file size, and video processing status (when applicable).

- **Write category:** read-only
- **Access:** requires the `RESOURCE_READ` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	The resource ID

`omumu_resource_list`

Lists uploaded files (videos, images, PDFs, etc.) on the current site. Optionally filter by a MIME type prefix (e.g. 'video/' or 'image/png') and paginate with limit/offset. Returns total count so callers can implement 'load more' paging.

- **Write category:** read-only
- **Access:** requires the `RESOURCE_READ` permission

Parameter	Type	Required	Description
`limit`	number	no	Maximum resources to return (default: 25, max: 100)
`mimeTypePrefix`	string	no	Optional prefix match against the resource's MIME type, e.g. 'video/' to return all videos or 'image/png' to return only PNG images. Omit to list all types.
`offset`	number	no	Number of resources to skip for pagination (default: 0)

`omumu_resource_update`

Updates the title and/or description of a SiteResource. Does not change the underlying file — re-upload for that. At least one of title/description must be provided.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	New description (optional)
`resourceId`	string	yes	The resource ID
`title`	string	no	New title (optional)

`omumu_resource_upload_presign`

Prepares the upload of any file type the site accepts (PDF, image, audio, text, office document, archive, video) without the bytes passing through the MCP channel. Check 'uploadProtocol' in the result. 'put': send exactly 'length' bytes of the file as the body of an HTTP PUT to 'uploadUrl' (no Authorization header; the URL is a single-use ticket that expires at 'expiresAt'). The response is 201 with the new resource (id, fileName, fileSize, uri); use its id with omumu_lesson_resource_create or omumu_set_image. A failed PUT needs a new presign. 'tus': a video; follow omumu_resource_upload_video_presign's instructions with the same fields (resourceId is usable right away). Uploads count against the site's per-file and total storage limits.

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`filename`	string	yes	Original file name including extension, e.g. 'worksheet.pdf'. Becomes the resource's file name.
`filetype`	string	yes	MIME type of the file, e.g. application/pdf, image/png, audio/mpeg, application/zip, video/mp4
`length`	number	yes	Exact file size in bytes; the upload is rejected if the body differs

`omumu_resource_upload_video_presign`

Creates a SiteResource for a video (videoProcessingStatus QUEUED) and returns a presigned TUS (tus.io) upload the caller PATCHes bytes to directly — the video never passes through the MCP server or the JSON-RPC channel. Use the returned resourceId right away with omumu_lesson_resource_create or omumu_set_image; poll omumu_resource_get for videoProcessingStatus once the upload finishes and Bunny's webhook reconciles it. To perform the upload: POST to the returned 'endpoint' with headers AuthorizationSignature, AuthorizationExpire, VideoId (the 'guid' field), LibraryId, Upload-Length (the file size), and Tus-Resumable: 1.0.0 — Bunny responds with a Location header, which is the URL to PATCH the file bytes to (Content-Type: application/offset+octet-stream, Upload-Offset: 0, Tus-Resumable: 1.0.0).

- **Write category:** standard write
- **Access:** requires the `RESOURCE_WRITE` permission

Parameter	Type	Required	Description
`filename`	string	yes	Original file name of the video, e.g. 'lesson-1.mp4'
`filetype`	string	yes	MIME type of the video: video/mp4, video/webm, video/ogg, video/quicktime, or video/x-matroska
`length`	number	yes	Total file size in bytes

`omumu_scale_create`

Creates a named scale on a score quiz (e.g. "Listening"). Assign questions to it with omumu_question_create/update's scaleId; the scale's maximum is derived from its questions' answer points, never set here. Optionally carries its levels inline and a lowest-scale text. Each level is a percent range with copy only (no tags, sequences or grants): the level with the highest minPercent at or below the respondent's percent on this scale is shown on its result card. lowestScaleText is shown once on the result page when this scale is the respondent's lowest by percent. Level text and lowestScaleText support result-copy placeholders (see omumu_note_create for the full set).

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`levels`	array	no	Inline levels: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]
`lowestScaleText`	string	no	Text shown when this scale is the respondent's lowest
`lowestScaleTextEncoding`	string	no	How lowestScaleText is encoded: PLAIN_TEXT (default), QUILL_JSON, or MANAGED_HTML
`name`	string	yes	The scale's name
`quizId`	string	yes	The quiz ID
`sortOrder`	number	no	Display order (defaults to last)

`omumu_scale_delete`

Deletes a scale. Questions assigned to it become unscored background questions rather than being deleted.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`scaleId`	string	yes	The scale ID to delete

`omumu_scale_update`

Updates a scale's name, sort order, levels and/or lowest-scale text. A levels array replaces the scale's levels wholesale rather than appending to them — include every level you want kept.

- **Write category:** standard write
- **Access:** requires the `QUIZ_WRITE` permission

Parameter	Type	Required	Description
`levels`	array	no	Replaces this scale's levels wholesale: [{name, minPercent (0-100), text?, textEncoding?, sortOrder?}]
`lowestScaleText`	string	no	New lowest-scale text, or null to remove
`lowestScaleTextEncoding`	string	no	How lowestScaleText is encoded: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`name`	string	no	New name
`scaleId`	string	yes	The scale ID
`sortOrder`	number	no	New display order

`omumu_set_image`

Attach an existing image resource to a course, module, or lesson. Use this to reuse an already generated image or to change an entity's image.

- **Write category:** standard write
- **Access:** requires the `COURSE_WRITE` permission

Parameter	Type	Required	Description
`resourceId`	string	yes	ID of the image resource to attach
`targetId`	string	yes	ID of the entity to set the image on
`targetType`	string	yes	Entity type: 'course', 'module', or 'lesson'

Selling

`omumu_offer_add_bump`

Adds a bump offer to an existing offer. The bump appears as a checkbox on the checkout page. Both offers must belong to the same site. An offer cannot be its own bump.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`bumpOfferId`	string	yes	The bump offer ID to add
`offerId`	string	yes	The parent offer ID

`omumu_offer_add_product`

Adds a product (course or skill) to an existing offer. The offer must not be live or ended. Subscription plans (SAAS) are added via omumu_offer_add_subscription_plan instead.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID
`productId`	string	yes	The product ID to add — a course id or a skill id (the meta_product id, which equals the underlying course/skill id)

`omumu_offer_add_subscription_plan`

Adds a subscription plan (Omumu platform access) to an existing offer. When the offer is purchased, the buyer gets a new Omumu site provisioned on the plan's tier. Use SAAS_BASIC for the Omumu Basic plan.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID
`subscriptionPlanId`	string	yes	The subscription plan ID (e.g., SAAS_BASIC)

`omumu_offer_create`

Creates a purchasable offer with title, price, courses, and payment providers. Setting validFrom to now or a past time publishes the offer. status is 'live' only when checkout takes orders; otherwise 'paused' with pausedReason. A payment provider the site hasn't connected is refused (Invoice always works). If the sales page still holds unrewritten placeholders, a validFrom is dropped rather than failing the creation: the offer comes back not_published with salesPagePlaceholders filled and a matching warnings message — rewrite the page with omumu_page_update, then publish with omumu_offer_update. Returns the offer with its checkout URL.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`buttonText`	string	no	Text for the purchase button (e.g., 'Buy now')
`courseIds`	array	no	Array of course ID strings to include in the offer
`currency`	string	no	Three-letter currency code (default: NOK)
`descriptionPageId`	string	no	Page ID for the offer description shown at checkout
`invoiceButtonText`	string	no	Text for the invoice request button at checkout, in the seller's words (at most 255 characters). Leave it out for the default wording in the site's language
`paymentProviders`	array	no	Array of payment provider names: STRIPE, VIPPS, INVOICE_REQUEST. For a free offer, leave paymentProviders out
`subscriptionPlanId`	string	no	Subscription plan ID to include (e.g., SAAS_BASIC for Omumu Basic). Bundles Omumu platform access into the offer.
`tagline`	string	no	A short tagline or subtitle
`thankYouPageId`	string	no	Page ID for the post-purchase thank-you page
`title`	string	yes	The offer title
`totalAmountCents`	number	yes	Total price in cents (e.g., 4900 for 49.00 NOK)
`validFrom`	string	no	Start date/time in ISO-8601 format (e.g., 2026-04-01T00:00:00)
`validTo`	string	no	End date/time in ISO-8601 format
`vatAmountCents`	number	no	VAT amount in cents (included in totalAmountCents)

`omumu_offer_delete`

Deletes an offer. Only offers that have not been sold can be deleted.

