

Package: paparazzi (via r-universe)

October 14, 2026

Title Scripted Demo Videos, Screenshots, and Tests for Web Pages

Version 0.0.0.9000

Description Script polished demo videos and screenshots of anything that runs in a web browser, including 'Shiny' apps, 'Quarto' documents, and live websites. A pipeable interface built on 'chromote' clicks, types, and waits the way a person would. While recording, a cursor glides between elements, text is typed one character at a time, and a camera zooms and pans to the action. Annotations, callouts, spotlights, captions, and redactions follow their elements as the page changes. Frame captures to an element or aspect ratio, emulate phones, dark mode, and other devices, and write MP4, WebM, or GIF files. Without a recording, the same scripts run at full speed as browser tests with auto-waiting actions and retrying expectations that work with 'testthat'.

License MIT + file LICENSE

URL <https://posit-dev.github.io/paparazzi/>,
<https://github.com/posit-dev/paparazzi>

BugReports <https://github.com/posit-dev/paparazzi/issues>

Depends R (>= 4.1.0)

Imports chromote, cli, jsonlite, later, pillar, processx, promises,
R6, rlang (>= 1.2.0), tibble, utils, withr

Suggests av, gifski, htmltools, httpuv, knitr, pkgdown, png,
rmarkdown, shiny, testthat (>= 3.2.0), xfun

Config/Needs/website av, brand.yml, gifski, png, shiny

Config/roxygen2/version 8.1.1

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first record, stage, example, cursor

Encoding UTF-8

Roxygen list(markdown = TRUE)

Collate 'actions.R' 'annotate-callout.R' 'annotate-spotlight.R'
 'overlay.R' 'annotate.R' 'camera.R' 'caption.R' 'context.R'
 'utils-purrr.R' 'cursor.R' 'device.R' 'drag-preview.R'
 'example.R' 'expect.R' 'fonts.R' 'frame.R' 'geometry.R' 'get.R'
 'inspect.R' 'js.R' 'keys.R' 'loc.R' 'nav.R' 'open.R'
 'paparazzi-package.R' 'record.R' 'resolve.R' 'scope.R'
 'screenshot.R' 'serve.R' 'shiny-input.R' 'stage.R' 'style.R'
 'utils-check.R' 'utils-examples.R' 'utils.R' 'wait.R'

VignetteBuilder knitr

Config/pak/sysreqs chromium make libssl-dev

Repository <https://posit-dev.r-universe.dev>

Date/Publication 2026-10-14 14:02:21 UTC

RemoteUrl <https://github.com/posit-dev/paparazzi>

RemoteRef HEAD

RemoteSha ab8fc677a3dff078a409d419b091a6b233c7abbd

Contents

paparazzi-actions	4
pz_act_blur	5
pz_act_click	6
pz_act_drag	7
pz_act_focus	10
pz_act_hover	11
pz_act_press	12
pz_act_scroll	14
pz_act_select_text	16
pz_act_type	17
pz_annotate	19
pz_annotate_callout	20
pz_annotate_caption	23
pz_annotate_clear	24
pz_annotate_redact	24
pz_annotate_spotlight	25
pz_camera	26
pz_camera_reset	27
pz_chromote	28
pz_close	29
pz_cursor_hide	29
pz_cursor_leave	30
pz_cursor_move	31
pz_cursor_show	33
pz_device	34
pz_example	36

pz_expect_attr	37
pz_expect_checked	38
pz_expect_class	39
pz_expect_count	40
pz_expect_enabled	41
pz_expect_exists	42
pz_expect_focused	43
pz_expect_in_viewport	44
pz_expect_js	45
pz_expect_style	46
pz_expect_text	48
pz_expect_title	49
pz_expect_url	50
pz_expect_value	51
pz_expect_visible	53
pz_find	54
pz_find_first	55
pz_find_last	56
pz_find_nth	57
pz_find_pop	58
pz_find_reset	59
pz_font_bunny	60
pz_font_file	60
pz_font_google	61
pz_frame	62
pz_get_attr	64
pz_get_count	65
pz_get_elements	66
pz_get_html	66
pz_get_rect	67
pz_get_style	68
pz_get_text	69
pz_get_title	70
pz_get_url	71
pz_get_value	71
pz_inspect	72
pz_js	73
pz_loc	74
pz_nav_goto	75
pz_open	77
pz_record	79
pz_record_hold	80
pz_record_pause	81
pz_record_start	82
pz_record_stop	84
pz_screenshot	85
pz_serve_quarto	87
pz_serve_shiny	88

pz_serve_static	89
pz_set_files	90
pz_set_shiny_input	91
pz_set_value	92
pz_stage	93
pz_stage_annotate	95
pz_stage_fonts	97
pz_stage_frame	99
pz_wait	100
pz_wait_for_js	101
pz_wait_for_navigation	102
pz_wait_for_shiny_idle	103
pz_wait_for_stable	104
pz_with_page	106
Index	108

paparazzi-actions	<i>How actions work</i>
-------------------	-------------------------

Description

The contract shared by the `pz_act_*`() functions.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

pz_act_blur	<i>Blur the focused element</i>
-------------	---------------------------------

Description

Removes focus from the current scope's element, or at the root context from whatever element currently has focus (`document.activeElement`). A no-op when the body is focused. Useful to clear focus rings before a screenshot.

Usage

```
pz_act_blur(ctx, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

[pz_act_focus\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_focus("#task-title")

# Remove the focus ring, e.g. before a screenshot
page |> pz_act_blur()
pz_expect_focused(page, target = "#task-title", not = TRUE)
pz_close(page)
```

pz_act_click

Click an element

Description

Auto-waits for the element to be actionable – visible with a non-empty box, the same “visible” `pz_expect_visible()` uses, and receiving pointer events at its center (not covered by another element) – then scrolls it into view and clicks the center of it with real browser input events: a mouse move to the point, then a left-button press and release. The page sees a trusted pointer sequence – exactly what a user’s click produces – so `:hover` state, focus, and click handlers all behave as they would live. While recording, the staging settings (`pz_stage()`) animate the scroll, the cursor glide, and the press; otherwise everything runs straight to the final state.

Usage

```
pz_act_click(ctx, target = NULL, ..., effect = NULL, effect_color = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope; at the root context a target is required.
<code>...</code>	Checked empty; reserved for future use.
<code>effect</code>	Click feedback while recording: <code>NULL</code> (the default) uses the page’s <code>pz_stage()</code> <code>click_effect</code> setting. <code>"press"</code> scales the cursor down while pressed, <code>"ring"</code> draws an expanding ring that fades out at the click point instead of scaling, and <code>"none"</code> shows nothing. Without a recording no effect is drawn.
<code>effect_color</code>	CSS color of the <code>"ring"</code> effect. <code>NULL</code> (the default) uses the page’s <code>pz_stage()</code> <code>click_effect_color</code> setting.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_hover()`, `pz_act_type()`, `pz_act_press()`

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_click("#toggle-help")
pz_get_text(page, target = "#toggle-help")

# In a scoped context, target = NULL clicks the scope element
page |>
  pz_find(pz_loc(".task-done", within = pz_loc(".task", has_text = "bank"))) |>
  pz_act_click()
pz_get_count(page, target = ".task.done")
pz_close(page)
```

`pz_act_drag`

Drag an element to another element or by an offset

Description

Auto-waits for the source to be actionable: visible, non-empty, and receiving pointer events at its center (not covered by another element). With `to`, the destination is checked for visibility and that it receives the drop at its center after the source is brought into view. It then drags with real mouse input: press at the source's center, move to the destination, release. When the source is a real HTML5 drag source (`draggable`, including inherited `draggable` or the image/`<a href>` defaults), the drag runs through the browser's drag pipeline instead: the press and move start a genuine `dragstart` (so `dataTransfer`

holds whatever the page put there), and the drop is delivered to the destination as trusted **dragenter/dragover/drop** events with that payload.

While recording, the staging (**pz_stage()**) shows the press at the source and glides the cursor while holding from the source to the destination, streaming the path's real input events (held moves, or **dragenter/dragover** on the HTML5 path) along the glide, so pointer-following content tracks the cursor and intermediate elements see the drag pass. The drop lands as the cursor arrives. On the HTML5 path the carry runs between the page's **dragstart** and the replayed **drop**, so **dragstart** styling stays visible through the carry, and the real pointer stays by the source until the release, so the destination shows no hover during the carry.

to names the element to drop onto; **by = c(x, y)** drops at that offset in pixels from the source's center. Supply exactly one.

Usage

```
pz_act_drag(ctx, target, to = NULL, ..., by = NULL, preview = TRUE)
```

Arguments

ctx	A paparazzi context.
target	A CSS selector string, a pz_loc() spec, or a list of specs (a union matching any of them). The drag source. Unlike most actions it is always required.
to	A CSS selector string, a pz_loc() spec, or a list of specs (a union matching any of them): the drop target. NULL omits the absolute destination. Supply exactly one of to or by .
...	Checked empty; reserved for future use.
by	Offset in pixels from the source's center, c(x, y) (or a single number for both axes). NULL disables the offset mode.
preview	Whether to show a static source image during recorded HTML5 drags and a brief post-drop settling animation. Must be named.

Value

ctx, invisibly.

Drag preview

With **preview = TRUE**, paparazzi carries a static image of the HTML5 drag source during an active, unpaused recording with a visible cursor. Paparazzi captures the image before **dragstart** styling; a handle inside a draggable element previews that element. After the drop, the image briefly settles to the same source node's new bounds if it survives with a supported, unchanged size. Paparazzi never hides or replaces the page's source. The preview is not the browser's native drag image and does not reproduce custom **DataTransfer.setDragImage()** feedback.

Paparazzi disables the optional preview with a warning in these cases. The drag continues without the preview.

- **Unreadable stylesheets**, often on `file://` pages: open the page over HTTP. `pz_open()` serves local `.html` paths over HTTP when `httpuv` is installed and loads. You can also serve the page with `pz_serve_static()` and open the server handle with `pz_open()`. Cross-origin stylesheets must also allow the browser to read their rules.
- **Iframes or SVG <use> references in the source**: use a source without these references; they can copy content outside the source subtree.
- **Unsupported CSS rule groups, imported stylesheets, or font sources**: move rules out of unsupported groups or imports and embed font sources as data URLs in `@font-face` rules. Registered fonts also need a readable embedded CSS representation.
- **Source-related or overlapping paparazzi redactions, or uncertain redaction state**: keep redactions separate from the source or turn off the preview. Paparazzi permits capture only when it can verify that redactions are separate; do not remove redactions from sensitive content for a preview.
- **Unsupported source geometry or content**: use a source without transforms, filters, clipping, overflowing content, shadow DOM, or generated content. Fixed or sticky descendants and out-of-box positioned content are also unsupported.

Other preview failures report the underlying error message. Use `preview = FALSE` to turn off capture, carry imagery, and settling. Unrecorded, paused, hidden-cursor, and ordinary mouse drags do no preview work.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_click()`, `pz_act_hover()`

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_text(page, target = ".task-title")

# The tasks are HTML5 drag sources; drop "Water the plants" on the first task
```

```

page |>
  pz_act_drag(
    pz_loc(".task", has_text = "plants"),
    to = pz_loc(".task", which = "first")
  )
pz_get_text(page, target = ".task-title")
pz_close(page)

```

pz_act_focus

Focus an element

Description

Scrolls the element into view (instantly) and focuses it via the browser's element focus method, so the page shows focus rings and enabled-input styles exactly as a user would see them. Focus is an element-state change, not an input event, so the direct method call is the faithful implementation (Playwright does the same).

Usage

```
pz_act_focus(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope; at the root context a target is required.
<code>...</code>	Checked empty; reserved for future use.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_blur()`, `pz_act_type()`

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_focus("#task-title")
pz_expect_focused(page, target = "#task-title")
pz_close(page)
```

pz_act_hover	<i>Hover the pointer over an element</i>
--------------	--

Description

Auto-waits for the element to be actionable – visible with a non-empty box and receiving pointer events at its center (not covered by another element) – then scrolls it into view and moves the pointer to the center of it with a real `mousemove` event, without pressing any button. This is what drives `:hover` styles and `mouseenter/mouseover` handlers.

Usage

```
pz_act_hover(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope; at the root context a target is required.
<code>...</code>	Checked empty; reserved for future use.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

[pz_act_click\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
passport <- pz_loc(".task", has_text = "passport")

# Tasks change colour on :hover
pz_get_style(page, "background-color", target = passport)
page |> pz_act_hover(passport)
pz_get_style(page, "background-color", target = passport)
pz_close(page)
```

`pz_act_press`

Press key combinations

Description

Presses one or more key combinations against whatever the page currently has focused, e.g. "Enter", "Control+A", `c("Shift+Tab", "Escape")`. A vector presses each combination fully (down then up) in order. Specs are "Mod+Mod+Key" strings with the modifiers Control, Shift, Alt, Meta, and Mod (matched case-insensitively) and a named key (Enter, Tab, Escape, Backspace, arrows, F1-F12, ...) or a single printable character. Mod resolves to Meta on browsers reporting a Mac platform (`navigator.userAgentData.platform` or `navigator.platform`), and Control otherwise. Following Playwright, an uppercase letter or shifted symbol implies Shift: "Control+A" sends Control+Shift+A.

Keys only reach focused elements; call `pz_act_click()` or `pz_act_focus()` first to focus the element you're typing into.

Usage

```
pz_act_press(ctx, key, ..., show_keys = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>key</code>	A character vector of key specs.
<code>...</code>	Checked empty; reserved for future use.
<code>show_keys</code>	NULL uses the page's <code>pz_stage()</code> setting (initially "none"). "words" shows named modifier keycaps, "mac" uses Mac symbols, and "both" shows Mod as Ctrl / . Keystroke callouts appear only in recordings; subsequent calls replace earlier ones. A vector is displayed as a single sequence. In a GIF recording, keystroke callouts need the av package.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_type()` to insert text.

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_act_type("Buy milkk", target = "#task-title") |>
  pz_act_press("Backspace")
pz_get_value(page, target = "#task-title")

# Enter submits the form; the page shows "Saving..." before the task appears
```

```

page |> pz_act_press("Enter")
pz_expect_text(page, "Saved", target = "#status")
pz_get_text(page, target = pz_loc(".task-title", which = "first"))

# A vector presses keys in sequence; + joins keys pressed together
page |> pz_act_press(c("Tab", "Shift+Tab"))
pz_expect_focused(page, target = "#task-title")
pz_close(page)

```

pz_act_scroll	<i>Scroll the page or an element into view</i>
---------------	--

Description

Exactly one of **target**, **by**, and **to**:

- **target**: auto-waits for a match, then scrolls it into view (instantly outside a recording).
- **by** = `c(x, y)`: scrolls the current scope's scroll container – the scope element or its nearest scrollable ancestor, or the page itself at the root context – by that many pixels.
- **to**: scrolls that same container to an edge or corner from the direction vocabulary: "top", "bottom", "left", "right", the four corners, or "center" (e.g. "bottom" scrolls to the end; "top right" to the top-right corner).

Scrolling is instant outside a recording; while recording it is staged as real mouse wheel events with the cursor over the container, so the video shows the scroll. Wheel scrolling is best-effort: if the page swallows the events, the instant scroll still guarantees the final position.

Usage

```

pz_act_scroll(ctx, target = NULL, ..., by = NULL, to = NULL, duration = NULL)

```

Arguments

ctx	A paparazzi context.
target	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). Scrolled into view. <code>NULL</code> disables the element-target mode; use by or to instead.
...	Checked empty; reserved for future use.
by	Offset in pixels, <code>c(x, y)</code> (or a single number for both axes). <code>NULL</code> disables the offset mode.
to	A direction string: the sides, the four corners, or "center". <code>NULL</code> disables the direction mode.
duration	Seconds per staged wheel scroll; <code>NULL</code> computes the time from the scroll distance and <code>cursor_speed</code> in <code>pz_stage()</code> . Applies only while recording, including target and any scroll needed to bring a scoped container into view. Use 0 to scroll instantly without wheel animation.

Value

ctx, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_find()`, `pz_act_click()`

Examples

```
page <- pz_open(pz_example("tasks"), height = 600)

# With a target, scroll it into view
page |> pz_act_scroll("#toggle-help")
pz_expect_in_viewport(page, target = "#toggle-help")

# With `to` or `by`, scroll the current scope's container: here the list
page |>
  pz_find(".task-list") |>
  pz_act_scroll(to = "bottom")
pz_expect_in_viewport(page, target = pz_loc(".task", which = "last"))

# At the root, the container is the page itself
page |> pz_act_scroll(to = "top")
pz_js(page, "window.scrollToY")
pz_close(page)
```

`pz_act_select_text` *Select text inside an element*

Description

Auto-waits for a match, then highlights the exact `text` inside it. `paparazzi` finds the substring among the element's text nodes and sets the page's real window selection, even when the match spans inline tags such as `<em>` and `<strong>`. Typing afterwards replaces it – call `pz_act_type()` with `target = NULL`, which inserts into whatever has focus.

Usage

```
pz_act_select_text(ctx, text, ..., target = NULL)
```

Arguments

<code>ctx</code>	A <code>paparazzi</code> context.
<code>text</code>	A string to select. Must appear exactly in the element, across tags if needed; not found is an error.
<code>...</code>	Checked empty; reserved for future use.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope; at the root context a target is required.

Details

While recording with a visible cursor, `paparazzi` animates a drag across the text, then sets the exact selection with the browser's Range API. The drag sends real mouse-move events, which can trigger page hover effects such as tooltips or hover menus. Outside a recording, selection is instant and leaves the cursor where it was, without sending mouse-move events.

A contenteditable target is focused too (dragging across editable text focuses it, and that focus is where the typing lands); a static target is not.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_type()`, `pz_set_value()`

Examples

```
page <- pz_open(pz_example("tasks"))

# Typing replaces the selection
page |>
  pz_find(pz_loc(".task", has_text = "Renew passport")) |>
  pz_act_click(".task-edit") |>
  pz_find(".task-title") |>
  pz_act_select_text("passport") |>
  pz_act_type("driving licence") |>
  pz_act_press("Enter") |>
  pz_get_text()
pz_close(page)
```

<code>pz_act_type</code>	<i>Type text into an element</i>
--------------------------	----------------------------------

Description

With a `target`, auto-waits for the element to be actionable – visible with a non-empty box and receiving pointer events at its center (not covered by another element) – then scrolls it into view, clicks the center of it (real mouse events, so the element genuinely gains focus), and inserts `text` at the caret – the caret lands where the click lands, just like a real user. With `target = NULL` at the root context, inserts into whatever element currently has focus; if nothing editable is focused, the text goes nowhere, exactly like typing into a page with no focused field.

Insertion is instant (one `insertText`), except while recording with `typing = "natural"` (the default; see `pz_stage()`): one `insertText` per character with randomized delays around `typing_speed`, so the video shows the text appearing. While recording, paparazzi fades the cursor out during typing so it doesn't cover the text. The next pointer action fades it back in. For a value-setting primitive that works on selects, checkboxes, and range inputs, see `pz_set_value()`.

Usage

```
pz_act_type(ctx, text, ..., target = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>text</code>	A string to type.
<code>...</code>	Checked empty; reserved for future use.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope or, at the root context, the focused element.

