{{ swl }}

Reference

interactive web apps in Python, without a JavaScript frontend

version 5.6.0-2-g29b47b0 · swl.seecr.nl

seecr

Contents

Setting up an application 2
The `SWL` object 2
Actions 3
Responses 3
Saved 4
What a request sends 5
Placeholders and rehydrate 5
Live updates 6
Deep links 6
Tabs 7
Tabs rendered by the server 8
Bootstrap tabs, loading when shown 8
Templates 9
`swl/base.j2` 9
Other templates 9
Form fields 10
Help balloons 10
Styling 10
Built-in views: `swl.views.account` 11
Javascript 12
Translations 12
Configuration 13
Testing 14
Debugging 14
Listing the routes 14
Troubleshooting 15

SWL (Seecr Web Lib) is a small layer on top of Starlette, Jinja and htmx 4. It does not need a CSS framework: its modal, dialog, toast and context menu are plain HTML (`<dialog>`, popover), and it can bring its own base styles or work with Bootstrap 5 (see Styling). This page lists everything SWL itself provides. For a guided introduction, read `example/README.md` first; this page is for looking things up.

Setting up an application

```
from starlette.applications import Starlette
from starlette.routing import Mount, Route
from seecr_stdlib.config import Config
from swl import SWL, StaticFiles, static_dir

swl = SWL(config=Config({}), templates=[my_templates_dir])  # your templates first, then SWL's
swl.register_modules(my_views)                               # modules with __actions__
swl.register_action_kwargs(db=db)                            # extra arguments for every action
swl.add_template_globals(VERSION=swl.SWL_VERSION)            # used by swl/base.j2

app = Starlette(routes=[
    swl.action_route,                                        # /action/{action}
    Route("/", swl.endpoint(page, redirect="/login")),       # a page for logged-in users
    Mount("/static", StaticFiles(directory=static_dir)),     # swl.js and friends
])
```

`swl.StaticFiles` is Starlette's `StaticFiles` (same arguments) that sends `Cache-Control: no-cache` with every file: the browser keeps its copy but asks the server before using it, a cheap `304 Not Modified` when it did not change. Without it a browser may keep using an old `swl.js` for days after an update, also when the page's own url has a `?version=`: JavaScript modules import `./swl.js` without it. Mount your own static files with it too.

Login needs Starlette's `SessionMiddleware` and `AuthenticationMiddleware` with `swl.session.SessionAuthBackend(userbase)`; see `example/swl_tasks/app.py` for a complete setup with users, sessions and the websocket.

The `SWL` object

Member	Purpose
`SWL(config, templates=[])`	`templates`: your template directories, searched before SWL's own
`register_modules(*modules)`	register every function in each module's `__actions__`
`register_action_kwargs(**kwargs)`	keyword arguments passed to every action (`swl` is always passed)
`action_route`	the Starlette route for all actions
`register(name)`	url of the action with that function name; raises if it does not exist
`endpoint(fn, authenticated=True, is_authorized=None, redirect=None, paths=None)`	wrap a page function: same arguments as an action, with a login check. `redirect`: where to send users who are not allowed (the requested path is kept in `session["next"]`); without it they get 403. `paths`: see Listing the routes
`success()`, `fail()`	start a response, see Responses
`template(name)`	a Jinja template, to `.render(...)` into a string
`template_response(request, name, context)`	a full page response
`add_template_globals(**kwargs)`, `add_template_filters(**kwargs)`	extend Jinja
`styles`, `scripts`	the web resources, used by `swl/base.j2`
`routes_route`	the `/.routes` endpoint, see Listing the routes
`SWL_VERSION`	version of SWL

Actions

An action is an async function that the browser can post to:

```
async def task_save(request, swl, db, **kwargs):
    async with request.form() as form:
        ...
    return swl.success().toast("Saved")

__actions__ = [
    task_save,                                          # login required
    dict(method=login, authenticated=False),            # open to everyone
    dict(method=admin_only, is_authorized=user_is_admin),    # login + your own check
]
```

- It is called as `fn(request, swl=..., **registered_kwargs)`. Always accept `**kwargs`, so adding a registered keyword later does not break existing actions.
- `is_authorized` is `async def check(request, **kwargs) -> bool`, called only for logged-in users.
- Not allowed: HTTP 403 (shown as an error in the browser). Unknown action: HTTP 404.
- Its url is `/action/<md5 of the function name>`. Function names must be unique over all registered modules. In templates and Python use `register("task_save")`, never the url itself.
- An action may also return any Starlette response, but only SWL responses do the things below.

Responses

`swl.success()` and `swl.fail()` return a builder. Each method returns the builder, so calls can be chained; the browser carries them out after the new content is in place, **in the order they were called**.