- **Write category:** standard write
- **Access:** requires the `OFFER_DELETE` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID to delete

`omumu_offer_dna_architect_brief`

Returns the complete offer-architecture task for a DNA — the method rules, the embedded research verbatim, interview context, the format-intent constraint when declared, the freshly-resolved fenced existing material, and the exact output schema. Pure read: no AI call, no budget interaction. Run your own architecture using the instructions and context, then submit with omumu_offer_dna_architect_submit. Step order: omumu_offer_dna_research_submit → omumu_offer_dna_architect_brief → omumu_offer_dna_architect_submit → omumu_offer_create and omumu_page_create.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_architect_submit`

Validates and stores an offer architecture the calling agent produced itself — no AI budget is spent. The proposal object must have architecture, compression, and score matching the schema returned by omumu_offer_dna_architect_brief. Degenerate, empty (no solutions and no stack), or malformed submissions are rejected with a descriptive error; nothing is stored on failure. Re-submitting replaces all three prior sections. Next step: create the offer with omumu_offer_create and its sales page with omumu_page_create, using the DNA's architecture.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id
`proposal`	object	yes	The proposal object (architecture, compression, score) matching the schema from omumu_offer_dna_architect_brief

`omumu_offer_dna_create`

Starts a research-based offer document (ADR-0034) from a creator interview. Returns its id; follow with omumu_offer_dna_research_brief.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`audience`	string	yes	Who it is for — the more specific the better
`container`	string	no	The intended size of the thing (format intent, not a price bucket): TINY (a single video or one short lesson), NORMAL (a few lessons up to modest modules), FLAGSHIP (a multi-module program), or UNDECIDED (the architect infers the scale — the default)
`idea`	string	yes	What the creator wants to sell, in a sentence or two
`knownPains`	string	no	Pains the creator already suspects (the research verifies or challenges them)
`priceIdea`	string	no	The creator's price idea, if any
`scopeNotes`	string	no	Free-text medium/cadence/delivery notes, e.g. "one 45-minute screencast" or "6 modules over 6 weeks"
`seedCourseId`	string	no	An existing Course on this site to build on — the architect reads its module and lesson titles as the real deliverables
`seedOutline`	string	no	Pasted outline or lesson material drafted outside Omumu — the offer is architected around it instead of an invented curriculum

`omumu_offer_dna_get`

Returns the full Offer DNA document: interview, cited research, architecture, compression (touchstone + big four), and score.

- **Write category:** read-only

- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_list`

Lists the site's Offer DNA documents, newest first, with their completion state.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

No parameters.

`omumu_offer_dna_research_brief`

Returns the complete voice-of-customer research task for a DNA — the method rules, interview context and the exact output schema the submission must match. Pure read: no AI call, no budget interaction. Run your own web research using the instructions and context, then submit with omumu_offer_dna_research_submit. Next step after submitting: omumu_offer_dna_architect_brief.

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id

`omumu_offer_dna_research_submit`

Validates and stores research the calling agent ran itself — no AI budget is spent. The research object must match the schema returned by omumu_offer_dna_research_brief. Degenerate, empty-findings, or malformed submissions are rejected with a descriptive error; nothing is stored on failure. Re-submitting replaces the research section while leaving any existing architecture untouched. Next step: omumu_offer_dna_architect_brief.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`id`	string	yes	The Offer DNA id
`research`	object	yes	The research object matching the schema from omumu_offer_dna_research_brie

`omumu_offer_get`

Gets an offer by ID with full details including checkout URL, status (with pausedReason when paused), products, pricing, payment providers, and salesPagePlaceholders (unrewritten starter prompts still on the sales page, each with section, field, and excerpt; empty when none remain or the offer has no sales page)

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

Parameter	Type	Required	Description
`offerId`	string	yes	The offer ID

`omumu_offer_grant_access`

Grants a user access to all products in an offer WITHOUT requiring payment. This bypasses the normal checkout/payment flow entirely. The user receives the same access as if they had paid: STUDENT role, product access entries, welcome email, and offer-tag execution. An offer containing a SAAS plan can only be granted by Omumu. The user must already exist on the same site as the offer.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`email`	string	yes	Email of the user to grant access to. Must already exist on the same site as the offer
`message`	string	no	Optional free-text reason for the grant (e.g. 'Demo account for Ole-Arvid'). Recorded in the audit log only, not stored on the access entry. Max 500 characters
`offerId`	string	yes	The offer to grant access to

`omumu_offer_list`

Lists all offers for the current site with their status (not_published/scheduled/live/paused/ended, with pausedReason when paused), products, and pricing

- **Write category:** read-only
- **Access:** requires the `OFFER_READ` permission

No parameters.

`omumu_offer_remove_bump`

Removes a bump offer from an existing offer.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`bumpOfferId`	string	yes	The bump offer ID to remove
`offerId`	string	yes	The parent offer ID

`omumu_offer_update`

Updates an offer's title, tagline, buttonText, invoiceButtonText, price, payment providers, or validity dates. Only provided fields are updated; others are preserved. Setting validFrom to now or a past time publishes the offer. status is 'live' only when checkout takes orders; otherwise 'paused' with pausedReason. A payment provider the site hasn't connected is refused (Invoice always works). Setting validFrom while the sales page still holds unrewritten place-

holders fails, listing each one with where to fix it — rewrite the page with omumu_page_update, then retry.

- **Write category:** standard write
- **Access:** requires the `OFFER_WRITE` permission

Parameter	Type	Required	Description
`buttonText`	string	no	New button text
`currency`	string	no	New currency code
`invoiceButtonText`	string	no	New text for the invoice request button at checkout (at most 255 characters). An empty string brings back the default wording
`offerId`	string	yes	The offer ID
`paymentProviders`	array	no	Array of payment provider names: STRIPE, VIPPS, INVOICE_REQUEST. Replaces the offer's current providers
`tagline`	string	no	New tagline
`title`	string	no	New title
`totalAmountCents`	number	no	New total price in cents
`validFrom`	string	no	New start date/time in ISO-8601 format
`validTo`	string	no	New end date/time in ISO-8601 format
`vatAmountCents`	number	no	New VAT amount in cents

Pages & Sales Pages

`omumu_evergreen_note_create`

Writes a new Evergreen note in the site's Notes section (/notes): a short public note on one idea, without a date, that keeps growing and links to other notes. These are not the quiz notes (omumu_note_*) and not pages. Evergreen notes are public the moment they are saved: there is no draft, and the result's publicUrl is where visitors read the note. In the body, [[Title]] names another

note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved. A title another note on the site already has, ignoring case, is refused and nothing is written.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`body`	string	no	The note's body: Markdown (headings, lists, quotes, code, bold, italic, links, images) or Quill Delta JSON, detected the way the page tools detect it. Use [[Title]] to link to another note.
`image`	string	no	Optional sharing image URL for link previews
`metaDescription`	string	no	Optional description for search results and link previews
`title`	string	yes	The note's title, unique among the site's notes ignoring case; at most 255 characters, with no square brackets

`omumu_evergreen_note_delete`

Deletes an Evergreen note; its URL then answers 404. Every link to it in the site's other notes turns back into the Stub [[Its last title]], so writing a note with that title again links them once more.