Value

`ctx`, invisibly.

Acting on the page

The `pz_act_*`() functions use the page the way a person would. They send real browser input, such as pointer moves, clicks and key presses, or use the browser's own focus and text-selection methods. They never set a form value directly.

An action with a `target` looks for it in the current scope and waits until it matches an element. Inside a scope, `target = NULL` acts on the scope's element, and scrolling with `by` or `to` scrolls the scope's container. An explicit target is scrolled into view first when needed. A few actions, like `pz_act_press()`, take no target and act on the focused element instead.

While the page is recording, actions are staged as `pz_stage()` sets them up: the cursor glides to pointer targets, typing is paced, scrolls normally use the mouse wheel, and the page holds for the staged `pause` afterward. Without a recording, actions go straight to their final state.

To set a value directly instead, without staging, use `pz_set_value()`, `pz_set_files()`, or `pz_set_shiny_input()`.

See Also

`pz_act_press()` for key combos (Enter, Control+A, ...) and `pz_act_click()`.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_type("Buy milk", target = "#task-title")
pz_get_value(page, target = "#task-title")

# Typing fires input events, so the page reacts: Add is now enabled
pz_expect_enabled(page, target = "#add-task")

# At the root, target = NULL types into the focused element
page |> pz_act_type(" and eggs")
```

```

pz_get_value(page, target = "#task-title")
pz_close(page)

```

pz_annotate	<i>Mark page elements</i>
-------------	---------------------------

Description

Draws an annotation for each element matched by **target**. Annotations follow their elements as the page scrolls or changes layout, and appear in screenshots and recordings until cleared with `pz_annotate_clear()`. Annotations belong to the current document; navigating away removes them. Marks respect the target's overflow containers (axis-aligned clipping only, not a custom `overflow-clip-margin`). Badges prefer a position above the visible mark's top-left corner, outside those clips. For targets inside the viewport, paparazzi shifts settled badges just enough to fit, overlapping the target if needed at the top edge. Badges larger than the viewport align with its top or left edge and may still be clipped; paparazzi does not resize them. Off-viewport targets keep their badges attached, and a screenshot's frame crop can still exclude a badge. Badges size to their labels, wrapping long labels at 20em rather than at the target's width. On sides where the target is not cut off, the clip leaves room for the mark's `pad` so its outline stays intact. A mark and its badge hide while the target is disconnected, not rendered, `visibility: hidden` with no visible descendants, or entirely clipped, even if its padding would reach into view. Zero-size targets on a clipping edge also stay hidden.

Usage

```

pz_annotate(
  ctx,
  target = NULL,
  ...,
  type = "box",
  label = NULL,
  pad = 0,
  reveal = c("auto", "fade", "draw", "pop", "slide", "wipe", "none"),
  id = NULL,
  color = NULL,
  label_fill = NULL,
  label_text_color = NULL,
  stroke_width = NULL,
  font_family = NULL,
  font_size = NULL
)

```

Arguments

ctx	A paparazzi context.
-----	----------------------

target	A selector, pz_loc() spec, or list of targets. NULL uses the current scope, or <code>document.body</code> at the root.
...	Checked empty; reserved for future use.
type	"box" (outline), "circle" (ellipse), "underline" (bottom stroke), or "highlight" (translucent fill blended over the page).
label	Optional badge. TRUE numbers the matches 1, 2, ...; one string or number repeats on each match. NULL omits the badge.
pad	Extra CSS pixels around each element: one number or <code>c(top, right, bottom, left)</code> . Defaults to zero.
reveal	"auto" (the default) uses "fade" for boxes and "draw" for other types. Other choices are "fade", "draw", "pop", "slide", "wipe", or "none". Reveals animate only during an active, unpaused recording; clearing reverses the reveal. Wipe sweeps clockwise from 12 o'clock.
id	Optional nonempty id. Reusing it replaces its annotations; NULL generates a unique id, so calls accumulate annotations. "spotlight" and "caption" are reserved for other types.
color	CSS accent color for the mark outline, or NULL for the staged default from pz_stage_annotate() .
label_fill, label_text_color	CSS colors for the badge background and text. NULL (the default) fills the badge with the mark's <code>color</code> and uses white text. The <code>fill</code> and <code>text_color</code> set by pz_stage_annotate() style callouts, not mark badges.
stroke_width	Mark stroke width in CSS pixels, or NULL for the staged <code>stroke_width</code> from pz_stage_annotate() .
font_family	CSS font family for the badge, or NULL for the staged default. A font object staged with pz_stage_fonts() is also accepted and resolves to its family with a sans-serif fallback.
font_size	Badge font size in CSS pixels, or NULL for the page default.

Value

ctx, invisibly.

See Also

[pz_annotate_clear\(\)](#), [pz_stage_annotate\(\)](#)

Description

Draws one callout per matching element, repeating **text** for every match. Callouts follow their targets during scrolling and layout changes, and appear in screenshots and recordings until cleared with `pz_annotate_clear()`. They belong to the current document and disappear on navigation. A bubble is kept within the viewport even if this moves it away from the requested side; text that cannot fit a tiny viewport is clipped. A viewport resize does not rewrap an existing callout. A callout hides while its target isn't rendered, is **visibility: hidden** with no visible descendants, or is entirely clipped by an overflow container. When the target is partly clipped, the leader points at its visible part; the bubble itself is never clipped.

Usage

```
pz_annotate_callout(
    ctx,
    text,
    ...,
    target = NULL,
    side = NULL,
    leader = TRUE,
    label = NULL,
    reveal = c("pop", "fade", "draw", "slide", "wipe", "none"),
    id = NULL,
    color = NULL,
    fill = NULL,
    text_color = NULL,
    stroke_width = NULL,
    distance = NULL,
    font_family = NULL,
    font_size = NULL
)
```

Arguments

ctx	A paparazzi context.
text	One nonempty string of literal text (not HTML).
...	Checked empty; reserved for future use.
target	A selector, <code>pz_loc()</code> spec, or list of targets. NULL uses the current scope, or <code>document.body</code> at the root.
side	A side ("top", "right", "bottom", "left") or a diagonal corner such as "top right". NULL chooses the cardinal side with the most room when drawn. The chosen side stays fixed.
leader	The line between the bubble and the target. TRUE (the default) draws a line with an arrow at the target, equal to <code>c(start = "none", end = "arrow")</code> ; FALSE draws no line, leaving a tooltip-style bubble. A named character vector sets a decoration for each end: start is the bubble side,

end is the target side, and an omitted name defaults to **"none"**. Decoration values are **"none"**, **"arrow"** (a filled triangle), **"dot"**, or **"bar"** (a perpendicular tick). The shaft stops at each decoration's base, and short leaders shrink the decorations.

label	Optional badge; TRUE numbers the matches, while one string or number repeats for each match. NULL omits the badge.
reveal	"pop" (the default), "fade" , "draw" , "slide" , "wipe" , or "none" . Animates only during an active, unpaused recording; clearing plays the reverse. "draw" draws the leader's shaft, then shows its decorations.
id	Optional nonempty id. Reusing it replaces its annotations; NULL generates a unique id, so calls accumulate annotations. "spotlight" and "caption" are reserved for other types.
color	Accent color for the bubble border, leader line and decorations; NULL uses the staged annotation color.
fill, text_color	CSS colors for the bubble background and text (and the callout's badge); NULL uses the staged fill/text_color from pz_stage_annotate() . Without a staged value, the fill follows the resolved color , and paparazzi chooses black or white text for the higher WCAG contrast ratio against the fill. Automatic contrast blends translucent fills over white, so set text_color explicitly with a translucent fill on a dark page. pz_annotate() mark badges do not use these; they follow their mark's color unless overridden per call with label_fill/label_text_color .
stroke_width	Leader line width in CSS pixels, or NULL for the staged stroke_width from pz_stage_annotate() . Decoration sizes scale with it. Shared with pz_annotate() marks.
distance	Bubble-to-target gap in CSS pixels, or NULL for the staged distance from pz_stage_annotate() . Without either, the gap is 24 with a leader and 8 without one.
font_family	CSS font family; NULL uses the staged default. A font object staged with pz_stage_fonts() is also accepted and resolves to its family with a sans-serif fallback.
font_size	Font size in CSS pixels; NULL uses the staged default.

Value

ctx, invisibly.

See Also

[pz_annotate\(\)](#), [pz_annotate_clear\(\)](#)

`pz_annotate_caption` *Add a screen-space caption*

Description

A caption stays on the page until replaced or cleared with `pz_annotate_clear(id = "caption")`. It persists through navigation and across recordings, and appears on stills. In a recording, a caption fades out when it is cleared or replaced; one that is still showing at `pz_record_stop()` stays on through the last frame.

Usage

```
pz_annotate_caption(
  ctx,
  text,
  ...,
  side = NULL,
  color = "white",
  font_family = NULL,
  font_size = 20
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>text</code>	Nonempty caption text. Newlines are preserved.
<code>...</code>	Checked empty.
<code>side</code>	Placement: "top" or "bottom" (the default). NULL (the default) uses the staged <code>caption_side</code> from <code>pz_stage_annotate()</code> (initially "bottom").
<code>color</code>	Text color; defaults to "white", independent of the staged annotation accent.
<code>font_family</code>	CSS font family. NULL uses the page's <code>font_family</code> setting in <code>pz_stage_annotate()</code> (initially sans-serif). A font object staged with <code>pz_stage_fonts()</code> is also accepted and resolves to its family with a sans-serif fallback.
<code>font_size</code>	Font size in CSS pixels; defaults to 20, independent of the annotation badge size. Captions scale with the output.

Value

`ctx`, invisibly.

See Also

[pz_annotate_clear\(\)](#), [pz_record_start\(\)](#), [pz_screenshot\(\)](#)

`pz_annotate_clear` *Clear page annotations*

Description

Remove an annotation by `id`, or remove every annotation when `id` is `NULL`. During an active recording, marks play their reveal in reverse before removal; paused recordings and stills clear instantly.

Usage

```
pz_annotate_clear(ctx, id = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>id</code>	Id to clear; <code>NULL</code> clears all. Reserved ids are allowed.
<code>...</code>	Checked empty; reserved for future use.

Value

`ctx`, invisibly.

`pz_annotate_redact` *Redact page elements*

Description

Covers every element matched by `target` with an instant opaque fill or backdrop blur in screenshots and recordings. Fill is the safe choice for secrets: blur can leave text partly legible. Redactions follow elements as they move and respect axis-aligned overflow clipping, except custom `overflow-clip-margin`. If a target disconnects, stops rendering, or is entirely clipped, its redaction hides until it becomes visible again. They belong to the current document and are lost on navigation. Redaction covers each element's border box plus `pad`, not overflowing descendants; target the overflowing element or add padding. Later top-layer UI (modal dialogs and popovers) paints above redactions, so targets inside an open modal or popover are rejected.

Usage

```
pz_annotate_redact(
  ctx,
  target = NULL,
  ...,
  method = c("fill", "blur"),
```

```

    pad = 0,
    id = NULL,
    color = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A selector, pz_loc() spec, or list of targets. NULL uses the current scope, or <code>document.body</code> at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>method</code>	"fill" (default) or "blur" (32 CSS px backdrop blur).
<code>pad</code>	Extra CSS pixels around each element; defaults to zero.
<code>id</code>	Optional nonempty id; reusing it replaces the old annotation. NULL generates a unique id, so calls accumulate annotations. "spotlight" and "caption" are reserved.
<code>color</code>	CSS color for fill; NULL uses near-black (#171717), not the staged mark color. A near-black opaque base stays underneath even when a custom color is translucent or invalid. For blur, only NULL is allowed.

Value

`ctx`, invisibly.

See Also

[pz_annotate\(\)](#), [pz_annotate_clear\(\)](#)

`pz_annotate_spotlight`

Spotlight page elements

Description

Dims the page except for a rounded cutout around each matched element. The scrim covers the document, including below-fold areas captured in screenshots. The spotlight follows its elements through scrolling and layout changes. Disconnected or zero-size elements lose their cutouts, and so do **visibility: hidden** elements with no visible descendants. Cutouts are clipped by the target's overflow containers (axis-aligned clipping only, not a custom **overflow-clip-margin**), so a target scrolled out of a scrolling container gets no cutout. It appears in screenshots and recordings until cleared, and is lost on navigation. Only one spotlight exists per page: a new call replaces the previous one, and [pz_annotate_clear\(\)](#) removes it by the reserved id "spotlight" or with a clear-all call. The cutout covers each target's padded border box; overflowing descendants and top-layer dialogs/popovers are not covered.

Usage

```

pz_annotate_spotlight(
  ctx,
  target = NULL,
  ...,
  pad = 0,
  dim = 0.6,
  reveal = c("fade", "none")
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A selector, pz_loc() spec, or list of targets. <code>NULL</code> uses the current scope, or <code>document.body</code> at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>pad</code>	Extra CSS pixels around each element: one number or <code>c(top, right, bottom, left)</code> . Defaults to zero.
<code>dim</code>	Opacity of the black overlay outside the cutouts, from 0 (transparent) to 1 (black). Defaults to 0.6.
<code>reveal</code>	"fade" (default) or "none". A fade plays only during an active, unpaused recording, and reverses on clear. Otherwise drawing and clearing are instant.

Value

`ctx`, invisibly.

See Also

[pz_annotate\(\)](#), [pz_annotate_clear\(\)](#)

`pz_camera`

Move the recording camera

Description

Moves an encode-time camera over a recording. The shot comes from **frame**: a [pz_frame\(\)](#) spec, or a bare locator (a CSS selector string, a [pz_loc\(\)](#) spec, or a list of either) promoted to one, whose union is the shot. In a scoped context **frame** may be omitted or `NULL` to shoot the scope's box; at the root it is required ([pz_camera_reset\(\)](#) returns to the home view). **frame** = `FALSE` shoots the scope's box, or the viewport at the root, unframed. The camera does not change the live page or still screenshots. Outside a recording it does nothing, and each [pz_record_start\(\)](#) begins at the recording frame.

Usage

```
pz_camera(ctx, frame, ..., duration = NULL, wait = FALSE)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>frame</code>	The shot's what-and-how: a <code>pz_frame()</code> spec or a bare locator promoted to one. Camera shots never inherit the staged frame: unset fields use the camera defaults (<code>pad = 24</code> , <code>target_box = "element"</code> , everything else as in <code>pz_frame()</code>). <code>ratio</code> and <code>when</code> are ignored, with a warning when set. <code>zoom</code> is magnification relative to the recording frame: <code>NULL</code> fits the padded target, capped at the capture's pixel density; a number fixes the shot at the home frame divided by <code>zoom</code> . <code>anchor</code> places the padded target in the shot. <code>frame</code> is required at the root. In a scoped context, leaving it out (or <code>NULL</code>) shoots the scope's box. A spec without a target also frames the scope's box, or the viewport at the root, so <code>pz_frame(zoom = 2)</code> zooms in on the middle of the view. <code>FALSE</code> shoots the scope's box or the viewport with no padding.
<code>...</code>	Checked empty; reserved for future use.
<code>duration</code>	Movement duration in seconds. <code>NULL</code> chooses a duration based on the pan and zoom distance. When the move wouldn't change the view, <code>NULL</code> takes the new shot instantly, adding no video time, so the camera follows the new target if the page scrolls later. A number holds the camera still for that long.
<code>wait</code>	Whether to wait for the move to finish before the next step. <code>FALSE</code> (the default) lets the next step run while the camera moves, so a cursor glide can happen during a zoom. Either way, a camera call first lets an earlier camera move finish, so calling <code>pz_camera()</code> on the same target again settles the camera there. <code>pz_record_hold()</code> , <code>pz_record_pause()</code> and <code>pz_record_stop()</code> always let a move finish first.

Value

`ctx`, invisibly.

<code>pz_camera_reset</code>	<i>Reset the recording camera</i>
------------------------------	-----------------------------------

Description

Returns the camera to the recording frame (or the full viewport). Outside a recording it does nothing.