Method	Effect in the browser
`hydrate(id, html, rehydrate=None)`	the element with that id gets `html` as content. `rehydrate=dict(action=url, context={...})` also changes how it reloads (see rehydrate). Hydrating the same id twice keeps the last
`rehydrate(*ids, context=None)`	these placeholders reload themselves; `context` is added to that one request
`open_modal(html, size="lg", scrollable=True, call=None, rehydrate=None)`	`html` in the modal; `size`: `"sm"`, `"lg"` or `"xl"`; `call`: function name(s) to run once the modal is visible; `scrollable`: only the body (`.swl-modal-body`) scrolls, the header and footer stay in place (`False`: the whole modal scrolls)
`hydrate_modal(html, rehydrate=None, call=None)`	replace the content of the (open) modal; `call` runs when it is or becomes visible
`close_modal()`	close the modal
`open_contextmenu(html, location)`	`html` as a menu at `location={"x": ..., "y": ...}` (use the posted `event_x` / `event_y`)
`toast(text, delay=None, background=None)`	a short message; `text` may be HTML; defaults from the config
`saved(id, delay=None)`	"Saved" for a moment at the element with that id, e.g. a field that saves itself on change: in its `.swl-saved` element, or in one added at its end. Fades out after `delay` ms (config `response.saved.delay`); call it for more than one element. See Saved
`call(function, **kwargs)`	`window[function](kwargs)`
`trigger(event, **detail)`	dispatch your own browser event with `event.detail = detail`; it bubbles up to `document`
`reset_form()`	reset the form of the element that made the request

Method	Effect in the browser
`session_store(key, value)`	`sessionStorage[key] = JSON.stringify(value)`
`set_url(url, **params)`	replace the address bar url (no page load)
`redirect(url)`	load `url` as a new page
`swl.fail().message(text)`	show `text` in the error dialog; nothing else happens

`html` is inserted as HTML. Render it from a Jinja template (autoescaped) or escape user input yourself.

What goes over the wire, in case you look at it in the browser:

Response	Status	Body	Headers
success with `hydrate` / `open_modal` / `toast` /…	200	`<hx-partial hx-target="#id" hx-swap="innerHTML">html</hx-partial>` per element	`HX-Trigger: {"swl-...": {...}}`
success without content	204	empty	`HX-Trigger`, `HX-Redirect`, `HX-Replace-Url` as needed
fail	422	empty	`HX-Trigger: {"swl-error": {"message": "..."}}` `HX-Reswap: none`

The event names in `HX-Trigger` are `swl-call`, `swl-rehydrate`, `swl-recipe`, `swl-modal-open`, `swl-modal-call`, `swl-modal-close`, `swl-contextmenu`, `swl-toast`, `swl-saved`, `swl-reset-form`, `swl-session-store` and `swl-error`, plus your own from `trigger()`.

Saved

A field that saves itself when it changes (`hx-post` on the field) tells the user it did with `saved`: a quiet "Saved" that fades out, where the user is looking, instead of a toast.

```
<div id="field-phone">
    <input name="phone" value="{{ customer.phone }}" hx-post="{{ register('save_phone') }}">
    <span class="swl-saved" role="status">
        <template><i class="bi bi-cloud-check"></i> {{ _("Saved") }}</template>
    </span>
</div>
```

```
async def save_phone(request, swl, **kwargs):
    async with request.form() as form:
        ...  # save it
    return swl.success().saved("field-phone")
```

SWL is the mechanism; the application decides how it looks. Put the `.swl-saved` element where it should appear, with what it should show in a `<template>`: SWL puts a copy of it in the element (in a `.swl-saved-content` span) and adds the class `swl-saved-shown`; after the delay the class goes and, once faded out, the copy too. Without a `<template>` it shows "✓ Saved" in the language of the page; without a `.swl-saved` element SWL adds one at the end of the element. `swl.css` only fades it in and out (in `:where()`, so your CSS wins), e.g.

```
.swl-saved { color: green; font-weight: 600; }
```

With `role="status"` (or `aria-live="polite"`) on the `.swl-saved` element from the start, screen readers read the copy out when it appears: it is added, not just made visible. A failed save (`swl.fail()`) shows its message as usual and no "Saved".

What a request sends

Element	Sends
`button`, `input`, `select`, `textarea` with `hx-post`	all fields of its form: the enclosing `<form>`, or the one named in its `form="..."` attribute
`form` with `hx-post` (posts on submit, also on Enter)	its fields
anything else (`div`, `a`, `tr`, `span`, a placeholder)	**only** its `hx-vals` and the fields of the elements in its `hx-include`, not the fields inside or around it
`swl_action(url, values)` in javascript	only `values`

On top of that:

- `hx-vals='{"key": "value"}'` adds values (write it as `hx-vals='{{ {"key": value}|tojson }}'` in Jinja); `hx-vals:inherited` on a parent adds them to every request from inside it.
- `placeholder=<id>` is added when the element has an id (unless a field with that name is sent already). Actions use it to know which element to hydrate.
- A rehydrate adds its `context`.
- A right-click (`hx-trigger="contextmenu prevent"`) adds `event_x` and `event_y`, the click position on the page.
- As in a normal form post: unchecked checkboxes and disabled fields are not sent.

Placeholders and rehydrate

A **placeholder** is an element that loads its own content from an action:

```
{% from "swl/placeholder.j2" import placeholder %}
{{ placeholder("orders", register('orders_list'), context={"customer": customer.id}) }}
```

renders

```
<div id="orders" hx-post="/action/..." hx-trigger="load, swl-refresh" hx-vals='{"customer": "c12"}'></div>
```

Argument	Meaning
`id`	element id, posted as `placeholder`
`action`	action url
`context`	values posted with every request (`hx-vals`)
`on_changed`	list of qids; reload when the server signals one of them changed (live updates)
`lazy=True`	do not load with the page, only when rehydrated
`trigger`	extra htmx triggers, e.g. `"input delay:300ms from:#filters"`
`attrs`	extra attributes, e.g. `{"hx-include": "#filters", "hx-sync": "this:queue last"}`
`tag`, `class_`	element tag (default `div`) and class

The action hydrates the placeholder: `swl.success().hydrate(placeholder_id, html)`. Read the id from the posted `placeholder` field when the same action serves several placeholders.