- **Write category:** standard write
- **Access:** requires the `PAGE_DELETE` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note's id

`omumu_evergreen_note_get`

Gets an Evergreen note: its title, publicUrl, metaDescription, image, and its body as Markdown in which every link to another note is [[Its current title]] and every Stub is [[Title]], ready to edit and pass back to omumu_evergreen_note_update. Markdown leaves out embedded videos, and sending it back puts an empty line between paragraphs that had none and reads Markdown characters the text holds as formatting: a line starting with '-', '#', '>' or a number and a dot becomes a list, heading or quote, and text between asterisks or underscores is emphasised; 'quillBody' is the stored Quill JSON, which keeps all of it. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`noteId`	string	yes	The note's id

`omumu_evergreen_note_list`

Lists the site's Evergreen notes, the most recently grown first (id, title, publicUrl), and its Stubs: the [[Title]] titles its notes name that no note has yet, each with how many notes name it, the most-named first. A much-named Stub is a good idea to write next. In the body, [[Title]] names another note: saving turns it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

No parameters.

`omumu_evergreen_note_update`

Grows or renames an Evergreen note; any parameter left out stays as it is. Renaming never breaks a link to the note. Evergreen notes are public the moment they are saved: there is no draft, and the result's publicUrl is where visitors read the note. In the body, [[Title]] names another note: saving turns

it into a link to the note with that title (matched ignoring case), which keeps working when that note is renamed. A [[Title]] no note has yet stays a Stub, an idea to write next, and becomes a link once a note with that title is saved. A title another note on the site already has, ignoring case, is refused and nothing is written.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`body`	string	no	The new body, replacing the old one. The note's body: Markdown (headings, lists, quotes, code, bold, italic, links, images) or Quill Delta JSON, detected the way the page tools detect it. Use [[Title]] to link to another note.
`image`	string	no	New sharing image URL; an empty string removes it
`metaDescription`	string	no	New description for search results; an empty string removes it
`noteId`	string	yes	The note's id
`title`	string	no	The new title, at most 255 characters, with no square brackets

`omumu_page_create`

Creates a new page. The page is live on the public site immediately — there is no draft step — and the result's publicUrl is the address it is served at. Two modes:

Slot page (recommended for sales pages): Pass 'template' (discriminator from omumu_template_list) and 'slotJson' (whole-document slot JSON validated against that template). For SALES templates, also pass 'offerId'. Wiring fields (offerId, purpose) must be tool parameters — never inside slot contents.

Classic page: Pass ‘contents’ with optional ‘encoding’ (QUILL_JSON or MANAGED_HTML). MANAGED_HTML is available but de-emphasised for selling pages — prefer slot pages instead.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`contents`	string	no	Classic page content. Raw HTML when encoding is MANAGED_HTML, Quill JSON or markdown otherwise. Not used when creating slot pages.
`encoding`	string	no	Content encoding for classic pages: QUILL_JSON (default) or MANAGED_HTML. Not used when creating slot pages.
`offerId`	string	no	Offer ID this sales page sells. Required for SALES templates; must be a valid offer belonging to this site. Never pass inside slot contents.
`slotJson`	string	no	Whole-document slot JSON validated against the named template. Required when 'template' is provided.
`slug`	string	yes	The page URL slug (e.g., 'my-course-launch')
`template`	string	no	Template discriminator from omumu_template_list (e.g. 'sales-cold-traffic'). Required for slot page creation; causes slotJson validation against the named template.
`title`	string	no	The page title
`type`	string	no	Page type: blog, frontpage, userpage, terms, privacy, cookies, template, block, offer, thank_you

`omumu_page_delete`

Deletes a page by ID

- **Write category:** standard write
- **Access:** requires the `PAGE_DELETE` permission

Parameter	Type	Required	Description
`pageId`	string	yes	The page ID to delete

`omumu_page_get`

Gets a page by ID or slug. At least one of pageId or slug must be provided. The result's 'public' field is false for a sales page whose offer is not published (visitors get a 404 there until it is), true otherwise.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`pageId`	string	no	The page ID (provide this or slug)
`slug`	string	no	The page slug (provide this or pageId)

`omumu_page_list`

Lists all pages for the current site, optionally filtered by type

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

Parameter	Type	Required	Description
`excludeOfferPages`	boolean	no	Exclude auto-generated offer pages
`type`	string	no	Filter by page type: blog, frontpage, userpage, terms, privacy, cookies, template, block, offer, thank_you

`omumu_page_update`

Updates a page. The change is live on the public site immediately — there is no draft step — and the result's publicUrl is the address the page is served at. For slot pages (encoding SLOT_JSON), pass 'slotJson' with the full updated slot document — wiring (offerId, purpose) is immutable on update and is not re-validated here. For classic pages, pass any combination of 'title', 'tagline', 'metaDescription', 'contents', 'image', or 'slug'.

- **Write category:** standard write
- **Access:** requires the `PAGE_WRITE` permission

Parameter	Type	Required	Description
`contents`	string	no	New content for classic pages. Format depends on page encoding: raw HTML for MANAGED_HTML pages, Quill JSON for QUILL_JSON pages.
`image`	string	no	New header image URL
`metaDescription`	string	no	New meta description for SEO
`pageId`	string	yes	The page ID
`rootPublished`	boolean	no	Serve the page at the site root (/<slug>) in addition to /page/<slug>. Reserved platform names (admin, terms, courses, …) are refused. Both addresses keep working; rendered links, canonical metadata, and the sitemap use the root URL. Omit to leave unchanged.
`slotJson`	string	no	Full updated slot JSON document for slot pages (encoding SLOT_JSON). The template discriminator is read from the existing slot document. Wiring is immutable.
`slug`	string	no	New URL slug (changes the page URL)
`tagline`	string	no	New tagline
`title`	string	no	New title

`omumu_template_list`

Returns all registered page templates with their section layouts and per-section JSON schema. Call this tool first when building a slot page so you know which template discriminator to pass to omumu_page_create and what slot

fields each section expects. Each entry includes: 'name' (the discriminator), 'purpose' (e.g. SALES), 'sections', and 'schema'.

- **Write category:** read-only
- **Access:** requires the `PAGE_READ` permission

No parameters.

Audience & Email

`omumu_email_followup_update`

Updates a follow-up message in an email sequence. All fields except followUpId are optional.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON
`delay`	string	no	New delay from the first email (absolute, NOT relative to previous). Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`followUpId`	string	yes	The follow-up ID to update
`subject`	string	no	New subject line

`omumu_email_inbox_get`

Retrieves full details of a specific email including headers, body content (text and/or HTML), attachments, and metadata. Use omumu_email_inbox_list first to find email IDs.

- **Write category:** read-only
- **Access:** requires the `EMAIL_INBOX_READ` permission

Parameter	Type	Required	Description
`emailId`	string	yes	The email ID to retrieve

`omumu_email_inbox_list`

Lists received emails in the inbox with optional filtering by status or search query. Returns email summaries (id, sender, subject, status, date) with pagination. Also returns unread count for quick monitoring.