Usage

```
pz_camera_reset(ctx, ..., wait = FALSE)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>wait</code>	Whether to wait for the move to finish before the next step. <code>FALSE</code> (the default) lets the next step run while the camera moves, so a cursor glide can happen during a zoom. Either way, a camera call first lets an earlier camera move finish, so calling <code>pz_camera()</code> on the same target again settles the camera there. <code>pz_record_hold()</code> , <code>pz_record_pause()</code> and <code>pz_record_stop()</code> always let a move finish first.

Value

`ctx`, invisibly.

<code>pz_chromote</code>	<i>Get the underlying ChromoteSession</i>
--------------------------	---

Description

Escape hatch for raw Chrome DevTools Protocol calls.

Usage

```
pz_chromote(ctx)
```

Arguments

<code>ctx</code>	A paparazzi context.
------------------	----------------------

Value

The `chromote::ChromoteSession` backing the page.

Examples

```
page <- pz_open(pz_example("tasks"))
session <- pz_chromote(page)

# Make raw Chrome DevTools Protocol calls
session$Browser$getVersion()$product
session$Performance$enable()
metrics <- session$Performance$getMetrics()$metrics
Filter(function(m) m$name == "Nodes", metrics)[[1]]$value
pz_close(page)
```

pz_close	<i>Close a page</i>
----------	---------------------

Description

Closes the page's browser session and any server it started. A page opened from a shared serving handle leaves the server running. Closing an already-closed page is a no-op.

Usage

```
pz_close(page)
```

Arguments

page	A PaparazziPage from <code>pz_open()</code> , or any context on it (such as the end of a chain). A chain from <code>pz_record_start()</code> whose recording is still running stops and writes it first.
------	--

Value

The page, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_close(page)

# Closing a closed page does nothing
pz_close(page)
```

pz_cursor_hide	<i>Hide the overlay cursor</i>
----------------	--------------------------------

Description

Hides the cursor set up by `pz_cursor_show()` or shown implicitly while recording. The cursor keeps its last position; showing it again returns it there. The hide is explicit and sticky: the cursor stays hidden until `pz_cursor_show()` or `pz_cursor_move()` shows it again – neither a recorded pointer action nor the `cursor` staging setting brings it back. That is what distinguishes `pz_cursor_hide()` from `pz_cursor_leave()`, which keeps the cursor visible and glides back in on the next action.

Usage

```
pz_cursor_hide(ctx, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.

Value

`ctx`, invisibly.

See Also

[pz_cursor_show\(\)](#), [pz_cursor_leave\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_cursor_show("#add-task") |>
  pz_cursor_hide() |>
  pz_screenshot(file.path(tempdir(), "no-cursor.png"))
pz_screenshot(page, frame = pz_frame("#new-task", pad = 24))
pz_close(page)
```

<code>pz_cursor_leave</code>	<i>Move the overlay cursor out of the frame</i>
------------------------------	---

Description

Sends the cursor out of the frame through `side`: while recording it glides out, otherwise it leaves immediately. The cursor is still visible (just off-frame); the next pointer action or cursor call glides it back in from that side.

Usage

```
pz_cursor_leave(ctx, side = "right", icon = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>side</code>	A side (" <code>top</code> ", " <code>bottom</code> ", " <code>left</code> ", " <code>right</code> ") or corner (" <code>top left</code> ", " <code>bottom right</code> ", ...) of the frame to leave through. Defaults to " <code>right</code> ".
<code>icon</code>	CSS cursor keyword to show during the exit. <code>NULL</code> (the default) keeps the icon already on the cursor: there is nothing to infer at the off-frame exit point. The exit icon becomes the last visible icon, so the next entrance starts with it.

Value

ctx, invisibly.

See Also

[pz_cursor_show\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "help.mp4")
page |>
  pz_record(path, {
    page |>
      pz_act_click("#toggle-help") |>
        # Move the cursor out of the way so the help text is unobstructed
        pz_cursor_leave("right", icon = "default") |>
        pz_record_hold(1)
  })
pz_close(page)
```

pz_cursor_move	<i>Move the overlay cursor to an element</i>
----------------	--

Description

Moves the cursor over **target**, showing it first if hidden. While recording the move is a glide whose duration scales with distance (see [pz_stage\(\)](#)); pass **duration** to override it. Without a recording the cursor jumps.

Usage

```
pz_cursor_move(
  ctx,
  target,
  ...,
  duration = NULL,
  icon = NULL,
  offset = c(0, 0)
)
```

Arguments

ctx	A paparazzi context.
target	A CSS selector string, a pz_loc() spec, or a list of specs. The cursor centers on the match.

...	Checked empty; reserved for future use.
duration	Glide duration in seconds; NULL computes one from the distance and the <code>cursor_speed</code> staging setting.
icon	CSS cursor keyword to show for this call. NULL (the default) infers the icon at the actual landing point: the first computed <code>cursor</code> other than <code>auto</code> on the element under the point or an ancestor. Every CSS cursor keyword is supported: for example, <code>default</code> , <code>pointer</code> , <code>text</code> , <code>crosshair</code> , <code>wait</code> , <code>zoom-in</code> , and the resize keywords. <code>none</code> hides the artwork without changing the overlay's position; an explicit <code>auto</code> shows the default arrow. A bare <code>url()</code> uses <code>default</code> ; a <code>url()</code> with a supported keyword fallback uses that keyword. Custom URL images are not drawn. An explicit icon lasts through this call's landing but does not override the next call's automatic inference. The selected icon stays on the cursor until the next destination; the first off-frame entrance starts with <code>default</code> unless overridden. While recording, an automatic move keeps the icon already visible until it enters the destination and switches there; an explicit <code>icon</code> applies from the start of the glide and stays through its landing and any press. The artwork tracks CSS zoom and the device pixel ratio internally, and a navigation re-injects the overlay with its last icon.
offset	Landing offset in viewport CSS pixels, <code>c(x, y)</code> with positive x to the right and positive y downward. A single number is recycled to both axes; the default <code>c(0, 0)</code> adds no offset. It applies only to this call and only to the drawn overlay, not to page pointer events. The cursor may land outside the viewport without clamping.

Value

`ctx`, invisibly.

See Also

[pz_cursor_show\(\)](#), [pz_cursor_leave\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "tour.mp4")

# While recording, the cursor glides between elements
page |>
  pz_record(path, {
    page |>
      pz_cursor_show(".filters", from = "left") |>
      pz_cursor_move("#toggle-help", duration = 1) |>
      pz_cursor_move("#add-task", icon = "crosshair")
  })
pz_close(page)
```

`pz_cursor_show` *Show the overlay cursor*

Description

Shows paparazzi's overlay cursor, optionally over a `target` element. While recording, the cursor appears with staging: it glides from its last position, fades in on the target when it has never been shown, or glides in from `from` (or the `enter` side set with `pz_stage()`) when entering the frame. Without a recording the cursor appears statically, which is how a cursor lands in a screenshot. Its default size is 1.75 times the original artwork; set `cursor_scale` with `pz_stage()` to change the size in videos and stills.

Usage

```
pz_cursor_show(ctx, target = NULL, ..., from = NULL, icon = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs. The cursor centers on the match. NULL uses the current scope's element or, at the root context, shows the cursor at its last position (or the viewport center the first time).
<code>...</code>	Checked empty; reserved for future use.
<code>from</code>	A side (" <code>top</code> ", " <code>bottom</code> ", " <code>left</code> ", " <code>right</code> ") or corner (" <code>top left</code> ", " <code>bottom right</code> ", ...) of the frame to enter from. NULL (the default) re-enters from the direction the cursor last left through (see <code>pz_cursor_leave()</code>); a cursor that has never been shown uses the <code>enter</code> setting of <code>pz_stage()</code> . Otherwise the cursor glides from its current position.
<code>icon</code>	CSS cursor keyword to show for this call. NULL (the default) infers the icon at the actual landing point: the first computed <code>cursor</code> other than <code>auto</code> on the element under the point or an ancestor. Every CSS cursor keyword is supported: for example, <code>default</code> , <code>pointer</code> , <code>text</code> , <code>crosshair</code> , <code>wait</code> , <code>zoom-in</code> , and the resize keywords. <code>none</code> hides the artwork without changing the overlay's position; an explicit <code>auto</code> shows the default arrow. A bare <code>url()</code> uses <code>default</code> ; a <code>url()</code> with a supported keyword fallback uses that keyword. Custom URL images are not drawn. An explicit icon lasts through this call's landing but does not override the next call's automatic inference. The selected icon stays on the cursor until the next destination; the first off-frame entrance starts with <code>default</code> unless overridden. While recording, an automatic move keeps the icon already visible until it enters the destination and switches there; an explicit <code>icon</code> applies from the start of the glide and stays through its landing and any press. The artwork tracks CSS zoom and the device pixel ratio internally, and a navigation re-injects the overlay with its last icon.

Value

ctx, invisibly.

See Also

[pz_stage\(\)](#) for cursor settings, [pz_cursor_hide\(\)](#), [pz_cursor_move\(\)](#), [pz_cursor_leave\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "add-button.png")

# Outside a recording the cursor appears at once, ready for a still
page |>
  pz_cursor_show("#add-task", icon = "pointer") |>
  pz_screenshot(path, frame = pz_frame("#new-task", pad = 24))
pz_screenshot(page, frame = pz_frame("#new-task", pad = 24))
pz_close(page)
```

pz_device

Set the page's size and display settings

Description

Configures device emulation for the page: viewport size, device scale factor, mobile-ness, zoom, color scheme, reduced motion, locale and time zone. Only **supplied** arguments change state: NULL (the default) means "leave this setting alone", so `zoom = 1`, not `zoom = NULL`, disables an active zoom.

Usage

```
pz_device(
  ctx,
  ...,
  width = NULL,
  height = NULL,
  scale = NULL,
  mobile = NULL,
  zoom = NULL,
  zoom_method = NULL,
  color_scheme = NULL,
  reduced_motion = NULL,
  locale = NULL,
  timezone = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>width, height</code>	Viewport size in CSS pixels. Only the supplied one changes; the other keeps its current value.
<code>scale</code>	Device scale factor: how many screen pixels each CSS pixel covers. <code>NULL</code> keeps the current value. Until you set it, the page renders at the browser's own scale (1), but once you set <code>width</code> , <code>height</code> , or <code>mobile</code> , it switches to 2 for sharp, retina-quality captures. Zooming with the default <code>zoom_method</code> multiplies the scale by <code>zoom</code> .
<code>mobile</code>	Emulate a mobile device (touch input, mobile viewport semantics)? <code>NULL</code> retains the stored value, or uses <code>FALSE</code> when unset.
<code>zoom</code>	Zoom factor; 1 disables zoom.
<code>zoom_method</code>	"viewport" or "css"; see above. <code>NULL</code> retains the stored method, or uses "viewport" when unset. Used when a <code>zoom</code> is (or later becomes) active.
<code>color_scheme</code>	"light" or "dark" for <code>prefers-color-scheme</code> .
<code>reduced_motion</code>	Emulate <code>prefers-reduced-motion</code> ? <code>TRUE</code> maps to <code>reduce</code> , <code>FALSE</code> to <code>no-preference</code> .
<code>locale</code>	An ICU locale, e.g. "de-DE".
<code>timezone</code>	An IANA time zone, e.g. "Pacific/Auckland".

Details

The two zoom methods trade off differently:

- `zoom_method = "viewport"` (the default) shrinks the CSS viewport and raises the device scale factor, so `vh` stays correct but media queries can start (or stop) matching.
- `zoom_method = "css"` applies a CSS `zoom` to the page instead – on every document, so it survives navigation and reload: layout and media queries are untouched, but `vh`-sized elements no longer fit the viewport (a `50vh` element covers the whole viewport at `zoom 2`).

`reduced_motion = TRUE` gives deterministic stills (animations stop); it's opt-in because it's usually wrong for videos.

Value

`ctx`, invisibly.

See Also

[pz_open\(\)](#) forwards its ... here.

Examples

```
page <- pz_open(pz_example("tasks"))

# Only the settings you supply change
page |> pz_device(width = 390, height = 844, mobile = TRUE)
pz_js(page, "[window.innerWidth, window.devicePixelRatio]")

page |> pz_device(color_scheme = "dark")
pz_get_style(page, "background-color", target = "body")
pz_close(page)
```

pz_example

Paths to example pages and apps

Description

paparazzi ships a few small pages and apps for its examples and articles. `pz_example()` returns the path to one of them, ready to pass to `pz_open()` or `pz_serve_shiny()`. The examples are:

Usage

```
pz_example(name = NULL)
```

Arguments

name	The name of an example, without a file extension. <code>NULL</code> returns the names of all examples.
-------------	--

Details

- **"tasks"**: a static task tracker page (an HTML file) with a form, a scrollable list of draggable tasks, filter links that change the URL fragment, inline title editing, an attachment input for new tasks, a help panel that starts hidden, and a "Start over" link that loads the page again. Adding a task shows "Saving..." for 400ms before the task appears.
- **"tasks-app"**: a Shiny app directory with a text input, a selectize priority input, an Add button, and a task list whose output takes half a second to render. Running it requires the shiny package.

Value

The absolute path to the example's file or directory, or a character vector of example names when `name` is `NULL`.

Examples

```

pz_example()
pz_example("tasks")

page <- pz_open(pz_example("tasks"))
pz_get_title(page)
pz_close(page)

```

pz_expect_attr	<i>Expect attributes</i>
----------------	--------------------------

Description

`pz_expect_attr()` passes when at least one element matches and every matched element satisfies every named attribute/value pair in Missing attributes satisfy no pair. Comparisons are exact by default; `.match = "contains"` or `"regex"` applies to all pairs in the call.

A length-1 value applies to every element. A vector requires exactly that many matches and compares pairwise, in order. With `.not = TRUE`, the expectation passes when the combined condition does not hold, including when no element matches.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```

pz_expect_attr(
  ctx,
  .target = NULL,
  ...,
  .match = c("exact", "contains", "regex"),
  .not = FALSE,
  .timeout = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>.target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of either. <code>NULL</code> selects the current context.
<code>...</code>	Named attribute/value pairs. Values are character vectors; use back-ticks for names such as <code>aria-expanded</code> . Dynamic dots support splicing a named list with <code>!!!</code> .
<code>.match</code>	Comparison mode for all pairs: <code>"exact"</code> (default), <code>"contains"</code> , or <code>"regex"</code> .

`.not` Invert the combined expectation?
`.timeout` Seconds to wait; NULL uses the session default.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_attr(
  pz_loc(".task", has_text = "tax"), `data-priority` = "high"
)

page |>
  pz_act_click("#toggle-help") |>
  pz_expect_attr("#toggle-help", `aria-expanded` = "true")
pz_close(page)
```

<code>pz_expect_checked</code>	<i>Expect elements to be checked</i>
--------------------------------	--------------------------------------

Description

`pz_expect_checked()` passes when at least one element matches and every match is checked (the browser's `:checked` selector, so checkboxes, radios, and select options all count). Toggle with `pz_act_click()` or `pz_set_value()`.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_checked(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). NULL means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; NULL (default) uses the session default, 0 checks once.

Value

ctx, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_checked(target = "#task-urgent", not = TRUE)
page |>
  pz_act_click("#task-urgent") |>
  pz_expect_checked(target = "#task-urgent")
pz_close(page)
```

pz_expect_class	<i>Expect a class</i>
-----------------	-----------------------

Description

`pz_expect_class()` passes when at least one element matches and every match carries `class`. With `not = TRUE` it passes when no match does, including when nothing matches. `class` is one class name, not a space-separated list: expect each class with its own call.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_class(ctx, class, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>class</code>	A single class name to expect on every match.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

ctx, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
water <- pz_loc(".task", has_text = "Water")
page |> pz_expect_class("done", target = water, not = TRUE)

page |>
  pz_find(water) |>
    pz_act_click(".task-done") |>
      pz_expect_class("done")
pz_close(page)
```

pz_expect_count	<i>Expect a number of matching elements</i>
-----------------	---

Description

`pz_expect_count()` passes when the number of elements matching `target` satisfies the requirement: `n` is exact, or `min` and/or `max` give an inclusive range (either may be `NULL`, meaning unbounded). With `not = TRUE` it passes when the count does anything else. Specify exactly one of `n` or `min/max`.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_count(
  ctx,
  n = NULL,
  target = NULL,
  ...,
  min = NULL,
  max = NULL,
  not = FALSE,
  timeout = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>n</code>	Exact expected count. Exclusive with <code>min</code> and <code>max</code> . <code>NULL</code> disables the exact-count check; supply <code>min</code> or <code>max</code> .
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.

...	Checked empty; reserved for future use.
min	Minimum count; with <code>max</code> , an inclusive range check. <code>NULL</code> omits the lower bound.
max	Maximum count; with <code>min</code> , an inclusive range check. <code>NULL</code> omits the upper bound.
not	Invert the check.
timeout	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_count(7, target = ".task")
page |> pz_expect_count(min = 1, max = 2, target = ".task.done")

# Expectations retry, so they wait for the page to catch up: the new task
# appears after a short "Saving..." delay
page |>
  pz_act_type("Buy milk", target = "#task-title") |>
  pz_act_click("#add-task") |>
  pz_expect_count(8, target = ".task")
pz_close(page)
```

<code>pz_expect_enabled</code>	<i>Expect elements to be enabled</i>
--------------------------------	--------------------------------------

Description

`pz_expect_enabled()` passes when at least one element matches and every match is enabled. An element counts as disabled when it matches the browser's `:disabled` selector, so inputs inside a disabled `<fieldset>` are disabled too.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_enabled(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). NULL means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; NULL (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))

# Add is disabled until the title has text
page |> pz_expect_enabled(target = "#add-task", not = TRUE)
page |>
  pz_act_type("Buy milk", target = "#task-title") |>
  pz_expect_enabled(target = "#add-task")
pz_close(page)
```

<code>pz_expect_exists</code>	<i>Expect at least one element to match</i>
-------------------------------	---

Description

`pz_expect_exists()` passes when at least one element matching `target` is in the DOM, visible or not. It's the one expectation where multiple matches don't all have to satisfy the check: existence needs only one. With `not = TRUE` it passes when nothing matches.

Expectations retry until they pass or `timeout` elapses, then return `ctx` invisibly. Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"` showing the target, the last observed value, and the time waited; inside `testthat`, the failure is instead reported as a test failure, and a pass counts as a successful `testthat` expectation.

Usage

```
pz_expect_exists(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_exists(target = ".task")

# The help panel is in the page even while it's hidden
page |> pz_expect_exists(target = "#help")
page |> pz_expect_exists(target = ".error-message", not = TRUE)

# A failing expectation retries until the timeout, then errors
try(pz_expect_exists(page, target = ".error-message", timeout = 0.5))
pz_close(page)
```

`pz_expect_focused` *Expect elements to be focused*

Description

`pz_expect_focused()` passes when at least one element matches and every match is the page's focused element (`document.activeElement`). Focus it with `pz_act_focus()` or a `pz_act_click()`, and remove it with `pz_act_blur()`.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_focused(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_act_click("#task-title") |>
  pz_expect_focused(target = "#task-title")

page |>
  pz_act_press("Tab") |>
  pz_expect_focused(target = "#task-priority")
pz_close(page)
```

`pz_expect_in_viewport`

Expect elements to be in the viewport

Description

`pz_expect_in_viewport()` passes when at least one element matches and every match overlaps the viewport: its bounding box crosses the visible area by any amount. Elements entirely above, below, or beside the fold fail; visibility itself is a separate expectation, `pz_expect_visible()`.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_in_viewport(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"), height = 500)

# The last task is inside the scrolling list, below its visible area
last_task <- pz_loc(".task", which = "last")
page |> pz_expect_in_viewport(target = last_task, not = TRUE)
page |>
  pz_act_scroll(last_task) |>
  pz_expect_in_viewport(target = last_task)
pz_close(page)
```

<code>pz_expect_js</code>	<i>Expect a JavaScript predicate to hold</i>
---------------------------	--

Description

`pz_expect_js()` passes when at least one element matches and the predicate holds for every match: `expr` is evaluated as a function receiving the element, e.g. `"el => el.scrollTop > 0"`. It's the escape hatch for conditions the catalog doesn't cover. A predicate that throws is a JavaScript error, not a failed check.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_js(ctx, expr, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>expr</code>	A JavaScript function receiving the element, as a string, e.g. <code>"el => el.scrollTop > 0"</code> .
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))

# The predicate receives each matching element
page |> pz_expect_js("el => el.scrollHeight > el.clientHeight", target = ".task-list")
page |> pz_expect_js("el => el.draggable", target = ".task")
pz_close(page)
```

<code>pz_expect_style</code>	<i>Expect computed styles</i>
------------------------------	-------------------------------

Description

`pz_expect_style()` passes when at least one element matches `.target` and every match has every named property set to the expected value, comparing **computed** styles – what the browser actually applied, the same values Playwright’s `toHaveCSS()` and jQuery’s `.css()` read.