Rehydrate asks a placeholder to reload itself, with the action and values it already has:

- from an action: `swl.success().rehydrate("orders", context={"page": 2})`;
- from javascript: `swl.rehydrate("orders", {page: 2})`;
- from the server, for all open pages: see live updates.

Any element can be rehydrated once it has a "recipe": an action and a context. A placeholder has one from the start. `hydrate(id, html, rehydrate=dict(action=url, context={...}))` sets or changes it, so the element reloads from that action with that context from then on (it becomes `hx-post` / `hx-vals` on the element).

Rehydrating only reloads the element itself, not placeholders around it. An id that is not on the page is ignored.

Live updates

1. Route `WebSocketRoute("/ws", dispatcher.endpoint)` with a `swl.dispatcher.Dispatcher()`, and load `/static/swl/websocket.js` in the page.
2. Give placeholders `on_changed=["<qid>", ...]` (or the attribute `data-swl-on-changed="qid1 qid2"`).
3. After a change, `await dispatcher.signal_change(obj)`; `obj.qid` is the qid (or pass the qid itself). Every open page reloads the placeholders that list it.

The qid is any string you choose: an object id (`"Order:123"`) to refresh only pages that show that order, or a collection name (`"orders"`). The websocket needs `uvicorn[standard]` (or `websockets`) on the server and websocket support in any proxy in front of it.

Only some users. Without more, the qid goes to *every* open page, also to pages that do not show it; only the reload itself checks who may see what. When the qid tells something (a private conversation, `"Chat:alice:bob"`), send it to the users concerned only:

```
await dispatcher.signal_change(conversation, users=["alice", "bob"])
```

The example's private tasks do this (`_signal_change` in `example/swl_tasks/views.py`).

A page belongs to the user that was logged in when it connected: `websocket.user.identity`, Starlette's `BaseUser.identity`, so the `AuthenticationMiddleware` must run for the websocket too (it does when it is app middleware). For SWL's `Userbase` users the identity is the username; an application's own user class implements `identity`. A user with the page open on two devices gets it on both. `Dispatcher(identify=fn)` takes another function `fn(websocket) -> identity or None`.

Dispatcher	
`Dispatcher(authenticated=False, identify=...)`	`authenticated=True`: refuse websockets without a logged-in user (close code 1008)
`await signal_change(obj, users=None)`	`obj`: object with a `qid`, or the qid; `users`: identities (or one) to send it to
`connected_users()`	identities with at least one open page
`Dispatcher(idle_timeout=seconds)`	close a page that sent nothing for that long (websocket.js pings every 10 s; background tabs about once a minute, so keep it above that)
`async user_connected(identity)` / `async user_disconnected(identity)`	for subclasses: the first page of a user connected, the last one closed

The user is looked up once, when the page connects. After a logout the page goes to the login page, which closes its websocket.

Missed changes. `websocket.js` reconnects when the connection drops (with a growing delay, at most 30 s). Changes signalled while it was down do not reach the page, so after a reconnect it reloads every placeholder with `on_changed`. A connection that dies without closing (a phone that slept, another network) is noticed by a ping without an answer within 5 s; when the page becomes visible again or the browser comes back online it pings at once.

Deep links

A screen that actions built up (a placeholder inside a placeholder inside a placeholder, a selected item, an open modal) is not in the url. Reload the page or open a bookmark, and you are back at the start. A deep link fixes that in three steps:

1. **Describe the screen in the url.** The action that changes the screen calls `set_url(...)`, e.g. `set_url(f"/orders/{order_id}")` or `set_url("/orders", order=order_id)`. The address bar changes; nothing is loaded.
2. **Route that url.** A Starlette route accepts it (path parameters or query parameters) and renders the page.
3. **Rebuild the screen** from what the url says, in one of two ways:
   - **Placeholder context**, when the state is *what a placeholder shows* (a filter, a page number, which order the detail panel shows): render the placeholder with that state as its `context`. It loads with it straight away, and every level can pass on what the level below needs, all the way down. `example/swl_tasks` does this for its search and status filter.
   - **Kickstart**, when the state is *the result of an action* (an open modal, a context menu, something an action hydrates into a nested placeholder): render the page with `kickstart={"action": swl.register("..."), "kwargs": {...}}`. After the page has loaded **and its placeholders have loaded** (no requests for 100 ms, or at most 5 seconds), the action is posted with those values, and its response is handled like any other. Because it waits, it can hydrate elements inside nested placeholders.

```
async def order_show(request, swl, db, **kwargs):
    """Opens an order in the modal, and puts it in the url."""
    async with request.form() as form:
        order_id = form["order"]
    html = swl.template("orders/order.j2").render(order=db.order(order_id))
    return swl.success().open_modal(html).set_url(f"/orders/{order_id}")


async def orders_page(request, swl, **kwargs):
    context = {"title": "Orders", "body_class": ""}
    if (order_id := request.path_params.get("order_id")) is not None:
        context["kickstart"] = {"action": swl.register("order_show"), "kwargs": {"order": order_id}}
    return swl.template_response(request, "orders/page.j2", context)


routes = [
    Route("/orders", swl.endpoint(orders_page, redirect="/login")),
    Route("/orders/{order_id}", swl.endpoint(orders_page, redirect="/login")),
]
```