- **Write category:** read-only
- **Access:** requires the `EMAIL_INBOX_READ` permission

Parameter	Type	Required	Description
`limit`	number	no	Maximum emails to return per page (default: 25, max: 100)
`page`	number	no	Page number for pagination (default: 1)
`search`	string	no	Search in sender address, sender name, subject, or recipient address
`status`	string	no	Filter by status: UNREAD, READ, ARCHIVED, HELD, or ALL (default: the inbox, which excludes deleted and held email)

`omumu_email_sequence_create`

Creates a new email sequence with an initial email and optional follow-up messages. Content can be Markdown or QuillDelta JSON (auto-detected). After creating, link it to an opt-in form via omumu_optinform_update(emailSequenceId) to trigger the sequence when users submit the form. To send a single one-off email to a contact from an automation, create a sequence containing just one message with initialDelay PT0S and start it with the automation's start_sequence action. Subject and content may use $first_name (the recipient's first name) and $quiz_result_link (the recipient's newest quiz result page on this site, for a sequence a quiz's result bucket starts); a variable that cannot be resolved renders as nothing.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	yes	Email content in Markdown or QuillDelta JSON
`followUps`	array	no	Array of follow-up messages with delay (offset from the first email, NOT relative to previous), subject, and content. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`initialDelay`	string	no	Delay before sending the first email. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w). Default: PT0S.
`name`	string	yes	Sequence name (unique identifier)
`subject`	string	yes	Subject line for the initial email

`omumu_email_sequence_get`

Retrieves a specific email sequence with all follow-up messages, plus memberCount, sentCount and heldCount. heldCount counts emails not delivered because the site has no sequence email service configured yet. No member identities are included.

- **Write category:** read-only
- **Access:** requires the `EMAIL_SEQUENCE_READ` permission

Parameter	Type	Required	Description
`sequenceId`	string	yes	The sequence ID to retrieve

`omumu_email_sequence_list`

Lists all email sequences for the current site with their follow-up messages and memberCount (how many people are enrolled). No member identities are included.

- **Write category:** read-only
- **Access:** requires the `EMAIL_SEQUENCE_READ` permission

No parameters.

`omumu_email_sequence_update`

Updates an existing email sequence's properties. All fields except sequenceId are optional.

- **Write category:** standard write
- **Access:** requires the `EMAIL_SEQUENCE_WRITE` permission

Parameter	Type	Required	Description
`content`	string	no	New content in Markdown or QuillDelta JSON
`initialDelay`	string	no	New delay from the subscription trigger. Duration string. Accepts ISO-8601 (e.g. PT4H, P1D, P2DT12H, PT0S) or shorthand <integer><m|h|d|w> (e.g. 30m, 4h, 7d, 2w).
`name`	string	no	New sequence name
`sequenceId`	string	yes	The sequence ID to update
`subject`	string	no	New subject line for initial email

`omumu_invite_get`

Gets a single Invite by ID with full config, stage (draft, live, or paused), window state (accepted of quota, next boundary), waitlist size, and placeholders (unrewritten starter prompts still on the invite page or in its delivery, waitlist-welcome, or reopen emails, each with section, field, and excerpt).

- **Write category:** read-only
- **Access:** requires the `INVITE_READ` permission

Parameter	Type	Required	Description
`inviteId`	string	yes	The Invite ID

`omumu_invite_launch`

Creates a complete Honest Scarcity Invite in one call: a course (or attaches an existing one), three seeded email sequences (delivery, waitlist-welcome, reopen), an opt-in form wired to the course grant and delivery sequence, the Invite itself, the waitlist tag action on the reopen sequence, and the invite page — all wired, but created as a draft: its page answers visitors with a 404 and its form refuses submissions until you take it live. Returns status 'draft', the page URL, the slot editor path, and placeholders (unrewritten starter prompts still on the page or in its emails, each with section, field, and excerpt) so you can customize copy with omumu_page_update and omumu_email_sequence_update before going live with omumu_invite_update (active: true); going live is refused while any remain. Cadence values: HOUR, DAY, WEEK, MONTH. openingDayOfWeek required for WEEK cadence (e.g. MONDAY). openingDayOfMonth required for MONTH cadence (1-31). openingTime format: HH:MM (24h); defaults to 09:00. zone: IANA timezone id (e.g. 'Europe/Oslo'); defaults to Europe/Oslo. seedLanguage: 'nb' for Norwegian Bokmål, 'en' (or omit) for English.

- **Write category:** standard write
- **Access:** requires the `INVITE_WRITE` permission

Parameter	Type	Required	Description
`cadence`	string	yes	Reset cadence: HOUR, DAY, WEEK, or MONTH
`courseId`	string	no	Existing course ID to attach; omit to create a new empty course
`name`	string	yes	Human-readable name for the Invite (also used as the page headline)
`openingDayOfMonth`	number	no	Day of month for MONTH cadence (1-31)
`openingDayOfWeek`	string	no	Day of week for WEEK cadence (e.g. MONDAY)
`openingTime`	string	no	Opening time in HH:MM (24h); defaults to 09:00
`quota`	number	yes	Slots available per window
`seedLanguage`	string	no	'nb' for Norwegian Bokmål, 'en' or omit for English
`zone`	string	no	IANA timezone id (e.g. 'Europe/Oslo'); defaults to Europe/Oslo

`omumu_invite_list`

Lists all Invites for the current site with their stage (draft, live, or paused), window state (accepted of quota, next boundary), and waitlist size.

- **Write category:** read-only
- **Access:** requires the `INVITE_READ` permission

No parameters.

`omumu_invite_update`

Updates an Invite's name, quota, schedule (cadence, opening moment, zone), or active flag. Changing cadence, openingDayOfWeek/openingDayOfMonth, openingTime, or zone recomputes the next window boundary. Raising or lowering quota takes effect immediately in the current window without invalidating existing acceptances. active=true takes a draft or paused Invite live (its page and form start working for visitors), but is refused while the invite page or its delivery, waitlist-welcome, or reopen emails still hold

unrewritten placeholders — the error lists each one with where to fix it; rewrite with omumu_page_update and omumu_email_sequence_update, then retry. active=false pauses a live Invite (its page 404s for visitors, its form refuses submissions) and lifts any deletion guards on its form and sequences; deactivating is never refused. Cadence values: HOUR, DAY, WEEK, MONTH. Day-of-week values: MONDAY..SUNDAY. openingTime format: HH:MM (24h). zone: IANA timezone id (e.g. 'Europe/Oslo').

- **Write category:** standard write
- **Access:** requires the `INVITE_WRITE` permission

Parameter	Type	Required	Description
`active`	boolean	no	Whether the Invite is active; deactivating lifts deletion guards on its form and sequences
`cadence`	string	no	Reset cadence: HOUR, DAY, WEEK, or MONTH
`inviteId`	string	yes	The Invite ID
`name`	string	no	Human-readable name for the Invite
`openingDayOfMonth`	number	no	Day of month for MONTH cadence (1-31)
`openingDayOfWeek`	string	no	Day of week for WEEK cadence (e.g. MONDAY)
`openingTime`	string	no	Opening time in HH:MM for DAY/WEEK/MONTH cadences
`quota`	number	no	Slots available per window
`zone`	string	no	IANA timezone id (e.g. 'Europe/Oslo')

`omumu_optinform_add_field`

Adds a new field to collect user information

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`fieldLabel`	string	no	Display label (e.g., 'Email Address')
`fieldName`	string	yes	Internal field name (e.g., 'email', 'firstName')
`fieldType`	string	no	Field type: EMAIL, TEXT, PHONE, or HIDDEN
`formId`	string	yes	The form ID
`placeholder`	string	no	Placeholder text for the input
`required`	boolean	no	Whether the field is required
`sortOrder`	integer	no	Position in the form (0 = first)

`omumu_optinform_create`

Creates a new opt-in form for collecting leads. The form comes with default email and name fields. After creating, use omumu_optinform_update to connect it to an email sequence (emailSequenceId) and/or course (courseId). Embed on a page with [optin-form id="FORM_ID"] on its own line (Markdown, Quill or HTML pages). For templated (SLOT_JSON) pages set the page's optinFormId instead. Forms can also be linked to quizzes as pre-quiz or post-quiz email capture.