Property names accept `snake_case`, converted to the CSS kebab-case spelling (`font_size` becomes `font-size`); custom properties (``--bs-primary` = "#0d6efd"`) pass through unchanged. By default the expected values are normalized in the browser: what you write (`"1.5rem"`, `"#0d6efd"`, `"50%"`, `"currentColor"`) is resolved against the same context the target sees, so it matches the computed value the browser reports. Numeric pixel values compare with a 0.5px tolerance.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```

pz_expect_style(
  ctx,
  .target = NULL,
  ...,
  .not = FALSE,
  .timeout = NULL,
  .normalize = TRUE
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>.target</code>	A CSS selector string, a pz_loc() spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root.
<code>...</code>	Property/value pairs, e.g. <code>color = "red"</code> . Dynamic dots: a list spliced in with <code>!!!</code> works. Names are the CSS property names, <code>snake_case</code> accepted.
<code>.not</code>	Invert the combined expectation?
<code>.timeout</code>	Seconds to wait; <code>NULL</code> uses the session default.
<code>.normalize</code>	Normalize expected values in the browser before comparing? See Details.

Details

With `.not = TRUE` the expectation passes when at least one property/value pair doesn't match (including when nothing matches). With `.normalize = FALSE` the expected values are compared as raw strings against the browser's computed output, for contexts the probe can't reproduce (unusual % cases, container query units); the pixel tolerance still applies.

Shorthand properties (`margin`, `border`, `background`, ...) are unreliable in computed styles and error with a longhand suggestion; check the longhand properties instead. A value the browser rejects errors immediately as invalid CSS, without retrying (only detected with `.normalize = TRUE`).

Value

`ctx`, invisibly.

Examples

```

page <- pz_open(pz_example("tasks"))

# Expected values are written as you'd write them in CSS
page |> pz_expect_style("#add-task", background_color = "#0d6efd")
page |> pz_expect_style("h1", font_size = "1.5rem")
page |> pz_expect_style("#help", display = "none")

```

```

page |>
  pz_act_click("#toggle-help") |>
    pz_expect_style("#help", display = "none", .not = TRUE)
pz_close(page)

```

pz_expect_text	<i>Expect element text content</i>
----------------	------------------------------------

Description

`pz_expect_text()` passes when at least one element matches and the text of every match satisfies `text`. Whitespace collapses on both sides before comparing, so "Save now" matches text reading "Save now".

A length-1 `text` applies to every match. A length-`n` `text` requires exactly `n` matches and compares pairwise, in order. With `not = TRUE`, the expectation passes when no match satisfies `text`, including when nothing matches.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```

pz_expect_text(
  ctx,
  text,
  target = NULL,
  ...,
  match = c("contains", "exact", "regex"),
  not = FALSE,
  timeout = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>text</code>	A character vector of expected text: length 1 applies to every match, length <code>n</code> is compared pairwise in order.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root, so <code>pz_expect_text(page, "Welcome")</code> checks the page text.
<code>...</code>	Checked empty; reserved for future use.
<code>match</code>	How to compare <code>text</code> : "contains" (substring), "exact", or "regex" (an R regex matched with <code>grepl()</code>).
<code>not</code>	Invert the check.

timeout Seconds to wait for the expectation to pass; `NULL` (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_text("Tasks", target = "h1", match = "exact")
page |> pz_expect_text("passport", target = pz_loc(".task", which = "first"))

# A vector compares pairwise with the matches, in order
page |> pz_expect_text(c("All", "Open", "Done"), target = ".filters a")

# Regular expressions use R's syntax
page |> pz_expect_text("[A-Z]", target = ".task-title", match = "regex")

# On failure, the error shows the target and the last text seen
try(pz_expect_text(page, "otters", target = "h1", timeout = 0.5))
pz_close(page)
```

<code>pz_expect_title</code>	<i>Expect the page title</i>
------------------------------	------------------------------

Description

`pz_expect_title()` passes when the page's `<title>` satisfies `title`. It works from any context, scoped or root: the title belongs to the page, not to an element.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_title(
  ctx,
  title,
  ...,
  match = c("contains", "exact", "regex"),
  not = FALSE,
  timeout = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>title</code>	The expected page title, a single string.
<code>...</code>	Checked empty; reserved for future use.
<code>match</code>	How to compare <code>url</code> : "contains" (substring), "exact", or "regex" (an R regex matched with <code>grepl()</code>).
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_title("Tasks", match = "exact")
pz_close(page)
```

<code>pz_expect_url</code>	<i>Expect the page URL</i>
----------------------------	----------------------------

Description

`pz_expect_url()` passes when the page's URL satisfies `url`. It works from any context, scoped or root: the URL belongs to the page, not to an element.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_url(
  ctx,
  url,
  ...,
  match = c("contains", "exact", "regex"),
  not = FALSE,
  timeout = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>url</code>	The expected URL, a single string.
<code>...</code>	Checked empty; reserved for future use.
<code>match</code>	How to compare <code>url</code> : "contains" (substring), "exact", or "regex" (an R regex matched with <code>grepl()</code>).
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_url("tasks.html")

page |>
  pz_act_click(pz_loc(".filters a", has_text = "Done")) |>
  pz_expect_url("#done$", match = "regex")
pz_close(page)
```

<code>pz_expect_value</code>	<i>Expect element values</i>
------------------------------	------------------------------

Description

`pz_expect_value()` passes when at least one element matches and the `value` property of every match satisfies `value`. Elements without a `value` property (paragraphs, divs) read as missing and satisfy no `value`. Checkbox and radio values come from their `value` attribute (the default is "on"), whatever their checked state.

A length-1 `value` applies to every match. A length-`n` `value` requires exactly `n` matches and compares pairwise, in order. With `not = TRUE`, the expectation passes when no match satisfies `value`, including when nothing matches.

Outside of `testthat`, a failure aborts with a classed error of class "paparazzi_expectation_failure"; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```

pz_expect_value(
  ctx,
  value,
  target = NULL,
  ...,
  match = c("contains", "exact", "regex"),
  not = FALSE,
  timeout = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>value</code>	A character vector of expected values: length 1 applies to every match, length <code>n</code> is compared pairwise in order.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root, so <code>pz_expect_text(page, "Welcome")</code> checks the page text.
<code>...</code>	Checked empty; reserved for future use.
<code>match</code>	How to compare <code>text</code> : "contains" (substring), "exact", or "regex" (an R regex matched with <code>grepl()</code>).
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```

page <- pz_open(pz_example("tasks"))
page |> pz_expect_value("normal", target = "#task-priority", match = "exact")

page |>
  pz_act_type("Buy milk", target = "#task-title") |>
  pz_expect_value("milk", target = "#task-title")
pz_close(page)

```

`pz_expect_visible` *Expect elements to be visible*

Description

`pz_expect_visible()` passes when at least one element matches and every match is visible. `pz_expect_hidden()` is exactly `pz_expect_visible(not = TRUE)`: it passes when no match is visible, including when nothing matches.

Visibility follows the browser's own `checkVisibility()` (<https://developer.mozilla.org/en-US/docs/Web/API/Element/checkVisibility>) with CSS checks, so `display: none` and `visibility: hidden` anywhere up the ancestor chain count as hidden. Opacity and viewport position are not considered; use `pz_expect_in_viewport()` for position.

Outside of `testthat`, a failure aborts with a classed error of class `"paparazzi_expectation_failure"`; inside `testthat`, the failure is reported as a test failure instead. See `pz_expect_exists()` for the retry, timeout, and bridge behavior shared by all expectations.

Usage

```
pz_expect_visible(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

```
pz_expect_hidden(ctx, target = NULL, ..., not = FALSE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context (so <code>pz_expect_exists()</code> on one trivially passes while the scope is live), the page body at the root.
<code>...</code>	Checked empty; reserved for future use.
<code>not</code>	Invert the check.
<code>timeout</code>	Seconds to wait for the expectation to pass; <code>NULL</code> (default) uses the session default, 0 checks once.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_expect_hidden(target = "#help")

page |>
  pz_act_click("#toggle-help") |>
```

```

pz_expect_visible(target = "#help")

# Every match must pass, so check one task or narrow the target
page |> pz_expect_visible(target = pz_loc(".task", has_text = "passport"))
pz_close(page)

```

pz_find

Find elements and push them as the current scope

Description

pz_find() resolves **target** (auto-waiting for at least one match), pins the matched set as the current scope, and returns a new context: later calls on it operate inside that scope. Explicit targets resolve lazily among the scope's descendants at use time – re-renders within the scope are fine – and **target = NULL** means the scope itself for the calls that accept one.

The pinned set is eager: it is fixed at find time, and a later re-render that detaches any of its elements aborts with a classed error on the next use – the scope is never silently re-queried (a lazy, reusable target is what **pz_loc()** is for). Multi-match targets pin the whole set.

The stack is immutable: **pz_find*()** never mutate the context they are called on, and contexts derived from the same parent share its pinned sets.

Usage

```
pz_find(ctx, target, ..., from_root = FALSE)
```

Arguments

ctx	A paparazzi context.
target	A CSS selector string, a pz_loc() spec, or a list of specs and strings (a union matching any of them). Required: there is nothing to find without a target, so NULL is an error. To narrow an existing scope, use pz_find_first() , pz_find_last() , or pz_find_nth() without a target.
...	Checked empty; reserved for future use.
from_root	Resolve the target from the page root instead of the current scope? The new scope is still pushed on top of the stack, so pz_find_pop() returns to the previous scope.

Value

A new context.

See Also

pz_find_first(), **pz_find_pop()**, **pz_find_reset()**

Examples

```

page <- pz_open(pz_example("tasks"))

# Inside a scope, targets resolve among the scope's descendants
page |>
  pz_find(pz_loc(".task", has_text = "dentist")) |>
  pz_act_click(".task-done")
pz_get_attr(page, "class", target = pz_loc(".task", has_text = "dentist"))

# Assign a scoped context to start more than one chain from it
task_list <- pz_find(page, ".task-list")
task_list
pz_get_count(task_list, target = ".task.done")

# from_root looks outside the current scope, but still pushes a new scope
task_list |>
  pz_find("#new-task", from_root = TRUE) |>
  pz_act_type("Buy milk", target = "#task-title")
pz_get_value(page, target = "#task-title")
pz_close(page)

```

pz_find_first	<i>Find the first match and push it as the current scope</i>
---------------	--

Description

`pz_find_first()` is `pz_find()` with `which = "first"`. With a `target`, it pins the spec's first match. Without a `target`, it narrows the current scope to its first element: one eager slice of the pinned set, with no re-query and no waiting.

Usage

```
pz_find_first(ctx, target = NULL, ..., from_root = FALSE)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string or a <code>pz_loc()</code> spec. <code>NULL</code> narrows the current scope; a union can't pick one match by position, and a spec that already carries <code>which</code> is an error.
<code>...</code>	Checked empty; reserved for future use.
<code>from_root</code>	Resolve the target from the page root instead of the current scope? Requires a <code>target</code> .

Value

A new context.

See Also

[pz_find\(\)](#), [pz_find_last\(\)](#), [pz_find_nth\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_find_first(".task") |>
  pz_get_text(target = ".task-title")

# Without a target, narrow the current scope to its first element
page |>
  pz_find(".task[data-priority='normal']") |>
  pz_find_first() |>
  pz_get_text(target = ".task-title")
pz_close(page)
```

pz_find_last	<i>Find the last match and push it as the current scope</i>
--------------	---

Description

[pz_find_last\(\)](#) is [pz_find\(\)](#) with `which = "last"`. With a `target`, it pins the spec's last match. Without a `target`, it narrows the current scope to its last element: one eager slice of the pinned set, with no re-query and no waiting.

Usage

```
pz_find_last(ctx, target = NULL, ..., from_root = FALSE)
```

Arguments

ctx	A paparazzi context.
target	A CSS selector string or a pz_loc() spec. NULL narrows the current scope; a union can't pick one match by position, and a spec that already carries <code>which</code> is an error.
...	Checked empty; reserved for future use.
from_root	Resolve the target from the page root instead of the current scope? Requires a <code>target</code> .

Value

A new context.

See Also

[pz_find\(\)](#), [pz_find_first\(\)](#), [pz_find_nth\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_find_last(".task") |>
  pz_get_text(target = ".task-title")
pz_close(page)
```

pz_find_nth	<i>Find the nth match and push it as the current scope</i>
-------------	--

Description

`pz_find_nth()` is `pz_find()` with `which = n`. With a `target`, it pins the spec's nth match; an out-of-range `n` means no match, so the call keeps auto-waiting like any `pz_loc()` resolution. Without a `target`, it narrows the current scope to its nth element: one eager slice of the pinned set, and an out-of-range `n` errors immediately – the set was fixed at pin time, so there is nothing to wait for.

Usage

```
pz_find_nth(ctx, n, target = NULL, ..., from_root = FALSE)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>n</code>	The match to pick, 1-based (" <code>first</code> "/" <code>last</code> " are the <code>pz_find_first()</code> / <code>pz_find_last()</code> wrappers, not values here).
<code>target</code>	A CSS selector string or a <code>pz_loc()</code> spec. <code>NULL</code> narrows the current scope; a union can't pick one match by position, and a spec that already carries <code>which</code> is an error.
<code>...</code>	Checked empty; reserved for future use.
<code>from_root</code>	Resolve the target from the page root instead of the current scope? Requires a <code>target</code> .

Value

A new context.

See Also

`pz_find()`, `pz_find_first()`, `pz_find_last()`

Examples

```
page <- pz_open(pz_example("tasks"))

# Mark the second task as done
page |>
  pz_find_nth(2, target = ".task") |>
  pz_act_click(".task-done")
pz_get_text(page, target = pz_loc(".task.done .task-title"))
pz_close(page)
```

pz_find_pop	<i>Pop the current scope</i>
-------------	------------------------------

Description

`pz_find_pop()` returns a context one scope level up the stack. The popped pinned set is NOT released – a context derived before the pop may still hold it – and popping never mutates the context it was called on. At the root it is a no-op returning `ctx` unchanged.

Usage

```
pz_find_pop(ctx)
```

Arguments

`ctx` A paparazzi context.

Value

A new context one level up, or `ctx` itself at the root.

See Also

[pz_find\(\)](#), [pz_find_reset\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_find(".task-list") |>
  pz_find_last(".task") |>
  pz_act_click(".task-done") |>
  # Back to the whole list, to check the result
  pz_find_pop() |>
  pz_expect_count(2, target = ".task.done") |>
  pz_get_count(target = ".task.done")
pz_close(page)
```

pz_find_reset	<i>Clear all scope, back to the root</i>
---------------	--

Description

`pz_find_reset()` returns a context with an empty scope stack. No pinned set is released – contexts derived before the reset may still hold them – and resetting never mutates the context it was called on. At the root it is a no-op returning `ctx` unchanged.

Usage

```
pz_find_reset(ctx)
```

Arguments

<code>ctx</code>	A paparazzi context.
------------------	----------------------

Value

A new root context, or `ctx` itself at the root.