Things to know:

- `set_url` *replaces* the url (no new history entry), so the browser's Back button does not step back through screens; it leaves the page.
- Set the url back when the state ends, e.g. `set_url("/orders")` in the action that closes the modal. The modal's own close button does not call the server; for that, reset the url in javascript: `document.getElementById("swl-modal").addEventListener("close", () => history.replaceState(null, "", "/orders"))`
- One kickstart per page. If rebuilding takes several steps, let that one action do them all (it can hydrate several placeholders and open a modal in one response).
- The values come from the url, so anyone can put anything there. The action checks them like any other request (it *is* the same action); the template escapes them safely into the page.
- From javascript: `swl.kickstart(url, values)` does the same at any moment.

`example/swl_tasks` shows the placeholder-context way for a list with a detail panel and tabs: the url is `/tasks/<id>/<tab>?q=...`, and a reload shows the same task on the same tab (section 7 of its README).

Tabs

There are two ways to make tabs. The examples use Bootstrap's `nav-tabs` look; with any framework you can use `<nav class="swl-tabs">` with buttons that have `aria-selected="true"` on the active one (see `example/swl_tasks/templates/tasks/detail.j2`). The second way needs Bootstrap's javascript.

Tabs rendered by the server

The tabs are buttons that post an action with the tab they stand for. The action renders the whole tab panel with that tab active and only that tab's content, and puts it in the url:

```
{# panel.j2, rendered by the item_panel action #}
<ul class="nav nav-tabs">
    {% for name in tabs %}
    <li class="nav-item">
        <button type="button" class="nav-link {{ 'active' if name == tab }}"
                hx-post="{{ register('item_panel') }}"
                hx-vals='{{ {"item": item.id, "tab": name}|tojson }}'>{{ name|capitalize }}</button>
    </li>
    {% endfor %}
</ul>
<div class="p-3">{% include "items/tab_" ~ tab ~ ".j2" %}</div>
```

```
async def item_panel(request, swl, db, **kwargs):
    async with request.form() as form:
        item_id = form["item"]
        tab = form.get("tab") if form.get("tab") in TABS else TABS[0]  # from the url: check it
    html = swl.template("items/panel.j2").render(item=db.item(item_id), tab=tab, tabs=TABS)
    return swl.success().hydrate("item-panel", html).set_url(f"/items/{item_id}/{tab}")
```

The server always knows which tab is open, so a page route for `/items/{item_id}/{tab}` can render the same panel straight into the page and a reload shows the same tab (see Deep links). This is the simplest way, and the one to use by default.

Bootstrap tabs, loading when shown

Bootstrap switches the panes in the browser; each tab loads its own content when it is shown (Bootstrap's `show.bs.tab` event). Use this when tabs are expensive to render and you don't want to render them before they are opened.

```
<ul class="nav nav-tabs">
    <li class="nav-item">
        <button type="button" class="nav-link active" data-bs-toggle="tab" data-bs-target="#tab-overview"
                hx-post="{{ register('item_tab') }}" hx-vals='{"tab": "overview"}'
                hx-trigger="load, show.bs.tab">Overview</button>
    </li>
    <li class="nav-item">
        <button type="button" class="nav-link" data-bs-toggle="tab" data-bs-target="#tab-history"
                hx-post="{{ register('item_tab') }}" hx-vals='{"tab": "history"}'
                hx-trigger="show.bs.tab">History</button>
    </li>
</ul>
<div class="tab-content">
    <div id="tab-overview" class="tab-pane active"></div>
    <div id="tab-history" class="tab-pane"></div>
</div>
```

- The action hydrates the pane of its tab (`hydrate("tab-history", html)`).
- Only the tab that is active when the page loads has `load` in its trigger; write it as `hx-trigger="{{ 'load, ' if active }}show.bs.tab"`.
- A tab loads again every time it is shown; `hx-trigger="show.bs.tab once"` loads it only the first time.
- Like any button, a tab posts its enclosing form, plus its `hx-vals`.
- For a deep link, let the action call `set_url(...)` with the tab, and let the page route render that tab as the active one.

This replaces `data-swl-tab` from SWL before htmx.

Templates

`swl/base.j2`

The page skeleton: the web resources (see Styling), htmx, SWL's styles, the message dialog and the toast.

- Variables: `title`, `body_class`, `VERSION` (required); `lang` (default: the language of the request, see Translations); `kickstart` (optional: `{"action": url, "kwargs": {...}}` posts that action once the page and its placeholders have loaded, see Deep links; `kwargs` may also be a JSON string).
- Blocks: `head` (all of it), `head_extra`, `body`, `body_extra` (after the scripts).
- It sets the htmx config `noSwap: [204, 304, "4xx", "5xx"]`: error responses are never put into the page.