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Description or subtitle. Supports rich text when descriptionEncoding is QUILL_JSON.
`name`	string	yes	Internal name for the form (admin use)
`title`	string	no	Display title shown to users

`omumu_optinform_delete`

Deletes an opt-in form and all its submissions

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID to delete

`omumu_optinform_get`

Gets an opt-in form by ID with all its fields. Returns descriptionEncoding (PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML) and trustLine if configured.

- **Write category:** read-only
- **Access:** requires the `OPTINFORM_READ` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID

`omumu_optinform_link_sequence`

Connects an opt-in form to an email sequence so submissions trigger subscription

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`emailSequenceId`	string	yes	The email sequence ID to link
`formId`	string	yes	The form ID

`omumu_optinform_list`

Lists all opt-in forms for the current site

- **Write category:** read-only
- **Access:** requires the `OPTINFORM_READ` permission

No parameters.

`omumu_optinform_remove_field`

Removes a field from a form by its field ID

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`fieldId`	string	yes	The field ID to remove
`formId`	string	yes	The form ID

`omumu_optinform_submissions_list`

Lists submissions for an opt-in form, newest first. Each entry contains the submission ID, timestamp, and the full submitted data (all field values). Returns respondent personal data, so it requires the dedicated leads:read scope (not granted by optinform:read).

- **Write category:** read-only
- **Access:** requires the `LEADS_READ` permission

Parameter	Type	Required	Description
`formId`	string	yes	The form ID
`limit`	number	no	Maximum number of submissions to return (default 50, max 500)

`omumu_optinform_update`

Updates an opt-in form's properties. Use this to connect the form to a course and email sequence, configure the submit button, and set up redirect behavior. Typical funnel setup: create sequence -> create form -> update form with emailSequenceId + courseId + redirectUrl -> embed in page.

- **Write category:** standard write
- **Access:** requires the `OPTINFORM_WRITE` permission

Parameter	Type	Required	Description
`courseId`	string	no	Course to grant access to — creates a user account and enrolls them on form submission
`description`	string	no	Description. Supports rich text when descriptionEncoding is QUILL_JSON.
`descriptionEncoding`	string	no	Encoding for description: PLAIN_TEXT, QUILL_JSON, or MANAGED_HTML
`emailSequenceId`	string	no	Email sequence to subscribe users to — triggers the sequence on form submission
`formId`	string	yes	The form ID
`name`	string	no	Internal name
`redirectUrl`	string	no	URL to redirect after submission
`submitButton`	string	no	Submit button text
`successMessage`	string	no	Message shown after successful submission
`title`	string	no	Display title
`trustLine`	string	no	Small muted text shown below submit button (e.g., 'No spam. Unsubscribe anytime.')

Automations

`omumu_automation_activate`

Activates or deactivates an automation. Active automations will trigger when their conditions are met. Before activating, run automation_simulate for each contact journey that matters and resolve any timing-trap warnings from the save/activate response — activation is the last step of the authoring loop (save, check warnings, simulate, activate).

- **Write category:** standard write

- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`active`	boolean	no	true to activate, false to deactivate (default: true)
`automationId`	string	yes	The automation ID

`omumu_automation_create`

Creates a new automation (inactive by default). Recommended authoring loop: 1) automation_save the nodes and connections, 2) read any timing-trap warnings in the save response, 3) automation_simulate the contact journeys that matter (especially 'contact does X, then Y during a delay'), 4) only then automation_activate.

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`description`	string	no	Description of what this automation does
`name`	string	yes	The automation name

`omumu_automation_generate_ids`

Generates unique IDs for use when creating new nodes and connections in canvas state

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`count`	integer	no	Number of IDs to generate (default: 5, max: 50)

`omumu_automation_get`

Gets an automation with its full canvas state (nodes, connections, configs). Use this to inspect or modify an existing automation.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID

`omumu_automation_list`

Lists all automations for the current site with their status

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

No parameters.

`omumu_automation_options`

Returns all available sequences, courses (with lessons), tags, forms, offers, quizzes, staff recipients, and supported node types. Use this to find valid IDs when building automation configs. Each supported node type carries a settings array describing its config keys: key, required, type, allowed values (when enumerated), and refersTo (which list above an id-valued key must be found in, e.g. "sequences" or "courses[].modules"). quizzes lists the site's quizzes (id + name) for the quiz_completed trigger. staffRecipients lists who may receive notify_site_staff emails (name + email); defaultRecipient is the authoring user's email. send_webhook's settings document its url (https, public address, re-checked at send time) and optional secret (signs the body with X-Omumu-Signature: sha256=); see omumu_automation_save's description for the posted JSON shape. There is no send_email node: to send a single email to a contact, create an email sequence containing one message with zero initial delay and start it with the start_sequence action.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

No parameters.

`omumu_automation_save`

Saves the full canvas state (nodes + connections) for an automation. Saving takes effect immediately, including on an already-active automation — there is no need to deactivate first. Canvas state format: {"nodes": [{"id": "…", "type": "tag_applied", "x": 0, "y": 0, "config": {…}}], "connections": [{"id": "…", "from": "nodeId", "to": "nodeId", "type": "default"}]}. Use automation_generate_ids to get IDs for new nodes/connections. Use automation_options to get valid IDs for sequences, courses, tags, forms, offers, quizzes, staff recipients — and, for every node type, its settings (config keys, required or optional, allowed values,

and what catalog list an id refers to). Completion triggers (module_completed, course_completed) are side-effects only (celebrate, tag, follow up) — on a SEQUENTIAL course the next module unlocks by itself, so do not wire module_completed to unlock_module for gating. Pair submission_received with unlock_module to open the next module when a student posts their work. Pair quiz_completed (or form_submit) with notify_site_staff to email a staff member a digest of every answer, the bucket, and the score. send_webhook posts the run data as JSON to a creator-controlled URL ({“event”, “siteId”, “occurredAt”, “runId”, “contact”: {“userId”,“email”,“firstName”,“lastName”,“phone”} when identified, “input”: the trigger’s run input, and for a quiz-triggered run “quiz”: {“quizId”,“label”,“bucketName”,“totalPercent”,“scalePercents”,“doorName”,“resultUrl”,“answers”:[{“question”,“answers”|“text”}]}}); config url must be https and resolve to a public address (checked again at send time), config secret is optional and signs the body as header X-Omumu-Signature: sha256=; a non-2xx response, a timeout or a refused connection is recorded as a FAILED step and the automation continues. There is no send_email node: to send a single email to a contact, create an email sequence containing one message with zero initial delay (omumu_email_sequence_create, initialDelay PT0S) and start it with the start_sequence action. The response includes timing-trap warnings (e.g. stop-sequence-before-delayed-start: a STOP_SEQUENCE may fire before a WAIT-delayed START_SEQUENCE and silently no-op; fix by starting the sequence immediately and moving the delay into the sequence’s initialDelay). Warnings are non-blocking, but do not activate past one without first proving the funnel correct with automation_simulate.