See Also

[pz_find\(\)](#), [pz_find_pop\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_find(".task-list") |>
  pz_find_first(".task") |>
  pz_act_click(".task-done") |>
  # The help toggle is outside the list, so start again from the root
  pz_find_reset() |>
  pz_act_click("#toggle-help") |>
  pz_expect_visible(target = "#help") |>
  pz_get_text(target = "#help")
pz_close(page)
```

pz_font_bunny	<i>Declare a font from Bunny Fonts</i>
---------------	--

Description

Declares a font face for `pz_stage_fonts()`, served by Bunny Fonts (<https://fonts.bunny.net>), a GDPR-friendly foundry that mirrors the Google Fonts collection and stylesheet format.

Usage

```
pz_font_bunny(family, weight = 400, style = "normal")
```

Arguments

family	The font family name, exactly as Google Fonts spells it, e.g. "Open Sans".
weight	Font weight, a number from 1 to 1000 (400 is regular, 700 is bold).
style	"normal" (the default) or "italic".

Value

A font object for `pz_stage_fonts()`.

See Also

`pz_stage_fonts()`, `pz_font_google()`, `pz_font_file()`

Examples

```
pz_font_bunny("Inter", style = "italic")
```

pz_font_file	<i>Declare a font from a local file</i>
--------------	---

Description

Declares a font face for `pz_stage_fonts()` from a local font file. The bytes are read by R and handed to the page directly, so the file works on `http` pages (which cannot load `file://` URLs) and is not subject to Content Security Policy font restrictions. You are responsible for the license of fonts you use.

Usage

```
pz_font_file(family, path, weight = 400, style = "normal")
```

Arguments

family	The font family name, exactly as Google Fonts spells it, e.g. "Open Sans".
path	Path to a .woff2, .woff, .ttf, or .otf file.
weight	Font weight, a number from 1 to 1000 (400 is regular, 700 is bold).
style	"normal" (the default) or "italic".

Value

A font object for `pz_stage_fonts()`.

See Also

`pz_stage_fonts()`, `pz_font_google()`, `pz_font_bunny()`

Examples

```
## Not run:
pz_font_file("My Brand Sans", "~/fonts/my-brand-sans.woff2")

## End(Not run)
```

`pz_font_google`

Declare a font from Google Fonts

Description

Declares a font face for `pz_stage_fonts()`. The family is identified by its Google Fonts name, and Chrome fetches the stylesheet and font files at staging time; nothing is downloaded by R.

Usage

```
pz_font_google(family, weight = 400, style = "normal")
```

Arguments

family	The font family name, exactly as Google Fonts spells it, e.g. "Open Sans".
weight	Font weight, a number from 1 to 1000 (400 is regular, 700 is bold).
style	"normal" (the default) or "italic".

Value

A font object for `pz_stage_fonts()`.

See Also

`pz_stage_fonts()`, `pz_font_bunny()`, `pz_font_file()`

Examples

```
pz_font_google("Inter", weight = 700)
```

pz_frame

Frame a capture region

Description

`pz_frame()` builds a lazy framing spec for a capture: the region is computed at capture time, then padded, nudged, grown to an aspect ratio, and clamped. Every capture takes one: screenshots through `pz_screenshot(frame =)`, recordings through `pz_record_start(frame =)`, and camera shots through `pz_camera(frame =)`. Wherever a frame is accepted, a bare locator (a CSS selector string, a `pz_loc()` spec, or a list of either) is promoted to `pz_frame(<locator>)`.

Every field defaults to `NULL`, meaning "inherit": at capture time an unset field takes the `pz_stage_frame()` value if one is staged, else the built-in default. The built-in defaults are `pad = 0`, `offset = c(0, 0)`, `anchor = "center"`, `when = "stop"`, and `target_box = "element"`, with `ratio`, `bounds`, `zoom`, and `target` unset. Camera shots never inherit the staged frame; their built-in defaults are `pad = 24` and `target_box = "element"`.

The framed region is computed in this order:

1. Union the bounding boxes of the target's matched elements. With `target_box = "annotated"`, include their attached annotations.
2. Expand by `pad`.
3. With `zoom = NULL`, shift by `offset`, then grow the shorter side to reach `ratio` (never shrink), placing the content by `anchor`. With a numeric `zoom`, the region is the view divided by `zoom` (the viewport for stills and recordings, the recording's home frame for camera shots), with `ratio` setting its aspect if given; the padded target is placed inside it by `anchor`, then shifted by `offset`.
4. Clamp to `bounds` and to the page's rendered area.
5. Round to whole pixels.

Usage

```
pz_frame(
  target = NULL,
  ...,
  ratio = NULL,
  pad = NULL,
  offset = NULL,
  anchor = NULL,
  bounds = NULL,
  when = NULL,
  target_box = NULL,
  zoom = NULL
)
```

Arguments

target	The element(s) the frame is computed from: a CSS selector string, a <code>pz_loc()</code> spec, or a list of either (the frame is the union of the matched elements' bounding boxes). <code>NULL</code> frames the current scope's box, or the viewport at the root.
...	Checked empty; reserved for future use.
ratio	Width/height ratio for the region, e.g. 16/9. <code>NULL</code> disables aspect-ratio expansion, keeping the region tight to its content. Ignored (with a warning) by camera shots.
pad	Padding in CSS pixels: one number for all sides, or <code>c(top, right, bottom, left)</code> . Negative values crop inside the content box.
offset	Nudge the region by <code>c(x, y)</code> pixels, applied after <code>pad</code> .
anchor	Where the content sits while the region grows around it: <code>"center"</code> , a side (<code>"top"</code> , <code>"bottom"</code> , <code>"left"</code> , <code>"right"</code>), or a corner (<code>"top left"</code> , <code>"top right"</code> , <code>"bottom left"</code> , <code>"bottom right"</code>). Case-insensitive; <code>"top right"</code> , <code>"right top"</code> , and <code>"top-right"</code> are equivalent.
bounds	A target (a CSS selector string, a <code>pz_loc()</code> spec, or a list of either) whose union box the region is clamped within. <code>NULL</code> adds no bounds target; the page's rendered area still limits the region.
when	When a recording measures the frame: <code>"stop"</code> (the default) measures against the final layout; <code>"start"</code> clips at capture start. Screenshots ignore this; camera shots warn.
target_box	<code>"element"</code> (the default) measures only matched elements. <code>"annotated"</code> also includes painted marks, callouts, and spotlight cutouts attached to those elements or their descendants; redactions and annotations on unrelated elements are excluded. Applies to stills and camera shots, not the recording's home frame.
zoom	Magnification relative to the full view (the viewport for stills and recordings, the recording's home frame for camera shots). <code>NULL</code> fits the region to the target plus <code>pad</code> , grown to <code>ratio</code> . A number fixes the region size at the view divided by <code>zoom</code> ; <code>anchor</code> places the padded target inside it. Stills crop only, never upscale: a zoomed still is a smaller PNG at the page's pixel ratio. The camera caps a fitted shot at the capture's pixel density; an explicit <code>zoom</code> past it warns at encode time.

Value

An S3 object of class `paparazzi_frame`.

Examples

```
# Frames are specs: nothing is measured until a capture uses them
pz_frame(pad = 32)
pz_frame(ratio = 16/9, pad = 24, anchor = "top")
pz_frame(".task-list", ratio = 4/3, bounds = "main")
```

```

page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "tasks.png")

# A locator target frames the union of its boxes
page |> pz_screenshot(path, frame = pz_frame(list("h1", ".filters"), pad = 16))

# Unset fields inherit the staged framing
page |>
  pz_stage_frame(pad = 16) |>
  pz_screenshot(path, frame = pz_frame("#new-task", ratio = 16/9))
pz_screenshot(page, frame = pz_frame("#new-task", ratio = 16/9))
pz_close(page)

```

pz_get_attr

Read an attribute of matching elements

Description

pz_get_attr() returns the named attribute of every element matching **target**, one entry per match. Missing attributes give NA.

Usage

```
pz_get_attr(ctx, name, target = NULL, ...)
```

Arguments

ctx	A paparazzi context.
name	The attribute name.
target	A CSS selector string, a pz_loc() spec, or a list of specs and strings (a union matching any of them). NULL means the current context: the pinned set at a scoped context, or the page body at the root.
...	Checked empty; reserved for future use.

Value

A character vector, one entry per match.

Examples

```

page <- pz_open(pz_example("tasks"))
pz_get_attr(page, "data-priority", target = ".task")

# Missing attributes are NA
pz_get_attr(page, "aria-label", target = "select, input[type='text']")
pz_close(page)

```

pz_get_count	<i>Count matching elements</i>
--------------	--------------------------------

Description

`pz_get_count()` returns the number of elements matching `target`, counted inside the current scope. Unlike the other getters it doesn't wait for a match: 0 is a valid answer, so it resolves once and returns immediately. One exception: on a scope whose pinned elements have left the page it raises `paparazzi_error_detached` instead of returning 0, because the pinned set promises a live set and is never silently re-queried.

Usage

```
pz_get_count(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, whose count comes back without a re-query, or the page body at the root.
<code>...</code>	Checked empty; reserved for future use.

Value

An integer.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_count(page, target = ".task")

# Zero is an answer, so pz_get_count() never waits
pz_get_count(page, target = ".error-message")

# Counts are relative to the current scope
page |>
  pz_find(".task-list") |>
  pz_get_count(target = ".task.done")
pz_close(page)
```

pz_get_elements	<i>Describe matching elements</i>
-----------------	-----------------------------------

Description

`pz_get_elements()` returns a summary of every element matching `target`, one row per match in match order: the lowercased tag name, the `id` and `class` attributes, and the whitespace-collapsed text. `id` and `class` are `NA` when the attribute is absent.

Usage

```
pz_get_elements(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root.
<code>...</code>	Checked empty; reserved for future use.

Value

A tibble with columns `tag`, `id`, `class`, `text`, one row per match, plus an `element` list-column of contexts scoped to each match, pinned at get time (see `pz_get_rect()`).

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_elements(page, target = "#new-task > *")
pz_close(page)
```

pz_get_html	<i>Read the HTML of matching elements</i>
-------------	---

Description

`pz_get_html()` returns the outer HTML of every element matching `target`, one entry per match.

Usage

```
pz_get_html(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root.
<code>...</code>	Checked empty; reserved for future use.

Value

A character vector, one entry per match.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_html(page, target = pz_loc(".task", which = "first"))
pz_close(page)
```

`pz_get_rect`

Read the geometry of matching elements

Description

`pz_get_rect()` returns the bounding box of every element matching `target`, one row per match in match order.

Usage

```
pz_get_rect(ctx, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root.
<code>...</code>	Checked empty; reserved for future use.

Value

A tibble with columns `x`, `y`, `width`, `height` (doubles, CSS pixels, viewport-relative), one row per match, plus an `element` list-column. Each `element` entry is a context scoped to that one match, pinned at get time, so a chain can continue from it: `rects$element[[2]] |> pz_act_hover()`.

Examples

```
page <- pz_open(pz_example("tasks"))
rects <- pz_get_rect(page, target = ".filters a")
rects

# Each row's element column is a context scoped to that match
rects$element[[3]] |>
  pz_act_click() |>
  pz_expect_url("#done")
pz_close(page)
```

pz_get_style	<i>Read computed styles of matching elements</i>
--------------	--

Description

`pz_get_style()` returns the **computed** styles of every element matching `target` – what the browser actually applied, the same values Playwright’s `toHaveCSS()` and jQuery’s `.css()` read – one row per match in match order. Like all getters it auto-waits for at least one match.

Usage

```
pz_get_style(ctx, props = NULL, target = NULL, ...)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>props</code>	A character vector of CSS property names, or <code>NULL</code> for all computed properties. With <code>NULL</code> the columns are the union of property names across matches, first-seen order (standard computed properties are the same for every element, but custom properties vary); a match that doesn’t report a property reads as <code>""</code> (empty string).
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, or the page body at the root.
<code>...</code>	Checked empty; reserved for future use.

Details

Property names accept `snake_case`, converted to the CSS kebab-case spelling (`font_size` becomes `font-size`, also the column name, so select it with `df$`font-size``); custom properties (`--bs-primary`) pass through unchanged. `props = NULL` returns every computed property the browser reports (longhands only).

Value

A tibble with one row per match, one character column per property, plus an **element** list-column: each entry is a context scoped to that one match, pinned at get time, so a chain can continue from it (see [pz_get_rect\(\)](#)).

See Also

[pz_expect_style\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_style(page, c("font-size", "font_weight"), target = "h1")

# One row per match
pz_get_style(page, "text-decoration-line", target = ".task-title")
pz_close(page)
```

pz_get_text

Read the text of matching elements

Description

[pz_get_text\(\)](#) returns the **textContent** of every element matching **target**, one entry per match. By default runs of whitespace are collapsed to single spaces and trimmed, matching [pz_expect_text\(\)](#); **raw = TRUE** returns the text exactly as the browser holds it.

Usage

```
pz_get_text(ctx, target = NULL, ..., raw = FALSE)
```

Arguments

ctx	A paparazzi context.
target	A CSS selector string, a pz_loc() spec, or a list of specs and strings (a union matching any of them). NULL means the current context: the pinned set at a scoped context, or the page body at the root.
...	Checked empty; reserved for future use.
raw	Return the text without collapsing whitespace?

Value

A character vector, one entry per match.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_text(page, target = "h1")

# One string per match
pz_get_text(page, target = ".task[data-priority='high'] .task-title")

# Whitespace is collapsed unless raw = TRUE
pz_get_text(page, target = "#help")
pz_get_text(page, target = "#help", raw = TRUE)
pz_close(page)
```

pz_get_title	<i>Read the page title</i>
--------------	----------------------------

Description

`pz_get_title()` returns the page's current title.

Usage

```
pz_get_title(ctx)
```

Arguments

`ctx` A paparazzi context.

Value

A character vector of length one.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_get_title(page)
pz_close(page)
```

pz_get_url	<i>Read the page URL</i>
------------	--------------------------

Description

`pz_get_url()` returns the page's current URL.

Usage

```
pz_get_url(ctx)
```

Arguments

`ctx` A paparazzi context.

Value

A character vector of length one.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_click(pz_loc(".filters a", has_text = "Open"))
basename(pz_get_url(page))
pz_close(page)
```

pz_get_value	<i>Read the value of matching elements</i>
--------------	--

Description

`pz_get_value()` returns the `value` property of every element matching `target`, one entry per match. Elements without a value property (non-form elements) give NA.

Usage

```
pz_get_value(ctx, target = NULL, ...)
```

Arguments

`ctx` A paparazzi context.

`target` A CSS selector string, a `pz_loc()` spec, or a list of specs and strings (a union matching any of them). `NULL` means the current context: the pinned set at a scoped context, or the page body at the root.

`...` Checked empty; reserved for future use.

Value

A character vector, one entry per match.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_type("Buy milk", target = "#task-title")
pz_get_value(page, target = list("#task-title", "#task-priority"))
pz_close(page)
```

`pz_inspect`

Inspect the current page state

Description

Prints a console summary of the page: URL, device, recording and cursor state, and – when `target` is given – the target’s matches resolved relative to the current scope **without** auto-waiting: a single resolution pass, each match shown as a short opening tag with its visibility, enabled state, and box (long lists are truncated).

The header names the context. At the root it reads `paparazzi page`. In a scoped context from `pz_find()` it reads `paparazzi scope`, with the scope stack right under it: live match counts for each level, plus a warning when pinned elements have gone stale.

`show = "screenshot"` additionally captures an annotated screenshot: dashed outlines over the scope’s elements, solid numbered outlines over the target’s matches, drawn under paparazzi’s shadow-root overlay, so they never appear in `pz_screenshot()` output. `show = "browser"` draws the same outlines in the live page, leaves them, and opens the browser via the page’s `$view()`. `show = "auto"` picks `"screenshot"` in interactive sessions and `"none"` otherwise.

Usage

```
pz_inspect(
  ctx,
  target = NULL,
  ...,
  path = NULL,
  show = c("auto", "screenshot", "browser", "none")
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> (default) omits the target section.

...	Checked empty; reserved for future use.
path	Where to write the annotated screenshot; with NULL, paparazzi writes a numbered chunk figure while knitting and a temporary file otherwise. Only used with <code>show = "screenshot"</code> .
show	How to visualize: "auto", "screenshot" (annotated capture), "browser" (outlines left in the live page), or "none".

Value

`ctx`, invisibly (visibly in an active recording chain). With `show = "screenshot"` and `path = NULL` while knitting, returns `ctx` visibly so knitr prints the summary and includes the annotated screenshot once. A subsequent action or scope change discards the pending image; a terminal getter leaves it for the next knitr print. If the context belongs to an active recording chain, knitr stops and includes the recording instead of the inspect image.

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_inspect()

# With a target: its matches, resolved inside the current scope
page |>
  pz_find(".task-list") |>
  pz_inspect(".task.done", show = "none")

# An annotated screenshot outlines the scope and the numbered matches
page |>
  pz_find(".task-list") |>
  pz_inspect(".task-done", show = "screenshot", path = file.path(tempdir(), "inspect.png"))
pz_close(page)
```

pz_js

Evaluate JavaScript in the page

Description

Evaluates `expr` in the page, optionally awaiting a returned promise, and returns the resulting value. This is the primitive all element work uses, and an escape hatch for one-off scripts. Unlike most `pz_*`() functions it **ends the chain**: it returns the value, not the context.