Load SWL itself in `body_extra`:

```
<script type="module" id="swl-run" src="/static/swl/swl-run.js"></script>
<script type="module" src="/static/swl/websocket.js"></script>  <!-- only for live updates -->
```

Other templates

Template	What	Needed for
`swl/modal.j2`	the modal: a `<dialog id="swl-modal">` with `#modal-content`	`open_modal`, `hydrate_modal`
`swl/modal-content.j2`	base for modal content: blocks `modal_title`, `modal_content`, and optionally `modal_icon` (left of the title, e.g. `<span class="swl-avatar swl-avatar-lg">`), `modal_subtitle` (a line below it) and `modal_footer`. Any button with `data-swl-close` (or `data-bs-dismiss="modal"`) closes the modal. Only `modal_content` (`.swl-modal-body`) scrolls, the header and footer stay in place (`open_modal(scrollable=False)`: the whole modal scrolls); content of your own without a `.swl-modal-body` scrolls as a whole	
`swl/contextmenu.j2`	the context menu: a popover `#swl-contextmenu`; put a `<div class="swl-menu">` with buttons in it	`open_contextmenu`
`swl/dialog.j2`, `swl/toast.j2`	message dialog (`<dialog>`) and toast (popover); included by `base.j2`	errors, `hx-confirm`, `toast`
`swl/placeholder.j2`	the `placeholder` macro	
`swl/inputs.j2`	macros `text_input`, `password_input`, `number_input`, `date_input`, `textarea_input`: a label above the control (`swl-field`) with `SWL.ui` classes, for every framework. `mandatory=True` makes it required, `optional=True` adds "(optional)" to the label, `hint` a help text below it, `id` a fixed id. See Form fields	

Template	What	Needed for
`swl/info.j2`	the `info(text, label=None)` macro: a small (i) button that shows `text` in a balloon. See Help balloons	
`swl/card.j2`, `swl/footer.j2`, `swl/alert.j2`, `swl/breadcrumbs.j2`	small Bootstrap pieces (older; need Bootstrap's CSS)	

Form fields

`swl/inputs.j2` builds a field the way SWL's own forms do: the label above the control, the classes of the framework.

```
{% from "swl/inputs.j2" import text_input, textarea_input %}
<form class="swl-stack" hx-post="{{ register('save_customer') }}">
    {{ text_input(_("Name"), "name", value=customer.name, mandatory=True, autocomplete="name") }}
    {{ text_input(_("Phone"), "phone", value=customer.phone, optional=True) }}
    {{ textarea_input(_("Notes"), "notes", value=customer.notes, optional=True,
                      hint=_("Only for your colleagues")) }}
</form>
```

When most fields of a form are required, mark the few optional ones (`optional=True` shows a quiet "(optional)" that is never split over two lines) instead of an asterisk on every required one. `mandatory=True` adds `required`, so the browser asks for the field before the form is sent.

Help balloons

An explanation that would be a hover text on a computer cannot be read on a phone. `info` puts it behind a small (i) instead: clicking or tapping it shows the text in a balloon, and a click elsewhere or Escape closes it.

```
{% from "swl/info.j2" import info %}
<label for="colour">{{ _("Main colour") }}</label>
{{ info(_("Buttons, links and what is selected")) }}
```

The balloon is a native popover (`popovertarget`), so it is on top of everything, also of an open modal (Escape then closes the balloon first); `swl.js` places it below the (i), or above it when there is no room, always inside the window. The text is escaped (pass `Markup` for HTML); `label` is what a screen reader says for the button (default "More information", translated). Put the (i) next to a label, not inside it: a click on a `<label>` also goes to its control. The look: `.swl-info` (the button) and `.swl-info-text` (the balloon), with the `--swl-*` variables.

Styling

Choose a CSS framework with one config key:

```
swl = SWL(config=Config({"ui": {"framework": "bulma"}}))
```

`ui.framework`	Loads[1]	Use it for
`"bootstrap"` (default)	Bootstrap 5.3 CSS and JS, Bootstrap Icons, htmx	existing applications built on Bootstrap
`"tabler"`	Tabler 1.6 (an admin design on top of Bootstrap 5, with its own javascript bundle that includes Bootstrap's), Bootstrap Icons, htmx	admin screens and dashboards; a Bootstrap application gets Tabler's look without changing its templates
`"bulma"`	Bulma 1.0 (CSS only), Bootstrap Icons, htmx	applications that like Bulma
`"swl"`	SWL's own `swl-base.css`, Bootstrap Icons, htmx	no CSS framework at all

The choice does two things:

[1] All from cdn.jsdelivr.net, with integrity hashes.

- **The web resources** that `base.j2` loads (`swl/resources/*-web-resources.json`). To load something else (your own mirror, other versions), point `paths.resources` at your own JSON file with `styles` and `scripts` (it must include htmx); the framework's classes still apply.
- **`SWL.ui`**: CSS classes for form elements and buttons in that framework, so a template works with each of them. SWL's own account and user screens use it, and so can yours:

```
<input name="title" class="{{ SWL.ui.input }}">
<button type="submit" class="{{ SWL.ui.button_primary }}">Save</button>
<button type="button" class="{{ SWL.ui.button }} {{ SWL.ui.small }}">Edit</button>
```

  Names: `input`, `select`, `textarea`, `checkbox`, `button`, `button_primary`, `button_danger`, `small`, `table` (see `swl/frameworks.py`). `SWL.framework` is the name, and `<html data-swl-framework="...">` lets your CSS differ per framework.