- **Write category:** standard write
- **Access:** requires the `AUTOMATION_WRITE` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID
`canvasState`	object	yes	The full canvas state JSON with nodes and connections

`omumu_automation_simulate`

Runs a scenario against the site’s automations in memory (no emails sent, nothing written) and returns the effect timeline plus any timing-trap warnings. All of the site’s automations run together so cross-flow races (e.g. a purchase stopping a delayed abandonment sequence) are visible. ‘events’ is an

array of {“at”: duration, “trigger”: triggerType, “contact”: alias}: ‘at’ is a duration from t0 as ISO-8601 (PT5M) or a friendly form (0s, 5m, 1h, 2d, 1w); ‘trigger’ is a contact trigger type (form_submit, purchase, tag_applied, page_visit); ‘contact’ is an optional alias (default ‘contact’) so several events can describe one contact’s journey. Each timeline entry is {atSeconds, at, contact, kind, target, effective}: kind is APPLY_TAG, REMOVE_TAG, START_SEQUENCE, STOP_SEQUENCE, EMAIL_SENT, ENROLL_COURSE, UNLOCK_MODULE, GRANT_ROLE or NODE_REJECTED; ‘effective’ is false for a no-op (e.g. a stop with no live subscription). Use this to verify a funnel behaves before activating it: don’t just look for the expected effects — scan the timeline for effective=false on stops/removals and for EMAIL_SENT entries the contact should NOT receive (an ineffective STOP_SEQUENCE followed by a later EMAIL_SENT is the classic cart-abandonment race). Simulating is step 3 of the authoring loop: save, check warnings, simulate the journeys that matter, then activate.

- **Write category:** read-only
- **Access:** requires the `AUTOMATION_READ` permission

Parameter	Type	Required	Description
`automationId`	string	yes	The automation ID to simulate (its site’s automations all run together)
`events`	array	yes	Ordered scenario events: [{“at”: “5m”, “trigger”: “purchase”, “contact”: “buyer”}]

Podcasting

`omumu_podcast_episode_create`

Creates a draft episode on the site’s show from an already-uploaded audio resource (upload the file first with omumu_resource_upload_presign; mp3, m4a, and wav are accepted). Mastering starts automatically and the episode is returned in PROCESSING status; publish it right away and it goes live when mastering ends, or poll omumu_podcast_episode_get until it reaches READY, then publish. With deferMastering the episode is returned UNMASTERED and nothing is mastered yet: start mastering with omumu_podcast_episode_master, or publish it and it is mastered first, then goes live.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`audioResourceId`	string	yes	ID of the uploaded audio resource to master into this episode
`deferMastering`	boolean	no	Leave the episode UNMASTERED instead of mastering it now (default false), so no mastering credits are spent on a recording that will be trimmed or split first. Mastering is deferred, never skipped: an UNMASTERED episode is mastered before it can go live
`episodeNumber`	number	no	Optional episode number within the season
`episodeType`	string	no	Episode type: 'full' (default), 'trailer', or 'bonus'
`explicit`	boolean	no	Per-episode explicit override; omit to inherit the show's flag
`noiseReduction`	boolean	no	Remove background noise when mastering (default false). Helps a recording made outdoors or in a noisy room, but can make a clean recording sound harsh. Applied only when the site masters with Auphonic
`season`	number	no	Optional season number
`showNotes`	string	no	Show notes describing the episode
`title`	string	yes	The episode title

`omumu_podcast_episode_get`

Gets one episode by id with its processing status (UNMASTERED, PROCESSING, READY, FAILED, PUBLISHED, SPLITTING while being split into other episodes, SPLIT once it has been), duration, error message when failed, pubDate when published, the latest re-master's status (NONE, PENDING, FAILED) with its error when failed, masteringMethod: what made the current master (auphonic, omumu_chain, or null when not mastered yet or mastered by an older Omumu chain), masteringFallback: why Auphonic could not master it when Omumu's chain made it instead (out_of_credits, key_rejected, production_failed, unreachable, timed_out), null when it did not fall back, noiseReduction: whether the episode is mastered with noise reduction, and publishIntent: whether it publishes itself when mastering ends.

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_list`

Lists the show's episodes with their processing status (UNMASTERED, PROCESSING, READY, FAILED, PUBLISHED, SPLITTING while being split into other episodes, SPLIT once it has been), duration, error message when failed, pubDate when published, the latest re-master's status (NONE, PENDING, FAILED) with its error when failed, masteringMethod: what made the current master (auphonic for the site's connected Auphonic account, omumu_chain for Omumu's own chain, null when not mastered yet or mastered by an older Omumu chain), masteringFallback: why Auphonic could not master it when Omumu's chain made it instead (out_of_credits, key_rejected, production_failed, unreachable, timed_out), null when it did not fall back, noiseReduction: whether the episode is mastered with noise reduction, and publishIntent: whether it publishes itself when mastering ends. The source of a split is listed too; it cannot be edited, published or re-mastered, and its parts are the episodes now.

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

No parameters.

`omumu_podcast_episode_master`

Starts mastering an UNMASTERED episode (one created with deferMastering), returning it in PROCESSING status; it reaches READY when mastering ends. Episodes in any other status are refused with their current status: their mastering has already started.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id
`noiseReduction`	boolean	no	Whether to master with noise reduction, for a recording made outdoors or in a noisy room. Omit to keep the episode's choice.

`omumu_podcast_episode_publish`

Publishes a READY episode: the pubDate is stamped now and the episode appears in the RSS feed. A PROCESSING episode records a publish intent instead and is returned unpublished with publishIntent true: it publishes itself when mastering ends, with the pubDate stamped then, and the intent survives a failed mastering and its retry. An UNMASTERED episode starts mastering with a publish intent the same way, so it goes live mastered. Cancel it with omumu_podcast_episode_publish_cancel until then. A FAILED episode is refused with its status.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_publish_cancel`

Cancels the publish intent of an episode that is still being mastered, or whose mastering failed: it stays a draft when mastering ends. Refused once the episode is published.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_episode_remaster`

Re-masters an episode's original upload with the current mastering chain. A READY episode returns to PROCESSING. A PUBLISHED episode stays in the feed with its current audio while remasterStatus is PENDING; when the new master lands it replaces the audio, and a failure sets remasterStatus FAILED with remasterError and leaves the published audio unchanged. The episode's transcript and edits are kept. Refused, with the reason, for published episodes whose listeners get an edited version, UNMASTERED, PROCESSING or FAILED episodes, episodes with a re-master already pending, the source of a split (SPLITTING or SPLIT), and the parts of a split.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id
`noiseReduction`	boolean	no	Re-master with or without background-noise removal, and keep that choice for the episode; omit to keep the episode's current choice

`omumu_podcast_episode_retry`

Re-runs mastering on a FAILED episode, returning it to PROCESSING. Episodes in any other status are refused with their current status.

- **Write category:** standard write
- **Access:** requires the `PODCAST_WRITE` permission

Parameter	Type	Required	Description
`episodeId`	string	yes	The episode id

`omumu_podcast_show_get`

Gets the site's podcast show: title, slug, description, language, and the public RSS feed URL. Errors if the site has no show yet (shows are created in the podcast admin).

- **Write category:** read-only
- **Access:** requires the `PODCAST_READ` permission

No parameters.

The Agent Surface

`omumu_profile_get`

Tells the host who this connection belongs to, so several connected Omumu sites read as separate accounts. Returns your user id on this site (stable, and different on each site), your name, your email and a nickname naming you and the site.

- **Write category:** read-only
- **Access:** all agents

No parameters.

`omumu_site_context`

Tells you which Omumu site this connection acts on. Call it first. Returns the site's canonical host and public URL, its site id, your role on the site (PLATFORM_ADMIN, HEADMASTER, STAFF, STUDENT, or null when you hold none), the site's enabled features, and highLevelConnected: whether a Headmaster has connected the site to HighLevel, which a quiz needs before it can deliver its respondents there (it is connected in the admin; no tool reads or writes it). Needs no scope and returns no personal data and no credential.