Usage

```
pz_js(ctx, expr, ..., await = TRUE, timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>expr</code>	A string of JavaScript to evaluate.
<code>...</code>	Checked empty; reserved for future use.
<code>await</code>	Await a promise returned by <code>expr</code> before returning its value.
<code>timeout</code>	Seconds before the evaluation fails; NULL uses the session default.

Value

The value produced by `expr` (converted to R), or NULL.

Examples

```
page <- pz_open(pz_example("tasks"))
pz_js(page, "document.querySelectorAll('.task').length")

# Arrays come back as lists
pz_js(page, "[...document.querySelectorAll('.task-priority')].map(el => el.textContent)")

# Returned promises are awaited
pz_js(page, "new Promise(resolve => setTimeout(() => resolve('done'), 100))")
pz_close(page)
```

<code>pz_loc</code>	<i>Locate elements by CSS, with text, position, and scope qualifiers</i>
---------------------	--

Description

`pz_loc()` builds a page-independent element spec: a CSS selector, optionally narrowed by required text content (`has_text`), a match position (`which`), and an ancestor scope (`within`). Specs are lazy – they resolve at use time, so they follow DOM changes between uses – and can be defined once and reused across sessions.

Wherever a spec is accepted, a bare string is promoted to `pz_loc(string)`, and a list of specs and strings is a union that matches the elements of any member.

Usage

```
pz_loc(css, ..., has_text = NULL, which = NULL, within = NULL)
```

Arguments

<code>css</code>	A CSS selector string.
<code>...</code>	Checked empty; reserved for future use.
<code>has_text</code>	Substring the element's text content must contain. <code>NULL</code> omits text filtering. Case-sensitive; whitespace collapses on both sides, so <code>has_text = "Save now"</code> matches text reading "Save now".
<code>which</code>	Pick one match by position: <code>"first"</code> , <code>"last"</code> , or a 1-based positive integer. Applied after <code>css</code> and <code>has_text</code> filtering; an out-of-range position means <i>no</i> match, not an error. <code>NULL</code> keeps all matches.
<code>within</code>	Only match descendants of elements matching this spec (a string or <code>pz_loc()</code> , itself fully qualified). If <code>within</code> matches nothing, the whole spec matches nothing. <code>NULL</code> omits the ancestor qualifier.

Value

An S3 object of class `paparazzi_loc`.

Examples

```
# A spec describes elements; nothing is looked up until you use it
pz_loc(".task")
pz_loc(".task", has_text = "passport")
pz_loc(".task", which = "last")
pz_loc(".task-done", within = pz_loc(".task", has_text = "dentist"))

page <- pz_open(pz_example("tasks"))

# Specs resolve each time they're used, so define them once and reuse them
high <- pz_loc(".task-title", within = ".task[data-priority='high']")
pz_get_text(page, target = high)

# A list of specs and strings matches the elements of any member
pz_get_count(page, target = list(".task.done", high))
pz_close(page)
```

<code>pz_nav_goto</code>	<i>Navigate the page</i>
--------------------------	--------------------------

Description

`pz_nav_goto()` navigates to `url`; `pz_nav_reload()` reloads the page; `pz_nav_back()`/`pz_nav_forward()` step through the history.

Every navigation resets the scope to the root: the returned context has an empty scope stack, and the pinned scope objects are released, so a context scoped **before** the navigation raises a detach error if used afterwards – re-scope with `pz_find()` on the returned root context. Session-level state (timeout and recording) is untouched, and staged settings carry over to the returned context.

Usage

```

pz_nav_goto(ctx, url, ..., wait = c("auto", "load", "shiny", "none"))

pz_nav_reload(ctx, ..., wait = c("auto", "load", "shiny", "none"))

pz_nav_back(ctx, ...)

pz_nav_forward(ctx, ...)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>url</code>	The URL to navigate to (any scheme, including <code>file://</code>).
<code>...</code>	Checked empty; reserved for future use.
<code>wait</code>	What to wait for before returning: "load" waits for the document to finish loading; "shiny" also waits for Shiny idle. "auto" (the default) uses "shiny" when the page was opened on an app handle or app path and lands on that app's origin, "load" otherwise. "none" returns without settling. <code>pz_nav_back()</code> and <code>pz_nav_forward()</code> take no <code>wait</code> : they wait for load only, and return at once at a history boundary, where nothing navigates. A Shiny page restored from the back/forward cache may not reconnect, so they don't wait for Shiny; call <code>pz_wait_for_shiny_idle()</code> if you need it.

Value

The root context, invisibly.

See Also

`pz_find_reset()` for scope-only resets (no navigation).

Examples

```

page <- pz_open(pz_example("tasks"))
url <- pz_get_url(page)

# The filter links change the URL fragment, so each filter is a history entry
page |> pz_nav_goto(paste0(url, "#done"))
pz_get_count(page, target = ".task:not(.hidden)")

page |> pz_nav_back()
pz_get_count(page, target = ".task:not(.hidden)")

page |> pz_nav_forward()
pz_get_url(page)

# Reloading keeps the URL, fragment included
page |> pz_nav_reload()

```

```

pz_get_url(page)
pz_close(page)

```

pz_open

Open a page

Description

Opens a URL, local file, local server, or existing [chromote::ChromoteSession](#) as a paparazzi page: the root context that starts every `|>` chain.

Usage

```

pz_open(
  x,
  ...,
  wait = c("auto", "load", "shiny", "none"),
  timeout = 10,
  shiny_options = list(),
  envvars = NULL
)

```

Arguments

x

What to open:

- a URL string (any scheme, including `file://`, `about:`, `data:`);
- a Shiny app directory or runnable app file (`app.R`, `app-*.R`, `*_app.R`, etc.);
- a `.qmd` or `.Rmd` file, or a Quarto project directory;
- a directory of static files, served over HTTP;
- an `.html` file, served over HTTP when the `httpuv` package is installed (as `file://` otherwise). The file's directory is the server root, so pages that reference assets outside it, such as `../deps/styles.css`, need an explicit `file://` URL or a handle from [pz_serve_static\(\)](#) on the right root;
- any other existing local file, opened as `file://` (use [pz_serve_static\(\)](#) to serve a page over HTTP);
- a handle from [pz_serve_shiny\(\)](#), [pz_serve_quarto\(\)](#), or [pz_serve_static\(\)](#), shared across pages;
- an existing `ChromoteSession` (wrapped as-is; nothing is navigated).

Path detection checks Shiny first (directories containing `app.R` or `server.R`, or named app files), then Quarto (`.qmd`, `.Rmd`, or a directory containing `_quarto.yml`), then static sites: other directories and, when `httpuv` is installed, `.html` files. `ui.R` and `server.R` passed alone open as files. A path starts a one-off server using the backend defaults; closing the page

	stops it. Closing a page opened from a handle leaves its server running. Shiny app objects are not supported; supply an app path or a running app's URL instead.
...	Forwarded to <code>pz_device()</code> as device settings (e.g. <code>width = 390, mobile = TRUE</code>); they must be named.
wait	What to wait for before returning. "auto" (the default) waits for Shiny idle on app paths and handles, and for load on other pages. "load" waits for the page to load. "shiny" explicitly waits for Shiny to connect and become idle; non-Shiny pages error. "none" returns without waiting. An existing <code>ChromoteSession</code> isn't navigated, so only "shiny" waits there.
timeout	Session default timeout in seconds; defaults to 10. Per-call <code>timeout = NULL</code> in waits and expectations uses this default.
shiny_options, envvars	Passed to <code>pz_serve_shiny()</code> when opening an app path. <code>envvars = NULL</code> adds no process environment overrides.

Value

A `PaparazziPage` (the root context).

Browser

`pz_open()` opens pages in chromote's default browser (`chromote::default_chromote_object()`) and starts it if none is running. A browser paparazzi starts keeps its profile in `tempdir()`, so the profile is removed when R exits, unless a `--user-data-dir` is already set with `chromote::set_chrome_args()`. A default set with `chromote::set_default_chromote_object()` is used as-is.

Examples

```
# An HTML file is served over HTTP so the page gets bfcache and other
# HTTP-only behavior
page <- pz_open(pz_example("tasks"))
pz_get_url(page)
pz_close(page)

# Named arguments in `...` set up the device before the page loads
phone <- pz_open(pz_example("tasks"), width = 390, height = 844, mobile = TRUE)
pz_js(phone, "window.innerWidth")
pz_screenshot(phone, frame = pz_frame("#new-task", pad = 16))
pz_close(phone)

# An app directory runs in a background R process that the page owns.
# pz_open() returns once Shiny has connected and gone idle.
page <- pz_open(pz_example("tasks-app"))
pz_get_text(page, target = "#summary")
pz_close(page) # also stops the app
```

pz_record	<i>Record a block of code</i>
-----------	-------------------------------

Description

The block form of `pz_record_start()`: starts recording, evaluates the embraced expression `code`, and stops and encodes on exit – including on error, so a failed run still produces the video up to the failure. Returns what `pz_record_stop()` returns, never the block's value: `ctx` invisibly when a path was given outside knitting, media while knitting, or a viewer preview when the path is `NULL` in an interactive session. Write `pz_record(code = { ... })` to omit the path.

Usage

```
pz_record(ctx, path = NULL, code, ...)
```

Arguments

ctx	A paparazzi context.
path	Output file path; the extension (<code>.mp4</code> , <code>.webm</code> , or <code>.gif</code>) selects the format. An existing file is overwritten. The directory must already exist. With <code>NULL</code> (the default) while knitting, a numbered file in the chunk's figure directory is used and the recording appears in the document; in an interactive session, a temporary file is used and shown in the viewer when recording stops. A path is required otherwise.
code	An expression to evaluate while recording.
...	Passed to <code>pz_record_start()</code> .

Value

`ctx`, invisibly, or printable media; see `pz_record_stop()`.

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "add-task.mp4")

# The recording stops and is written when the block exits, even on error
page |>
  pz_record(path, {
    page |>
      pz_act_type("Buy milk", target = "#task-title") |>
      pz_act_click("#add-task") |>
      pz_expect_count(8, target = ".task")
  }) |>
  pz_screenshot(file.path(tempdir(), "after.png"))
file.exists(path)
```

```
pz_close(page)
```

pz_record_hold	<i>Hold the current frame while recording</i>
----------------	---

Description

Lingers on the current frame for **seconds** in the output video. Unlike `pz_wait()`, the hold advances only the video clock. Captions, annotations, and keystroke callouts added afterward appear after the held segment. When the page is not recording this is a no-op, so debugging a chain with the recording commented out doesn't pay for video-only pauses. A camera move still in progress (see `pz_camera()`) finishes before the hold begins.

Usage

```
pz_record_hold(ctx, seconds)
```

Arguments

ctx	A paparazzi context.
seconds	Seconds to hold the frame in the output.

Value

ctx, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))

# Without a recording, pz_record_hold() returns straight away
system.time(pz_record_hold(page, 2))

path <- file.path(tempdir(), "help.mp4")
page |>
  pz_record_start(path) |>
  pz_act_click("#toggle-help") |>
  # Give viewers a second to read the help text
  pz_record_hold(1) |>
  pz_record_stop()
pz_close(page)
```

pz_record_pause	<i>Pause and resume a recording</i>
-----------------	-------------------------------------

Description

`pz_record_pause()` cuts the stretch up to the next `pz_record_resume()` out of the recording: no frames are captured and the video clock stops, so the pause leaves no trace in the output.

Usage

```
pz_record_pause(ctx)

pz_record_resume(ctx)
```

Arguments

<code>ctx</code>	A paparazzi context.
------------------	----------------------

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "add-task.mp4")
page |>
  pz_record_start(path) |>
  pz_act_click("#task-title") |>
  pz_record_pause() |>
  # Filling in the form is cut from the video
  pz_set_value("Buy milk", target = "#task-title") |>
  pz_set_value("high", target = "#task-priority") |>
  pz_record_resume() |>
  pz_act_click("#add-task") |>
  pz_expect_count(8, target = ".task") |>
  pz_record_stop()
pz_close(page)
```

pz_record_start	<i>Record a page interaction</i>
-----------------	----------------------------------

Description

`pz_record_start()` begins recording the page to a video or GIF; `pz_record_stop()` ends the recording, encodes it, and writes the file. `pz_record_pause()/pz_record_resume()` cut stretches out of the recording, and `pz_record_hold()` lingers on the current frame. `pz_record()` is the block form: it records for the duration of an expression and stops on exit, including on error.

The output format comes from the `path` extension: `.mp4` (h264), `.webm`, or `.gif`. Both methods produce a video at the requested `fps`; when a new frame is not available, the previous frame is repeated.

Usage

```
pz_record_start(
  ctx,
  path = NULL,
  ...,
  method = c("poll", "screencast"),
  frame = NULL,
  fps = 15,
  scale = NULL,
  hold = c(0.5, 1),
  keep_frames = FALSE,
  captions = c("burn", "vtt", "both"),
  format = c("auto", "mp4", "webm", "gif")
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>path</code>	Output file path; the extension (<code>.mp4</code> , <code>.webm</code> , or <code>.gif</code>) selects the format. An existing file is overwritten. The directory must already exist. With <code>NULL</code> (the default) while knitting, a numbered file in the chunk's figure directory is used and the recording appears in the document; in an interactive session, a temporary file is used and shown in the viewer when recording stops. A path is required otherwise.
<code>...</code>	Checked empty; reserved for future use.
<code>method</code>	Capture method: <code>"poll"</code> (the default) for regular, higher-resolution captures, or <code>"screencast"</code> for captures driven by visual changes. See <i>Choosing a capture method</i> for tradeoffs.
<code>frame</code>	Area to show in the finished recording: a <code>pz_frame()</code> spec, or a bare locator (a CSS selector string, a <code>pz_loc()</code> spec, or a list of either) promoted to one; <code>NULL</code> uses the staged default set with <code>pz_stage_frame()</code> , if any,

	or shows the full viewport; FALSE ignores the default. Unset fields inherit the staged value, then the built-in default. A frame with target_box = "annotated" set explicitly is invalid for the home frame; an annotated box inherited from the staged frame silently measures element boxes only for recording.
fps	Frames per second in the finished recording. Poll aims to capture at this rate; screencast capture depends on visual changes.
scale	Output size: NULL (the default) disables resizing, keeping the captured size; a number up to 4 is a scale factor; a larger number is the output width in pixels (height scales proportionally).
hold	Seconds to hold the first and last frame, c(first, last) .
keep_frames	Keep the captured PNG frames in a <name>_frames/ directory next to path . Otherwise frames are written to a temporary directory and deleted after encoding.
captions	"burn" (the default) draws captions on the recording; "vtt" writes a WebVTT sidecar without drawing captions; "both" does both. WebVTT needs an MP4 or WebM recording, not a GIF.
format	The format when path is NULL : "auto" (the default) records MP4, or GIF when knitting to a non-HTML format, where video can only be linked. With a path , the extension decides and format must be "auto" .

Value

A context in the recording chain, invisibly.

Choosing a capture method

"poll" (the default) takes screenshots at roughly the requested **fps**. Choose it when resolution matters: it captures at the page's full pixel density. **"screencast"** records when the page changes, which can capture more intermediate states of an animation. It does not guarantee a capture rate and may have lower resolution. For example, with a 640-by-480 viewport on a 2x display, poll frames are 1280-by-960 pixels while screencast frames may be 640-by-480 pixels. The **scale** argument sets output size, but enlarging a lower-resolution frame does not add detail.

Both methods record while **paparazzi** runs browser actions or **pz_wait()**. Long-running R code between browser actions, including **Sys.sleep()**, can cause intermediate changes to be missed; use **pz_wait()** when you want a visible wait in the video.

Output and framing

Stopping always captures one final frame of the page state at **pz_record_stop()**, so a change made just before the stop still appears; the last-frame hold repeats that final frame.

Use **pz_frame()** (or the staged default set with **pz_stage_frame()**) to crop the finished recording. By default the frame follows the layout at stop; **pz_frame(when = "start")** fixes it to the layout at start. If the page navigates, a frame tied to the previous page falls back to the full viewport unless it was fixed at the start. MP4 dimensions are rounded down

to multiples of 4 and WebM to even pixels; GIF keeps whole pixels. Set the viewport size before `pz_record_start()`; resizing during a framed or camera recording is not supported.

WebVTT captions (when requested) are written as a `.vtt` file next to the MP4 or WebM. Cue times include the first/last holds and explicit `pz_record_hold()` intervals but exclude pauses. Caption burn and VTT use the current caption set by `pz_annotate_caption()`.

MP4 and WebM recordings require **av**; GIF recordings require **gifski**. Simple GIFs need only gifski, but two features call in extra packages: framed GIFs require **png**, and GIFs with camera movement or burned-in captions require **av**. Packages are checked at `pz_record_start()`, or during a GIF recording at the first camera move or caption.

Recording chains

`pz_record_start()` returns a context that belongs to the new recording. Chainable functions return it (and contexts derived from it with `pz_find()`) visibly while the recording runs, and printing it stops the recording. So a chain that starts with `pz_record_start()` needs no `pz_record_stop()` when it is printed: at the end of a knitted chunk expression the recording appears in the document, and in the console it is shown in the viewer. Assign the chain or call `pz_record_stop()` to record across several statements. Chains that don't come from `pz_record_start()`, such as `page |> pz_act_click()`, keep returning invisibly and never stop a recording.

See Also

`pz_record()`, `pz_record_hold()`, `pz_frame()`, `pz_stage_frame()`

Examples

```
page <- pz_open(pz_example("tasks"), width = 800, height = 900)
path <- file.path(tempdir(), "help.mp4")

# Record the help panel opening, cropped to the page's card
page |>
  pz_record_start(path, frame = pz_frame("main")) |>
  pz_act_click("#toggle-help") |>
  pz_expect_visible(target = "#help") |>
  pz_record_stop()
file.exists(path)
pz_close(page)
```

`pz_record_stop`

Stop a recording and write the video

Description

Ends the recording started with `pz_record_start()`, encodes the captured frames, and writes the file given to `pz_record_start()`.

Usage

```
pz_record_stop(ctx)
```

Arguments

ctx A paparazzi context.

Value

ctx, invisibly, when the path was supplied outside knitting. While knitting, returns a printable image for GIF, HTML video for MP4/WebM, or a video link in non-HTML output, even when the path was supplied explicitly. Without a path in an interactive session, returns a preview that shows the recording in the viewer when printed.