SWL's modal, message dialog, toast and context menu do not depend on the framework: they are plain HTML (`<dialog>`, popover) styled by `swl.css`. On phones they are at most as high as what is visible (`100dvh`, so the buttons of a long modal are not behind the browser's bars) and below 576 pixels they take almost the whole width. In a modal only the body scrolls (`open_modal(scrollable=True)`, the default). Their colours are CSS variables that follow Bootstrap's, Tabler's or Bulma's colours when one is loaded; a theme is a few lines in your own stylesheet:

```
:root { --swl-accent: #FAA21B; --swl-radius: .25rem; }
```

`--swl-bg`, `--swl-fg`, `--swl-muted`, `--swl-border`, `--swl-subtle-bg`, `--swl-accent`, `--swl-accent-fg`, `--swl-radius`, `--swl-danger`, `--swl-warning`, `--swl-success`, `--swl-info`.

`swl.css` also has layout classes that work with every framework: `swl-container`, `swl-nav` (top bar, with `swl-brand`), `swl-columns` (columns that stack on small screens; `swl-wide` makes the first one wider), `swl-card` (with a `<header>` and `swl-card-body`), `swl-tabs`, `swl-stack`, `swl-row`, `swl-end`, `swl-field` (a label above an input; `swl-optional`, `swl-hint`), `swl-list`, `swl-badge`, `swl-muted`, `swl-small`, `swl-hidden`, `swl-link` (a button that looks like a link), `swl-icon-button` and `swl-menu`. The example application uses only these and `SWL.ui`, so it runs with each framework: `python -m swl_tasks --framework bulma`.

With `"swl"`, `swl-base.css` styles plain HTML (forms, buttons, tables, headings) and follows the browser's light or dark mode (Bulma does too; Bootstrap only with `data-bs-theme`).

Light and dark. `/static/swl/theme.js` lets the visitor choose light, dark or automatic (the computer's setting) for every framework: it sets `data-theme` (swl-base.css, Bulma) and `data-bs-theme` (Bootstrap) on `<html>` and remembers the choice in the browser. Load it in the head, so the page is painted in the right colours at once, and call `swl_theme.set("dark")`, `"light"` or `"auto"` from a button; a `swl-theme` event on `document` reports changes:

```
{% block head_extra %}<script src="/static/swl/theme.js"></script>{% endblock %}
```

Switching the framework in the browser (for a demo or a style guide): add `swl.frameworks.FrameworkCookieMiddleware` to the application. A cookie `swl_framework` with a framework name (`swl`, `bootstrap`, `tabler`, `bulma`) then overrides `ui.framework` for that visitor; set it from javascript and reload the page. The example application has both switches in its top bar (`templates/tasks/style_switch.j2`, `static/tasks.js`).

In every template: `register(name)` and `SWL` (the `SWL` object).

Built-in views: `swl.views.account`

Register with `swl.register_modules(account)` and `register_action_kwargs(userbase=...)`:

Action	
`login`	open to everyone; posts `username`, `password`, `next`; redirects to `next`

Action	
`show_account`	"My account" modal to change your own password (`change_password`, confirms with a toast)
`users_show`	modal with all users; creating, deleting and making admin need an admin

`account.logout` is a page function for a `/logout` route.

Javascript

On `window` after `swl-run.js` has loaded:

`swl_action(url, values)` / `swl.action(...)`	post to an action from javascript; `values` is an object or a query string; returns a promise
`swl.rehydrate(id, context)`	reload a placeholder
`swl.kickstart(url, values)`	like `swl_action`, but first waits until the page and its placeholders have loaded
`swl.ui.confirm(message)`	SWL's confirm dialog; a promise of `true` / `false`
`swl.ui.error(message)`, `swl.ui.info(message)`	message dialogs
`swl_theme.get()`, `swl_theme.set("light" | "dark" | "auto")`	light or dark colours; only when the page loads `/static/swl/theme.js` (see Styling)

Events you can listen to:

`swl-modal-shown` on `#swl-modal`	the modal opened (`open_modal`)
`close` on `#swl-modal`	the modal closed (a close button, `close_modal()`); it is a native `<dialog>`
`swl-theme` on `document`	the colours changed (`theme.js`); `e.detail.mode`
your own, on `document`	`swl.success().trigger("name", **detail)`

Functions the server calls with `call()` or `open_modal(call=...)` must be global: define them in a classic `<script>` or assign them to `window` in a module.

Attributes SWL adds to htmx:

`hx-confirm="..."`	shown in SWL's dialog instead of the browser's (`hx-confirm="js:..."` is left to htmx)
`data-swl-validate="fn"`	`window.fn()` must return true, or the request is not sent
`data-swl-on-changed="qid ..."`	see live updates
`data-swl-close`	on a button inside a `<dialog>` (the modal): closes it; `data-bs-dismiss="modal"` works too

Everything else is plain htmx 4. The texts `swl.js` shows (dialog buttons, error messages) are put in the page by `base.j2`, in the page's language (see Translations).