- **Write category:** read-only
- **Access:** all agents

No parameters.

`omumu_submission_get`

Reads one hand-in: everything its list row carries, plus the assignment's id and title, the participant's text, the handed-in file's name, type and size, the file's text in file.text when it is a .txt or .md file of at most 1 MB, the transcript of an audio or video file when there is one (its status otherwise), the link to the hand-in's page, and every feedback row oldest first: its kind (AI_RATING or HUMAN_REVIEW), body, optional score, createdAt, and the author's name for a staff review. Never includes mentor notes, the rater's reading trace or a file download link. Requires the handin:read scope. Reading changes nothing, and marks nothing as seen.

- **Write category:** read-only
- **Access:** requires the `HANDIN_READ` permission

Parameter	Type	Required	Description
`submissionId`	string	yes	The submission ID (see omumu_submission_list)

`omumu_submission_list`

Lists the hand-ins (submissions) on an assignment, oldest first by when they were handed in. Pass since to get only those handed in at or after a time: a daily poll passes the submittedAt of the last hand-in it saw and skips the ids it already has. Each row carries the submission id, the participant (userId, name, email), submittedAt, the light (GREEN, YELLOW or RED on a Check-in, null otherwise), the mentor's reviewState, the transcriptStatus, and hasAiFeedback and hasStaffFeedback. Paginate with limit and offset; totalCount counts every match. Use omumu_submission_get to read one. Returns participants' personal data, so it requires the dedicated handin:read scope. Reading changes nothing.

- **Write category:** read-only
- **Access:** requires the `HANDIN_READ` permission

Parameter	Type	Required	Description
`assignmentId`	string	yes	The assignment ID (see omumu_assignment_list)
`limit`	number	no	Maximum hand-ins to return (default: 50, max: 100)
`offset`	number	no	Number of hand-ins to skip for pagination (default: 0)
`since`	string	no	Optional ISO-8601 time, inclusive: only hand-ins handed in at or after it, e.g. 2026-10-27T08:00:00+01:00. Without an offset it is read in the server's time zone. Omit to list every hand-in.

Product Skills

`omumu_skill_get`

Returns the complete SKILL.md instructions for a skill the user is entitled to. Use after omumu_skills_list to load the workflow and follow its steps. Returns an error if the user is not entitled or the skill is not published.

- **Write category:** read-only
- **Access:** requires the `SKILL_READ` permission

Parameter	Type	Required	Description
`slug`	string	yes	The skill slug, e.g. 'create-course'

`omumu_skills_list`

Lists every published Omumu skill (workflow for AI assistants) visible to the caller's site, marking which the user is entitled to read. Use to discover what workflows are available — course creation, validation, sales pages, etc. Set entitledOnly=true to hide skills the user is not entitled to.

- **Write category:** read-only
- **Access:** requires the `SKILL_READ` permission

Parameter	Type	Required	Description
`entitledOnly`	boolean	no	If true, only return skills the user is entitled to. Default false.

Permission scopes

Permission	Grants
ANALYTICS_DETAILED	Access detailed analytics and conversion data
ANALYTICS_READ	Read analytics and performance data
API_KEY_MANAGE	Manage API keys for the site
AUTOMATION_READ	Read automations, canvas state, and options
AUTOMATION_WRITE	Create, update, activate, and deactivate automations
BRAND_READ	Read the site's Brand: published and draft choices and derived colors
BRAND_WRITE	Update the site's Brand draft; publishing stays a human step in admin
COURSE_DELETE	Delete courses and course components
COURSE_READ	Read courses and course structure
COURSE_WRITE	Create and update courses, modules, and lessons
EMAIL_INBOX_READ	Read received emails in the inbox
EMAIL_SEQUENCE_READ	Read email sequences and their content
EMAIL_SEQUENCE_WRITE	Create and update email sequences and follow-ups
FUNNEL_READ	Read funnels and their branched step graph
FUNNEL_WRITE	Add upsell steps to funnels (requires UPSELLS feature on the site's tier)
HANDIN_READ	Read hand-ins, their transcripts and feedback, including participants' personal data
INVITE_READ	Read Invites, window state, and waitlist size
INVITE_WRITE	Update Invites (name, quota, schedule, active flag) and launch
[illegible]	[illegible] course, email sequences, an opt-in form, and a page back

CHAPTER 12

The Command Line and Companion Apps

Everything in this manual so far happens in a chat with your assistant or in the admin. This chapter is for the other two places work gets done: a terminal, and the devices you record on. **omumu-cli** puts the whole tool surface of your site behind one command, for you, for a script, or for an agent such as Claude Code running on your laptop. **Omumu Pod** records a podcast episode on your phone and hands it to your site's podcast pipeline. **Omumu Loupe** records your screen on a Mac, zooms and pans for you, and uploads the result into a lesson. The chapter says what each does, how it gets in, and where each stops — and which of them you can get today.

Concepts

One tool surface, several front doors. Your site's AI connection (the Agent Surface chapter) is a set of named tools — `omumu_offer_list`, `omumu_page_update` and so on. The CLI calls those same tools over the same connection; it adds nothing the chat cannot do and leaves nothing out. What it adds is a terminal: typed arguments, JSON output, exit codes, and a self-describing schema an agent can read instead of guessing.

Two ways in. `omumu login` opens your browser and asks you to authorise the CLI on the site, exactly as you authorise a chat app — the Headmaster approves once, and the CLI gets an access token for that site. For a script or a scheduled job there is no browser, so you create an **API key** under *Integrations → MCP API Keys* with only the permissions the job needs and pass it with `--token` or the `OMUMU_API_KEY` variable. The browser login asks for a fixed, broad set of permissions and cannot ask for less; a narrow key is the way to give a script exactly what it needs.

The config file is a credential. Logins are stored in `~/.omumu/config.json`, one profile per site (`--name` on login, `OMUMU_SITE` to choose one), readable by your user only. A browser login's token lasts an hour and the CLI does not renew it; when a call says authentication failed, log in again. An API key lasts until you delete it.

Companion apps use the same sign-in. Pod and Loupe sign in to a site by its host name, in the system browser, with the same authorisation screen; their tokens live in the phone's or Mac's keychain and renew themselves. Each app then talks to the site through the tool surface: an upload, a tool call to create the episode or attach the video, and a poll until processing is done.

Hands-on

Install and log in

Install with Homebrew (`brew tap omumuas/omumu && brew install omumu`; a Mac on Apple silicon gets a native binary, other machines a jar that needs Java 21). Then:

```
omumu login --url https://mira.myomumu.com
```

The CLI registers itself with the site, prints the authorisation address and opens it in your browser; approve, and it reports *Logged in.* `omumu status` confirms the connection and counts the tools you can reach. Log in to a second site with `--name` to keep both.

Run any tool

Mira wants the week's offers in a terminal: `omumu call offer_list` (the `omumu_` prefix is optional). Every tool takes its arguments as `--arg name=value`, typed from the tool's schema — numbers, booleans and JSON arrays or objects are parsed — or as one JSON object with `--input`, inline, from a file (`@offer.json`) or from stdin (`-`); use a file or stdin for anything secret, so it stays out of your shell history.

```
omumu call offer_create --arg title="Automate Your Week" --
arg totalAmountCents=149000 \
    --arg 'paymentProviders=["VIPPS"]'
omumu call page_update --input
'{"pageId":"BBSR5F6MDX5OG","contents":"# About Mira"}'
```

`--json` gives machine-readable output for every command, and `omumu schema --json` lists every tool with its parameters and the exact `omumu call` line for it, which is what an agent reads first. A misspelled tool or argument is refused with the known names. A handful of common tasks also have their own commands: `omumu course list`, `get` and `create`, and `omumu skill upload` for platform skills.