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "done.gif")
page |>
  pz_record_start(path, frame = pz_frame(".task-list", pad = 8)) |>
  pz_act_click(pz_loc(".task-done", which = "first")) |>
  pz_record_stop()
file.exists(path)
pz_close(page)
```

<code>pz_screenshot</code>	<i>Take a screenshot</i>
----------------------------	--------------------------

Description

Captures a PNG of the page. With an explicit **path**, returns the context invisibly so screenshots slot into `|>` chains. Without a path, the capture is a terminal step: a figure in a knitted document or a printable preview in an interactive session.

The captured region comes from **frame**:

- `NULL` (the default): the staged framing set with `pz_stage_frame()`, or, with none staged, the current scope's box (the viewport at the root), unframed.
- A `pz_frame()` spec, or a bare locator promoted to one: a CSS selector string, a `pz_loc()` spec, or a list of either, framing the union of the matched elements' bounding boxes. A selector matching several elements is a union, not an error. Fields the spec leaves unset inherit the staged value, then the built-in default.
- `FALSE`: the scope's box or viewport, unframed.

What a frame without its own target measures resolves by precedence: the scope (scoped contexts), then the staged frame's target, then the viewport.

Screenshots are captured at the page's current device pixel ratio: the PNG's pixel dimensions are the captured CSS size multiplied by the dpr.

A screen-space caption set with `pz_annotate_caption()` is composited onto the captured PNG after framing; captioned stills require **png**.

Usage

```
pz_screenshot(ctx, path = NULL, ..., frame = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>path</code>	File path the PNG is written to; an existing file is overwritten. With <code>NULL</code> (the default) while knitting, a numbered file in the chunk's figure directory is used and included in the document; in an interactive session, a temporary PNG is shown when the result is printed. In pkgdown examples, the temporary PNG is embedded in the reference page. A path is required otherwise. In a document, end the pipe with <code>pz_screenshot()</code> to include it; give intermediate screenshots a path to keep chaining.
<code>...</code>	Checked empty; reserved for future use.
<code>frame</code>	What to capture and how to frame it: a <code>pz_frame()</code> spec or a bare locator (a CSS selector string, a <code>pz_loc()</code> spec, or a list of either) promoted to one; <code>NULL</code> (the default) uses the staged framing, if any; <code>FALSE</code> captures the scope's box or viewport unframed.

Value

With an explicit path, `ctx`, invisibly. Without a path while knitting, a knitr image; without a path in an interactive session or in pkgdown examples, an image preview. These image results are terminal, not contexts.

See Also

`pz_frame()`, `pz_stage_frame()`

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "tasks.png")

# At the root, frame = NULL captures the viewport
page |> pz_screenshot(path)

# A locator captures its box; a list captures the union of the boxes
page |> pz_screenshot(path, frame = ".task-list")
page |> pz_screenshot(path, frame = list("#new-task", ".filters"))
```

```
# Padding, aspect ratio and anchoring come from pz_frame()
page |> pz_screenshot(path, frame = pz_frame("#new-task", pad = 16))
file.exists(path)
pz_screenshot(page, frame = pz_frame("#new-task", pad = 16))
pz_close(page)
```

pz_serve_quarto *Serve a document or project with Quarto*

Description

Starts a local **quarto preview** process for a **.qmd** or **.Rmd** file, or a directory containing **_quarto.yml**. Quarto renders **.Rmd** files with **knitr**. The Quarto CLI must be on **PATH**, or its executable path must be set in **QUARTO_PATH**; the quarto R package is not required.

Usage

```
pz_serve_quarto(path, ..., render = FALSE)
```

Arguments

path	A path to a .qmd or .Rmd file, or a Quarto project directory.
...	Reserved; must be empty.
render	Whether to request a full render before previewing. FALSE passes --no-render ; projects use Quarto's preview preparation and cached execution results. Quarto still renders standalone documents on startup. TRUE passes --render all .

Details

Input watching and automatic navigation are disabled. Quarto may still reload pages after resource changes, such as CSS edits; keep those resources unchanged during a capture. Interactive documents (**runtime: shiny** or **server: shiny**) are not supported.

Value

A **PaparazziServe** handle with **\$url**, **\$port**, **\$stop()**, **\$is_running()**, and **\$logs()**. Each handle owns its process, so multiple previews can run at once. Standard output and errors go to a temporary log file, readable during and after the preview. Pass the handle to **pz_open()** to share it across pages; closing those pages leaves it running. **\$stop()** is idempotent; use **withr::defer(server\$stop())** for cleanup. A finalizer stops the preview as a last resort.

pz_serve_shiny	<i>Run a Shiny app in a background process</i>
----------------	--

Description

Starts a Shiny app – a directory or an app file – in a background R process and returns a handle for its lifecycle. One app can back any number of pages: pass the handle to `pz_open()` for each page. Closing those pages leaves the app running.

Usage

```
pz_serve_shiny(app, ..., envvars = NULL, shiny_options = list(), timeout = 10)
```

Arguments

app	A path to a Shiny app directory or an app file (anything <code>shiny::runApp()</code> accepts as a path).
...	Reserved; must be empty.
envvars	Named character vector of environment variables set in the app process, on top of the inherited environment. <code>NULL</code> adds no environment overrides.
shiny_options	Additional options for <code>shiny::runApp()</code> , e.g. <code>list(test.mode = TRUE)</code> . <code>appDir</code> is reserved; use <code>app</code> to select the app. <code>host</code> and <code>port</code> are managed by paparazzi: <code>host</code> defaults to "127.0.0.1", and <code>port</code> (default <code>NULL</code>) picks a random free port. If startup fails because the port was taken, a new port is tried.
timeout	Seconds to wait for the app to start listening; defaults to 10.

Details

The handle has `$stop()` and `$logs()` methods and a `print()` method showing the URL, port, and status – the one place the paparazzi API uses methods rather than `pz_*`() functions. `$stop()` interrupts, waits, then kills, and is idempotent, so `withr::defer(app$stop())` is a safe cleanup. A finalizer stops the app as a last resort.

App stdout and stderr go to a temporary log file (never an undrained pipe), readable mid-run with `$logs()`.

Value

A `PaparazziServe` handle with public fields `url` and `port` and methods `stop()`, `logs()`, and `is_running()`.

Examples

```
app <- pz_serve_shiny(pz_example("tasks-app"))
app

# Pages opened on a handle share the app, here at two screen sizes
desktop <- pz_open(app, width = 1280)
phone <- pz_open(app, width = 390, mobile = TRUE)
pz_get_text(phone, target = "#summary")
pz_close(desktop)
pz_close(phone)

# Closing the pages leaves the app running until you stop it
app$is_running()
head(app$logs())
app$stop()
```

pz_serve_static	<i>Serve static files over HTTP</i>
-----------------	-------------------------------------

Description

Serves a directory or a single HTML file on a free local port. For a file, its directory is served and the handle URL points to that file. Other files in that directory are also accessible over HTTP.

Usage

```
pz_serve_static(path, ..., root = NULL)
```

Arguments

path	A path to a directory or an <code>.html</code> file.
...	Reserved; must be empty.
root	The directory to serve as the server root. Defaults to the file's directory for an <code>.html</code> file. Only supported when <code>path</code> is a file, which must live inside <code>root</code> (both are resolved with <code>normalizePath()</code> , so symlinks are resolved first, and <code>root</code> can't be the filesystem root); the handle URL points at <code>path</code> relative to <code>root</code> . Use it when the page references assets outside its own directory, such as <code>../deps/styles.css</code> .

Value

A `PaparazziServe` handle with `$url`, `$port`, `$stop()`, `$is_running()`, and `$logs()`. Static servers have no captured logs: `$logs()` returns `character()`. Pass the handle to `pz_open()` to share a server across pages. Closing those pages leaves it running. `$stop()` is idempotent; use `withr::defer(server$stop())` for cleanup. A finalizer stops the server as a last resort.

Examples

```
# pz_open() serves HTML files over HTTP on its own; pz_serve_static()
# creates a handle you can share across pages instead
server <- pz_serve_static(pz_example("tasks"))
page <- pz_open(server)
pz_get_url(page)
pz_close(page)
server$stop()
```

pz_set_files	<i>Attach files to a file input</i>
--------------	-------------------------------------

Description

Auto-waits for a match, then sets the input's files to the given local files through the browser's own file-input channel, so the input's `FileList` holds the real names, sizes, and contents, and `change` fires exactly as if the files had been picked in a dialog.

Usage

```
pz_set_files(ctx, files, ..., target = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>files</code>	A character vector of paths to existing local files.
<code>...</code>	Checked empty; reserved for future use.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs (a union matching any of them). <code>NULL</code> uses the current scope; at the root context a target is required.

Value

`ctx`, invisibly.

See Also

[pz_set_value\(\)](#)

Examples

```
notes <- file.path(tempdir(), "meeting-notes.txt")
writeLines("Agenda: plants, parcel, passport", notes)

page <- pz_open(pz_example("tasks"))
page |> pz_set_files(notes, target = "#attachment")
pz_get_text(page, target = "#attachment-name")
pz_close(page)
```

`pz_set_shiny_input` *Set a bound Shiny input*

Description

Uses the input's registered Shiny binding to update the visible control and notify the server. The `id` is the complete DOM ID, including any module prefix (for example, "`mod-text`"); it is searched only in the current scope, including the scope element itself. This does not set unbound inputs, action buttons, or file inputs.

Usage

```
pz_set_shiny_input(ctx, id, value, ..., wait = TRUE)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>id</code>	Full DOM ID of a bound Shiny input.
<code>value</code>	Value accepted by the binding's <code>setValue()</code> . Date ranges accept <code>c(start, end)</code> or <code>list(start = ..., end = ...)</code> , with ISO date strings. Date-valued sliders accept ISO date strings, converted to JavaScript timestamps. Top-level <code>NULL</code> and atomic <code>NA</code> are rejected.
<code>...</code>	Checked empty; reserved for future use.
<code>wait</code>	If <code>TRUE</code> , call <code>pz_wait_for_shiny_idle()</code> after the change. If <code>FALSE</code> , return immediately after dispatching the change.

Value

`ctx`, invisibly.

Examples

```
page <- pz_open(pz_example("tasks-app"))

# Each call waits for Shiny to go idle, so the server has seen the value
page |>
  pz_set_shiny_input("title", "Buy milk") |>
  pz_set_shiny_input("priority", "high") |>
  pz_act_click("#add") |>
  pz_expect_text("3 tasks", target = "#summary")
pz_get_attr(page, "data-priority", target = ".task")
pz_close(page)
```

pz_set_value	<i>Set the value of a form control</i>
--------------	--

Description

Auto-waits for a match, then sets the value instantly – never staged as typing, even while recording – and dispatches **input** and **change**, so the page reacts exactly as if a user had made the edit. Framework-controlled inputs (React and friends) notice the change: the value is assigned through the browser’s native value setter, not the instance-level property those frameworks intercept.

Usage

```
pz_set_value(ctx, value, ..., target = NULL)
```

Arguments

ctx	A paparazzi context.
value	A string (most controls), a number (range, number), or TRUE/FALSE (checkboxes, radios).
...	Checked empty; reserved for future use.
target	A CSS selector string, a pz_loc() spec, or a list of specs (a union matching any of them). NULL uses the current scope; at the root context a target is required.

Details

Covers text inputs and textareas (clear with `pz_set_value(ctx, "")`), native `<select>` elements (matched by option **value**), checkboxes and radios (TRUE/FALSE; a radio set to TRUE unchecks the others in its group), and range, date, and number inputs. A content-editable element (plain or framework-driven, e.g. ProseMirror) has its content replaced in one step.

Value

ctx, invisibly.

See Also

[pz_act_type\(\)](#) for visible, keystroke-by-keystroke input and [pz_set_files\(\)](#).

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_set_value("Buy milk", target = "#task-title") |>
  pz_set_value("high", target = "#task-priority") |>
  pz_set_value(TRUE, target = "#task-urgent")
pz_get_value(page, target = list("#task-title", "#task-priority"))
pz_expect_checked(page, target = "#task-urgent")

# An empty string clears a text input
page |> pz_set_value("", target = "#task-title")
pz_expect_enabled(page, target = "#add-task", not = TRUE)
pz_close(page)
```

pz_stage

Stage animation settings

Description

Sets the staging options that animate pointer actions, typing, and scrolling while the page is recording. Only supplied arguments change – an omitted argument leaves its setting alone, an explicit NULL restores the built-in default, and a value sets it (so `pz_stage(cursor = FALSE)` can be undone with `pz_stage(cursor = NULL)`). Staged settings apply to the pipeline the staged context flows through: `pz_find*()` contexts inherit them, and they persist across recordings inside the chain. A fresh pipeline from the original context starts from the defaults, so assign the result (`page <- page |> pz_stage(...)`) to set page-wide defaults. Without a recording, staging is skipped and the chain runs straight to its final state – except `cursor = TRUE`, which shows a static cursor in screenshots too.

Usage

```
pz_stage(
  ctx,
  ...,
  cursor = NULL,
  cursor_speed = NULL,
  cursor_scale = NULL,
  enter = NULL,
```

```

    typing = NULL,
    typing_speed = NULL,
    pause = NULL,
    camera_follow = NULL,
    show_keys = NULL,
    click_effect = NULL,
    click_effect_color = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>cursor</code>	Cursor visibility: <code>NULL</code> (the default) shows the cursor only while recording, <code>TRUE</code> shows it always (stills too), <code>FALSE</code> never shows it. Supply <code>NULL</code> to restore the default.
<code>cursor_speed</code>	Staging speed in pixels per second for cursor glides and recorded scrolls. The default, 500 px/s, gives viewers time to follow the movement; 400-600 px/s is a useful range for deliberate actions. Each glide uses its straight-line distance; each scroll uses the largest axis delta. Duration is <code>clamp(0.25 + distance / cursor_speed, 0.5, 2)</code> seconds: the 0.25s base makes short moves less sensitive to speed, and the 0.5-2s limits mean extreme speeds have little effect. Supply <code>NULL</code> to restore the default.
<code>cursor_scale</code>	Cursor size relative to the original 20px overlay artwork, from greater than 0 to 5. The default is 1.75 (about 35px); use 1 for the original size. Applies to recordings and stills. Supply <code>NULL</code> to restore the default.
<code>enter</code>	Where a cursor that has never been shown first appears: <code>NULL</code> (the default) fades in on the target; a side (" <code>top</code> ", " <code>bottom</code> ", " <code>left</code> ", " <code>right</code> ") or corner (" <code>top left</code> ", " <code>bottom right</code> ", ...) starts off-frame past that edge and glides in. Supply <code>NULL</code> to restore the default.
<code>typing</code>	Typing style while recording: " <code>natural</code> " (the default) inserts one character at a time with randomized delays; " <code>instant</code> " inserts the whole string at once. Supply <code>NULL</code> to restore the default.
<code>typing_speed</code>	Natural typing speed in characters per second. The default is 16. Supply <code>NULL</code> to restore the default.
<code>pause</code>	Seconds to hold after each action while recording. The default is 0. Supply <code>NULL</code> to restore the default.
<code>camera_follow</code>	Whether pointer and typing actions automatically pan a zoomed recording camera to keep their target in view. Defaults to <code>TRUE</code> ; <code>FALSE</code> disables it and <code>NULL</code> restores the default.
<code>show_keys</code>	Keystroke callouts for <code>pz_act_press()</code> : " <code>none</code> " (default), " <code>words</code> ", " <code>mac</code> ", or " <code>both</code> ". Supply <code>NULL</code> to restore the default. Callouts appear only in recordings.
<code>click_effect</code>	Click feedback shown by <code>pz_act_click()</code> while recording: " <code>press</code> " (the default) scales the cursor down while pressed, " <code>ring</code> " draws an expanding ring that fades out at the click point instead of scaling, and " <code>none</code> " shows nothing. Supply <code>NULL</code> to restore the default.

`click_effect_color`

CSS color of the "ring" click effect. The default is "#e11d48". Supply NULL to restore the default.

Value

ctx, invisibly.

See Also

[pz_stage_frame\(\)](#), [pz_stage_annotate\(\)](#), [pz_cursor_show\(\)](#), [pz_record_start\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))

# Staged settings reach the pipelines the staged context flows
# through; assignment makes them page-wide defaults
page <- page |> pz_stage(enter = "left", cursor_speed = 500, typing_speed = 20, pause = 0.3)

# Recorded clicks show a custom-colored ring
page <- page |> pz_stage(click_effect = "ring", click_effect_color = "#2563eb")

path <- file.path(tempdir(), "add-task.mp4")
page |>
  pz_record(path, {
    page |>
      pz_act_type("Buy milk", target = "#task-title") |>
      pz_act_click("#add-task")
  })

# NULL restores a setting's built-in default
page <- page |> pz_stage(pause = NULL)

# cursor = TRUE shows the cursor in screenshots too
page |>
  pz_stage(cursor = TRUE) |>
  pz_act_hover("#task-title") |>
  pz_screenshot(file.path(tempdir(), "cursor.png"), frame = "#new-task")
pz_close(page)
```

Description

Sets style defaults for new annotations in screenshots and recordings. An omitted argument leaves its setting alone, an explicit `NULL` restores the built-in default, and a value sets it. The defaults apply to the pipeline the staged context flows through: `pz_find*()` contexts inherit them, while a fresh pipeline from the original context starts from the defaults – assign the result (`page <- page |> pz_stage_annotate(...)`) to set page-wide defaults. Per-call annotation style arguments override these defaults; note the per-call names on `pz_annotate()` differ (`label_fill`/`label_text_color`) from the staged names (`fill`/`text_color`), and the staged fill and text color style callout chrome only, not mark badges.