Translations

Texts are written in English and marked with `_()`; SWL picks the language per request and translates them with gettext (Babel). SWL's own texts (dialogs, account and user screens, messages of `swl.views.account`) come in Dutch, German, French and Spanish. The example (`example/README.md`, "Translations") is translated into Dutch, German and Spanish.

Marking texts. In Python:

```
from swl.i18n import _, ngettext, N_

swl.fail().message(_("Passwords do not match"))
_("Delete %(name)s?", name=user.name)
ngettext("%(num)d user", "%(num)d users", len(users))
STATUSES = {"open": N_("Open")}       # marked, translated later: _(STATUSES[s])
```

`_()` translates when it is called, in the language of the request it is called for; texts in module constants are marked with `N_` and translated where they are used. In templates:

```
{{ _("Users") }}  {{ _("Delete %(name)s?", name=user.name) }}
{% trans count=users|length %}{{ count }} user{% pluralize %}{{ count }} users{% endtrans %}
```

Choosing the language. Add the middleware after the `AuthenticationMiddleware` and list the languages:

```
swl = SWL(config=Config({"i18n": {"locales": ["en", "nl"]}, ...}))
app = Starlette(..., middleware=[..., Middleware(AuthenticationMiddleware, ...),
                                 swl.i18n_middleware])
```

For every request and websocket it takes the first of: the application's selector (`swl.i18n.locale_selector(fn)`, `fn(conn)` returns a language or `None`; `conn.user` is there, e.g. for a language in the user's profile), the cookie `swl_lang`, the browser's `Accept-Language`, the first of `i18n.locales`. Only listed languages are used, `nl-BE` matches `nl`. Without the middleware everything is English, as written.

Catalogs. An application keeps its translations in `<package>/locale`:

```
python -m swl.i18n_tool extract myapp        # texts -> myapp/locale/messages.pot
python -m swl.i18n_tool init myapp nl        # once per language: .../nl/LC_MESSAGES/messages.po
python -m swl.i18n_tool update myapp         # after extract: new texts into every .po
python -m swl.i18n_tool compile myapp        # .po -> .mo, which is what is used
```

and adds them with `swl.i18n.add_translations(path / "locale")`; they win over SWL's. Commit the `.po` and the `.mo` files.

Dates and numbers. Template filters (and functions in `swl.i18n`) in the language of the request: `format_date(d, "medium")`, `format_time`, `format_datetime(dt, tzinfo=...)`, `format_skeleton(d, "MMMEd")` (`Mon, Nov 3` / `ma 3 nov`), `format_number`, `format_timedelta`.

`SWL.lang`	the language of the request for `<html lang>` (`nl`, `pt-BR`)
`swl.i18n.activate(locale)` / `deactivate(token)`	a language outside a request (tests, jobs)
`swl.i18n.current_locale()`	`"nl"`, or `None` outside a request

Configuration

`SWL(config=Config({...}))` reads:

Key	Default	
`actions.path`	`/action`	url prefix of actions
`actions.hash`	`true`	`false`: use function names in urls instead of hashes
`ui.framework`	`"bootstrap"`	`"swl"`, `"bootstrap"`, `"tabler"` or `"bulma"`; see Styling
`paths.resources`	the framework's	your own JSON file with `styles` and `scripts` for `base.j2` (must include htmx)
`urls.cdn`	`https://cdn.jsdelivr.net`	`{BASE_URL}` in resource urls; `base.j2` preconnects to it
`response.toast.delay`	`500`	toast duration in ms

Key	Default	
`response.toast.background`	none	toast background color
`response.saved.delay`	`2500`	how long "Saved" shows (`saved()`), in ms
`i18n.locales`	`["en"]`	the languages to use, the first is the default; see Translations
`i18n.cookie`	`swl_lang`	the cookie that chooses a language
`routes.allowlist`	`ROUTES_ALLOWLIST`	who may read `/.routes` : IPs/CIDRs, a list or a comma-separated string; see Listing the routes

Testing

Post to actions with Starlette's `TestClient` ; `swl.test_utils` checks SWL responses:

```
from swl.test_utils import assert_fail, assert_success

response = client.post(swl.register("task_save"), data={"title": "x"})
events = assert_success(response)            # status 200/204; returns the HX-Trigger events
assert events["swl-toast"] == {"delay": 500}

assert_fail(client.post(swl.register("task_save"), data={}), "A task needs a title")
```

`response.text` holds the `<hx-partial>` elements, `response.headers["HX-Redirect"]` etc. the rest. `example/swl_tasks/app_test.py` has complete examples, including login and the websocket.

Debugging

- **Network tab.** Every action is a POST to `/action/...` . The body shows what was sent; the *response headers* show what SWL told the page to do (`HX-Trigger`). A 204 with an empty body is normal for actions that only rehydrate, toast or close something.
- **htmx logging.** Type `htmx.config.logAll = true` in the browser console to see every htmx event, with the element and request.
- **Console errors.** SWL reports functions it cannot find (`SWL: function not found: ...`). A `422` line is a `swl.fail()` and is expected.
- **Which action is `/action/3f2a...` ?** Set `actions.hash` to `false` during development to see function names in urls.
- **Test the action alone.** Most problems are easier to see in a `TestClient` test than in the browser.