Let an agent drive it

Claude Code on Mira's laptop gets a line in its instructions: *"Omumu is managed with `omumu call`; run `omumu schema --json` to see the tools."* From there it lists, creates and updates like the chat assistant does, with one difference worth knowing: a terminal agent is only as careful as its instructions, and page and course writes are live at once. Give it an API key with the permissions the job needs rather than your browser login.

Record an episode on your phone

Omumu Pod, coming to Google Play, is the podcast chapter's phone end: sign in to the site, record as many takes as you like (redo the last one with a tap), trim each on its waveform, reorder and merge them, then give the episode a title, show notes, a type and a season and number. The app uploads the audio, creates the episode, waits while the site masters it, and publishes it into the feed with one tap — or retries if mastering failed. The site needs a show first (the Podcasting chapter), and the Podcasting plan feature.

Record your screen on a Mac

Omumu Loupe is the Mac app we record our own tutorials with; it is not distributed to creators yet. It records a screen or a window with system audio and your microphone, follows your cursor and clicks with automatic zoom and pan, and gives you a timeline to adjust every move it made, with click ripples and keystroke badges if you want them. Export to MP4, MOV or GIF up to 4K — or sign in to your site and *Upload to Omumu*: pick a course, a module and the lesson the recording belongs to, or upload it as a standalone resource to place later (the Courses & Content chapter).

What you cannot do yet

Stated plainly: `omumu signup`, which creates a whole new site from the terminal behind a one-time card payment in the browser, is switched off on omumu.com today, so a site still starts at `/trial` (Getting Started). The CLI's browser login asks for one broad set of permissions and leaves out the podcast tools, so podcast work from the terminal needs an API key that includes them; browser tokens expire after an hour and are not renewed. The CLI has no commands of its own beyond courses and skills — everything else is `omumu call`. Omumu Pod is not in the store yet, Loupe is not distributed, and Omumu Cam, a phone video recorder in the same family, is not released.

Reference

No MCP tools cover this chapter's capabilities yet.

CHAPTER 13

Product Skills

Your AI assistant can already do things on your site: the tools in the Agent Surface chapter let it create courses, pages, offers and email. What the tools don't carry is *method* — the order to do things in, the questions to ask you first, the mistakes to avoid. That is what a skill is for. A skill is a written workflow your assistant loads and follows, so that "help me build a course" turns into an interview, an angle, a structure and a built course rather than a pile of generated text.

This chapter is short on purpose. Skills are Omumu's own, you don't write them yet, and there are a handful of them. What it covers is what they are, which ones you have, and the two ways to use them: live over the connection, where your assistant loads a skill when it needs it, and installed in your AI client, where the skill lives permanently. A path such as *Products → Skills* names a group in the admin's navigation and the page under it.

Concepts

A skill is a folder with a manifest. The manifest is a file named `SKILL.md`: a short header naming the skill and describing when to use it, followed by the instructions themselves. The description is the part that matters most to the assistant — it is how the assistant decides, from what you said, that this skill applies. Alongside the manifest a skill can ship supporting files: the steps of

a longer workflow, reference notes, templates. Omumu's `create-course` skill, for example, is its manifest plus four step files (extract, angle, structure, build). The whole folder is the skill; the manifest alone is a summary of it.

Skills and tools are not the same thing. A tool does one concrete thing against live data — create a lesson, list your offers — and your assistant calls it with exact arguments. A skill is procedure: which tools to call, in what order, after asking you what. A skill made only of words can change how your assistant works but cannot touch your site; it needs the tools for that. Some skills also lean on what your AI client can do on its own, such as reading web pages: `validate-idea` and `competitor-scan` do their research that way, so they need a client that can browse.

Skills are published by Omumu and come with your plan. Every skill on your site is a platform skill, written and published by Omumu. If your plan lets you connect an AI assistant, you are entitled to every published skill, with nothing to buy and nothing to grant. Your assistant sees the whole published list, and each entry says whether you are entitled to it. Today three are published: `create-course`, `validate-idea` and `competitor-scan`.

A skill is loaded, not remembered. Over the connection, your assistant fetches the skill's text at the moment it needs it, so you always get the current version. An installed copy is a snapshot: it stays as it was the day you downloaded it until you download it again.

Hands-on

Say Mira has a webinar's worth of notes on automating invoice reminders and wants them to become her second course.

Let your assistant build a course with a skill

Mira says: *"I have notes from my invoice-automation webinar. Turn them into a course on my site."*

The assistant lists the skills the site offers, sees that `create-course` is the one for this, loads it and follows it: it asks Mira what she actually knows and who it is for, proposes an angle, lays out modules and lessons for her to approve, and only then builds the course with the course tools. Mira never pastes the

notes into a page herself; the skill's job is to keep the assistant from skipping the thinking.

Underneath, two tools do the work. `omumu_skills_list` returns every published skill visible to the site, with its slug, name, description and whether the caller is entitled to it; `omumu_skill_get` returns the full manifest of one skill the caller is entitled to. Your site's onboarding page for assistants tells any AI that reads it to do exactly this: list the skills, load the one that fits, and follow it in the conversation rather than working from memory. A skill the caller is not entitled to is refused with a plain error, and the assistant is told not to recommend it.

There is no admin-UI path for this: using a skill is something your assistant does.

See which skills you have

Open *Products → Skills*. The page lists every published skill you are entitled to, with its description and a download button. The same list, with the entitlement flag, is what `omumu_skills_list` returns to your assistant. On a plan without an AI connection the page is not offered; if Omumu has published nothing yet, the page says so and notes that your assistant can still use Omumu through the connection.

Install a skill in Claude.ai

Loading a skill over the connection is enough for most work. Install it when you want it present in every conversation without the assistant first looking it up — for example, if Mira builds courses often and wants Claude to reach for the course workflow on its own.

On *Products → Skills*, choose *Download .skill* on the skill you want. The file is a ZIP holding the skill's folder — its manifest and every supporting file — in the shape Claude's uploader expects. In Claude.ai, upload the file under *Customize → Skills* in your settings. The skill then activates by itself whenever your prompt matches its description. The AI setup page (*Integrations → Set up your AI assistant*) links to the Skills page at the bottom for the same reason.

For Claude Code, unzip the file and put the folder under `~/.claude/skills/` for yourself, or under `.claude/skills/` in a project you share.

The downloaded manifest carries a note of who downloaded it. Skills are Omumu's product; the file is for your own clients.

What you cannot do yet

Stated plainly: you cannot write a skill of your own on your site, upload one, or sell one. The platform's skill model has room for site-owned skills and a creator-to-creator marketplace, but neither is built, and your offers cannot include a skill. Publishing skills is Omumu's job today. If a workflow you need is missing, that is feedback for Omumu, not something the Skills page can take.

Reference

`omumu_skill_get`

Returns the complete SKILL.md instructions for a skill the user is entitled to. Use after omumu_skills_list to load the workflow and follow its steps. Returns an error if the user is not entitled or the skill is not published.

- **Write category:** read-only
- **Access:** requires the `SKILL_READ` permission

Parameter	Type	Required	Description
`slug`	string	yes	The skill slug, e.g. 'create-course'

`omumu_skills_list`

Lists every published Omumu skill (workflow for AI assistants) visible to the caller's site, marking which the user is entitled to read. Use to discover what workflows are available — course creation, validation, sales pages, etc. Set entitledOnly=true to hide skills the user is not entitled to.

- **Write category:** read-only
- **Access:** requires the `SKILL_READ` permission

Parameter	Type	Required	Description
`entitledOnly`	boolean	no	If true, only return skills the user is entitled to. Default false.