Listing the routes

`GET /.routes` answers with JSON listing every route of the app (path, an anchored regex, methods, ...), e.g. for a proxy that only forwards known paths. Add it to the app's routes:

```
swl = SWL(config=Config({"routes": {"allowlist": ["127.0.0.1", "10.0.0.0/8"]}}))
app = Starlette(routes=[..., swl.routes_route])
```

Only clients in the allow list get an answer, all others a 404. The allow list holds IPs and CIDRs, as a list or a comma-separated string. `swl.routes_route` takes it from the config key `routes.allowlist` ; without that key from the environment variable `ROUTES_ALLOWLIST` (default `127.0.0.1,::1`). Behind a proxy the client is the proxy, unless the ASGI server trusts its `X-Forwarded-For` (uvicorn `--proxy-headers --forwarded-allow-ips=<proxy-ip>`).

A route with a parameter, like `/{slug}`, is listed as a pattern that matches every path of that shape. To list only the paths that exist, give its endpoint a `paths` function. It gets the same arguments as the page function (sync or async) and returns the paths, relative to the route's `Mount` like the route's own path. It is called on every request to `/.routes`; each path becomes a static entry with an exact regex, and the pattern itself is left out:

```
def project_paths(request, store, **kwargs):
    return [f"/{project.slug}" for project in Project.all(store) if project.active]

Route("/{slug}", swl.endpoint(project_page, authenticated=False, paths=project_paths))
```

Troubleshooting

Nothing happens when I click. - Is htmx loaded? `htmx.version` in the console. If it is undefined, check the resources and the `integrity` hash (a wrong hash makes the browser refuse the script). - Does the element have `hx-post`? Is it inside content you inserted with your own javascript? Then call `htmx.process(element)`. - Did a request go out (network tab)? If it did, see the next questions.

An error dialog says the action is not allowed (403). Not logged in, the session expired, or `is_authorized` returned false. Register open actions with `authenticated=False`.

My fields are not posted. - Placeholders, links and other non-form elements only send `hx-vals` and `hx-include`, see What a request sends. Add `hx-include="#my-form"` or `hx-include="closest form"`. - `hx-get` and `hx-delete` do not send the enclosing form; use `hx-post` or `hx-include`. - `hx-vals` must be valid JSON: double quotes inside, single quotes around the attribute.

An upload gives "This is too large to send" (HTTP 413). The web server in front of the application refused the request before it got there: raise its limit, e.g. nginx's `client_max_body_size` (1 MB by default), a little above the largest upload the application accepts.

An attribute on a parent element is ignored. htmx 4 does not inherit attributes unless asked: write `hx-confirm:inherited`, `hx-vals:inherited`, `hx-target:inherited`.

Advice I found online does not work. Most htmx examples on the web are for htmx 2. Common differences in htmx 4: inheritance is explicit (above); events are named `htmx:after:swap` instead of `htmx:afterSwap`; `hx-disable` is now `hx-ignore` and `hx-disabled-elt` is now `hx-disable`; `hx-vars` and the `HX-Trigger-After-Settle` header are gone. See https://four.htmx.org/whats-new-in-htmx-4.

After a reload the screen is back at its start. The screen is not in the url. See Deep links.

A kickstarted action does not show anything. Check the page source for the kickstart script (did the route set `kickstart`?) and the network tab for the request. If its target is inside a placeholder, that placeholder must load without the kickstart's help: the kickstart waits for loading to stop, it cannot make a placeholder appear.

The page reloads when I press Enter. A form with an `action` or `method` attribute submits normally. Remove them (SWL stops Enter in bare forms), or give the form `hx-post`.

`hydrate` runs but nothing changes. The id in `hydrate()` must match an element on the page, exactly (it is not a CSS selector). The modal content is `"modal-content"` (`swl.responses.MODAL_CONTENT`); it only exists when the page includes `swl/modal.j2`.

The modal, context menu or toast does not show. Include `swl/modal.j2` / `swl/contextmenu.j2` in the page; `base.j2` includes the toast and dialog. They need a browser with `<dialog>` and the popover API (all current browsers). An application that has its own Bootstrap modal markup with id `swl-modal` needs Bootstrap's javascript.

"Function not found" for my `call()`. The function is not global; see Javascript.

A rehydrate does nothing. The element must be on the page and be a placeholder (or have a recipe from `hydrate(..., rehydrate=...)`). Check the id.

My placeholder loads twice / requests are cancelled. Two triggers fire close together (a rehydrate and a live update, or fast typing). Add `hx-sync`: `"this:queue last"` runs them one after another and keeps the last; `"this:replace"` cancels the running one.

Live updates do not arrive. Check that `websocket.js` is loaded, the `/ws` route exists, the server has `uvicorn[standard]` (a log line "No supported WebSocket library detected" means it doesn't), the qid in `signal_change` matches `on_changed` exactly, and that `signal_change` is awaited. With `users=...`: the page's user must have that identity (`connected_users()` shows who is connected); with `authenticated=True`: the websocket must get the session cookie.

Error pages appear inside my page. The page lacks the htmx config from `base.j2`: `<meta name="htmx-config" content='{"noSwap": [204, 304, "4xx", "5xx"]}'>`.

`pip install` fails on `seecr-stdlib`. SWL needs `seecr-stdlib`, which is not on PyPI. Install it from Seecr's repository first.