Usage

```
pz_stage_annotate(
  ctx,
  ...,
  color = NULL,
  fill = NULL,
  text_color = NULL,
  stroke_width = NULL,
  distance = NULL,
  font_family = NULL,
  font_size = NULL,
  caption_side = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>color</code>	CSS accent color for new annotations: mark outlines, callout bubble borders, leader lines and decorations. The default is <code>"#e11d48"</code> . Supply <code>NULL</code> to restore the default.
<code>fill</code>	CSS background color for new callout bubbles and their badges. By default, the fill follows each callout's resolved <code>color</code> . Supply <code>NULL</code> to restore this default. Callout chrome only: <code>pz_annotate()</code> mark badges follow their mark's <code>color</code> unless overridden per call with <code>label_fill</code> .
<code>text_color</code>	CSS text color for new callout bubbles and their badges. By default, paparazzi chooses black or white for the higher WCAG contrast ratio against the resolved fill. Supply <code>NULL</code> to restore this default. Callout chrome only, like <code>fill</code> .
<code>stroke_width</code>	Stroke width in CSS pixels for new annotation marks and callout leader lines; decoration sizes scale with it. The default is 3. Supply <code>NULL</code> to restore the default.
<code>distance</code>	Bubble-to-target gap in CSS pixels for new callouts, with or without a leader. By default the gap is 24 with a leader and 8 without one. Supply <code>NULL</code> to restore the default.

font_family	CSS font family for new annotation badges and key callouts. The default is "sans-serif". Supply NULL to restore the default. A font object staged with <code>pz_stage_fonts()</code> is also accepted and resolves to its family with a sans-serif fallback.
font_size	Badge font size in CSS pixels. The default is 14. Supply NULL to restore the default.
caption_side	Default side for new <code>pz_annotate_caption()</code> captions: "bottom" (the default) or "top". Supply NULL to restore the default.

Value

ctx, invisibly.

See Also

`pz_stage()`, `pz_stage_frame()`, `pz_annotate()`

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "tasks.png")

# Assignment makes these styles page-wide defaults for new annotations
page <- page |> pz_stage_annotate(color = "#2563eb", font_size = 16)
page |> pz_annotate("#add-task", label = TRUE) |> pz_screenshot(path)

# A one-off pipeline can undo a default with NULL: its callout uses
# the built-in red again
page |>
  pz_stage_annotate(color = NULL) |>
  pz_annotate("#task-title", label = TRUE) |>
  pz_screenshot(path, frame = pz_frame("#task-title", pad = 16, target_box = "annotated"))
pz_close(page)
```

`pz_stage_fonts`

Stage fonts for annotations and captions

Description

Loads one or more fonts into the page so that everything Chrome draws for it — annotation badges, callout bubbles, captions, and key callouts — can use them. Declare fonts with `pz_font_google()`, `pz_font_bunny()`, or `pz_font_file()` and pass them in ...

Usage

```

pz_stage_fonts(
  ctx,
  ...,
  on_error = getOption("paparazzi.stage_fonts.on_error", "stop")
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Font objects from <code>pz_font_google()</code> , <code>pz_font_bunny()</code> , or <code>pz_font_file()</code> .
<code>on_error</code>	What to do when a font fails to load: "stop" (the default) aborts without staging any of the call's fonts, "warn" warns and stages the fonts that loaded, and "ignore" stages the fonts that loaded, silently. Set with <code>options(paparazzi.stage_fonts.on_error =)</code> to change the default.

Details

Staging is an ahead-of-time step: `pz_stage_fonts()` blocks until every face has loaded (or fails per `on_error`), so annotating is never slowed by a font load and never flashes a fallback font. Staged fonts persist across recordings; after navigation they are re-added to the new document lazily, the first time something is annotated there.

Repeated calls add fonts. A face whose family, weight, and style are already staged is replaced by the new one. There is no clear or reset.

Remote stylesheets and font files are fetched by Chrome, not R, and stay in Chrome's HTTP cache. On pages whose Content Security Policy blocks the fetch (`connect-src` for the stylesheet, `font-src` for the font files) the load fails and is routed through `on_error`. `pz_font_file()` sidesteps CSP entirely because its bytes are handed to the page directly.

Value

`ctx`, invisibly.

See Also

`pz_stage_annotate()`, whose `font_family` sets the default family for annotations and key callouts

Examples

```

page <- pz_open(pz_example("tasks"))
page |>
  # on_error = "warn" keeps unattended renders (CI, pkgdown) going when
  # the font CDN is unreachable; drop it to make a missing font fatal.
  pz_stage_fonts(pz_font_google("Silkscreen"), on_error = "warn") |>
  # font_family takes a CSS family string; use the staged family with
  # a generic fallback
  pz_stage_annotate(font_family = '"Silkscreen", sans-serif') |>

```

```

pz_annotate("#add-task", label = "New") |>
  pz_screenshot(file.path(tempdir(), "tasks.png"))
pz_close(page)

```

`pz_stage_frame` *Stage the default framing*

Description

`pz_stage_frame()` stores a default `pz_frame()` on the context: every screenshot and recording started from the staged context resolves its own `frame =` field by field, and an unset field takes the staged value if one is set, else the built-in default. `frame = FALSE` disables framing for a single call. The staged frame applies to the pipeline the staged context flows through – `pz_find*()` contexts inherit it – while a fresh pipeline from the original context starts from the defaults; assign the result (`page <- page |> pz_stage_frame(...)`) to set page-wide defaults. `pz_stage_frame(NULL)` clears an inherited staged frame for that pipeline only, and a new `pz_stage_frame()` call replaces the staged frame wholesale (unset fields fall back to built-in defaults, they do not merge with a previously staged frame). Camera shots ignore the staged frame; see `pz_camera()`.

Unlike `pz_stage()`'s animation settings, framing (like `pz_stage_annotate()`'s styles) applies to screenshots as well as recordings. An annotated staged frame measures attached annotations in stills; recordings use only its element boxes for the home frame.

Usage

```

pz_stage_frame(
  ctx,
  ...,
  ratio = NULL,
  pad = NULL,
  offset = NULL,
  anchor = NULL,
  bounds = NULL,
  target_box = NULL,
  zoom = NULL
)

```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	The default frame's target: a CSS selector string, a <code>pz_loc()</code> spec, or a list of either. Several unnamed arguments are unioned. With no target, framed captures use the current scope's box (the viewport at the root). A single <code>NULL</code> clears the staged frame for this pipeline.
<code>ratio, pad, offset, anchor, bounds, target_box, zoom</code>	Framing settings, as in <code>pz_frame()</code> ; <code>NULL</code> leaves the field unset, so captures fall back to the built-in default for it.

Value

ctx, invisibly.

See Also

[pz_stage\(\)](#), [pz_stage_annotate\(\)](#), [pz_frame\(\)](#), [pz_screenshot\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
path <- file.path(tempdir(), "tasks.png")

page <- page |> pz_stage_frame(pad = 24)
# Framed by the default: the form plus 24px of padding
page |> pz_screenshot(path, frame = "#new-task")
# Fields not set here still come from the staged default
page |> pz_screenshot(path, frame = pz_frame("#new-task", ratio = 16/9))
# FALSE turns framing off for one call
page |> pz_screenshot(path, frame = FALSE)
# NULL clears the staged default for this pipeline only
page |> pz_stage_frame(NULL) |> pz_screenshot(path, frame = "#new-task")
pz_close(page)
```

pz_wait

Wait while pumping the page's event loop

Description

Pauses for **seconds**, driving chromote's child event loop so timers scheduled on it (e.g. recording capture) keep firing during the wait. Never sleeps without pumping the loop.

Usage

```
pz_wait(ctx, seconds)
```

Arguments

ctx A paparazzi context.

seconds Number of seconds to wait.

Value

ctx, invisibly.

Examples

```
page <- pz_open(pz_example("tasks"))

# A fixed pause; prefer an expectation or wait_for function when you
# know what you're waiting for
page |>
  pz_act_click("#toggle-help") |>
  pz_wait(0.5)
pz_close(page)
```

pz_wait_for_js	<i>Wait until a JavaScript condition holds</i>
----------------	--

Description

Polls `expr` until it evaluates truthy, then returns `ctx` invisibly. Each poll awaits a promise `expr` returns, so async conditions work; JavaScript truthiness applies (a non-empty string or a non-zero number counts). An `expr` that throws is a JavaScript error, not a failed poll.

Usage

```
pz_wait_for_js(ctx, expr, ..., timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>expr</code>	A string of JavaScript that evaluates truthy when the condition holds. A returned promise is awaited first.
<code>...</code>	Checked empty; reserved for future use.
<code>timeout</code>	Seconds before giving up; NULL uses the session default.

Details

There is deliberately no element-state wait (`pz_wait_for(target, state =)`): `pz_expect_visible()`, `pz_expect_hidden()`, and `pz_expect_exists()` with `not = TRUE` already retry, so they wait.

Value

`ctx`, invisibly.

See Also

[pz_wait_for_stable\(\)](#), [pz_wait_for_navigation\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks"))
page |>
  pz_act_type("Buy milk", target = "#task-title") |>
  pz_act_press("Enter") |>
  pz_wait_for_js("document.querySelectorAll('.task').length === 8")
pz_get_text(page, target = pz_loc(".task-title", which = "first"))
pz_close(page)
```

```
pz_wait_for_navigation
```

Wait for a navigation to finish

Description

The explicit wait after an action that navigates – a clicked link, a submitted form, a JS redirect. Paparazzi never detects navigations on its own, so a wait marks exactly where one is expected. Call it immediately after the action. It waits for the new document to finish loading and then hold still for a moment. Three cases pass:

Usage

```
pz_wait_for_navigation(
  ctx,
  ...,
  wait = c("auto", "load", "shiny", "none"),
  timeout = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>wait</code>	What to wait for: "load" settles the navigation; "shiny" also waits for Shiny idle after load. "auto" uses "shiny" only when the page was opened on an app handle or app path and lands on that app's origin; otherwise it uses "load". "none" skips settling and only resets the scope.
<code>timeout</code>	Seconds before giving up; NULL uses the session default.

Details

- the navigation finished before the wait started: the current document differs from the document where the preceding action began (including a page restored from the back/forward cache);
- the navigation is in flight when the wait starts, and is waited out;

- the navigation begins while the wait is running. One scheduled beyond the timeout can't be caught – block on its trigger with `pz_wait_for_js()` first.

A page where nothing navigates times out with a classed error rather than passing, and so does a second wait after the same action: a successful wait uses up the action's navigation.

On success the wait resets the scope to the root and releases every pinned scope object: contexts scoped before the navigation error on their next use instead of acting on a stale set.

Value

ctx, invisibly, with the scope reset to the root.

See Also

`pz_find_reset()`, `pz_wait_for_js()`, `pz_wait_for_stable()`

Examples

```
page <- pz_open(pz_example("tasks"))
page |> pz_act_click(pz_loc(".task-done", within = pz_loc(".task", has_text = "bank")))
pz_get_count(page, target = ".task.done")

# "Start over" is a link to a new copy of the page
page |>
  pz_act_click("#start-over") |>
  pz_wait_for_navigation()
pz_get_count(page, target = ".task.done")
pz_close(page)
```

`pz_wait_for_shiny_idle`

Wait until a Shiny page is idle

Description

Waits for the Shiny connection, for `<html>` to lose `shiny-busy`, and for every `.recalculating` output to finish. All three conditions must hold continuously for at least 200ms, including brief busy/recalculating transitions. A page without the Shiny global errors instead of waiting; a page whose Shiny app never connects times out.

Usage

```
pz_wait_for_shiny_idle(ctx, ..., timeout = NULL)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>timeout</code>	Seconds before giving up; NULL uses the session default.

Value

`ctx`, invisibly.

See Also

[pz_open\(\)](#)

Examples

```
page <- pz_open(pz_example("tasks-app"))

# Clicking Add makes the server re-render the task list, which takes a moment
page |>
  pz_set_value("Buy milk", target = "#title") |>
  pz_act_click("#add") |>
  pz_wait_for_shiny_idle()
pz_get_text(page, target = "#summary")
pz_close(page)
```

<code>pz_wait_for_stable</code>	<i>Wait until an element stops changing</i>
---------------------------------	---

Description

Samples the `prop` property of every element matching `target` (or the current context) until the sampled value has held still for `for_ms` milliseconds: text has stopped arriving, or an animation has come to rest. `prop = "rect"` samples the bounding box instead of a property, rounded to whole pixels.

Usage

```
pz_wait_for_stable(
  ctx,
  ...,
  target = NULL,
  prop = "textContent",
  for_ms = 500,
  timeout = NULL
)
```

Arguments

<code>ctx</code>	A paparazzi context.
<code>...</code>	Checked empty; reserved for future use.
<code>target</code>	A CSS selector string, a <code>pz_loc()</code> spec, or a list of specs and strings (a union matching any of them). <code>NULL</code> means the current context: the pinned set at a scoped context, the page body at the root.
<code>prop</code>	The element property to sample: any single JavaScript property name (<code>"textContent"</code> , <code>"value"</code> , <code>"scrollTop"</code> , ...), or <code>"rect"</code> for the bounding box.
<code>for_ms</code>	Milliseconds the sampled value must hold still before the wait passes.
<code>timeout</code>	Seconds before giving up; <code>NULL</code> uses the session default.

Details

The wait first waits (up to `timeout`) for `target` to match, then samples within a second, separate `timeout` budget of its own: a target that appears near the locator deadline still gets its full `for_ms` window.

There is deliberately no element-state wait: `pz_expect_visible()`, `pz_expect_hidden()`, and `pz_expect_exists()` with `not = TRUE` already retry, so they wait.

Value

`ctx`, invisibly.

See Also

`pz_wait_for_js()`, `pz_wait_for_navigation()`

Examples

```
page <- pz_open(pz_example("tasks"))

# The status line reads "Saving..." and then "Saved"; wait for it to settle
page |>
  pz_act_type("Buy milk", target = "#task-title") |>
  pz_act_click("#add-task") |>
  pz_wait_for_stable(target = "#status", for_ms = 500)
pz_get_text(page, target = "#status")
pz_close(page)
```

`pz_with_page`

Open a page that closes when a block or calling frame exits

Description

`pz_with_page()` evaluates `code` with the page open and closes it on exit, including on error. `pz_local_page()` opens a page and closes it when the calling frame (e.g. a test) exits, via `withr::defer()`. Both accept an already-open page or context (e.g. the staged result of `pz_stage()`) or anything `pz_open()` accepts, and always close the underlying page on exit: the block owns the resource. A context is passed through as itself, so its staged settings reach the code it flows into.

Usage

```
pz_with_page(x, code, ...)
```

```
pz_local_page(x, ..., .env = caller_env())
```

Arguments

<code>x</code>	An open page or context, or anything <code>pz_open()</code> accepts.
<code>code</code>	Code to run while the page is open. An expression (evaluated as-is; useful when <code>x</code> is an already-open page the code can reference) or a function, called with the page (or context) as its only argument. A braced <code>{ }</code> block is always treated as an expression, even if it returns a function.
<code>...</code>	Passed to <code>pz_open()</code> when <code>x</code> is not already a page or context.
<code>.env</code>	The frame whose exit closes the page.

Value

`pz_with_page()` returns the page (or context) invisibly; `pz_local_page()` returns it visibly.

Examples

```
path <- file.path(tempdir(), "task-list.png")

# The page closes when the function returns, even if it errors
pz_with_page(pz_example("tasks"), function(page) {
  pz_screenshot(page, path, frame = ".task-list")
})
file.exists(path)

# pz_local_page() ties the page to the calling function, e.g. a test
count_tasks <- function() {
  page <- pz_local_page(pz_example("tasks"))
  pz_get_count(page, target = ".task")
}
```

pz_with_page

107

```
}  
count_tasks()
```

Index

chromote::ChromoteSession, 77
chromote::default_chromote_object(), 78
chromote::set_chrome_args(), 78
chromote::set_default_chromote_object(), 78

grepl(), 48, 50–52

normalizePath(), 89

paparazzi-actions, 4
pz_act_blur, 5
pz_act_blur(), 11, 43
pz_act_click, 6
pz_act_click(), 9, 12, 15, 18, 38, 43, 94
pz_act_drag, 7
pz_act_focus, 10
pz_act_focus(), 5, 12, 43
pz_act_hover, 11
pz_act_hover(), 7, 9
pz_act_press, 12
pz_act_press(), 4, 5, 7, 9, 10, 12, 13, 15, 16, 18, 94
pz_act_scroll, 14
pz_act_select_text, 16
pz_act_type, 17
pz_act_type(), 7, 11, 13, 16, 17, 93
pz_annotate, 19
pz_annotate(), 22, 25, 26, 96, 97
pz_annotate_callout, 20
pz_annotate_caption, 23
pz_annotate_caption(), 84, 86, 97
pz_annotate_clear, 24
pz_annotate_clear(), 19–23, 25, 26
pz_annotate_clear(id = caption), 23
pz_annotate_redact, 24
pz_annotate_spotlight, 25
pz_camera, 26
pz_camera(), 80, 99

pz_camera_reset, 27
pz_camera_reset(), 26
pz_chromote, 28
pz_close, 29
pz_cursor_hide, 29
pz_cursor_hide(), 34
pz_cursor_leave, 30
pz_cursor_leave(), 29, 30, 32–34
pz_cursor_move, 31
pz_cursor_move(), 29, 34
pz_cursor_show, 33
pz_cursor_show(), 29–32, 95
pz_device, 34
pz_device(), 78
pz_example, 36
pz_expect_attr, 37
pz_expect_attr(), 37
pz_expect_checked, 38
pz_expect_checked(), 38
pz_expect_class, 39
pz_expect_class(), 39
pz_expect_count, 40
pz_expect_count(), 40
pz_expect_enabled, 41
pz_expect_enabled(), 41
pz_expect_exists, 42
pz_expect_exists(), 37–46, 48–51, 53, 101, 105
pz_expect_focused, 43
pz_expect_focused(), 43
pz_expect_hidden (*pz_expect_visible*), 53
pz_expect_hidden(), 53, 101, 105
pz_expect_in_viewport, 44
pz_expect_in_viewport(), 44, 53
pz_expect_js, 45
pz_expect_js(), 45
pz_expect_style, 46
pz_expect_style(), 46, 69

pz_expect_text, 48
 pz_expect_text(), 48, 69
 pz_expect_title, 49
 pz_expect_title(), 49
 pz_expect_url, 50
 pz_expect_url(), 50
 pz_expect_value, 51
 pz_expect_value(), 51
 pz_expect_visible, 53
 pz_expect_visible(), 6, 44, 53, 101, 105
 pz_find, 54
 pz_find(), 15, 54–59, 72, 75, 84
 pz_find_first, 55
 pz_find_first(), 54–57
 pz_find_last, 56
 pz_find_last(), 54, 56, 57
 pz_find_nth, 57
 pz_find_nth(), 54, 56, 57
 pz_find_pop, 58
 pz_find_pop(), 54, 59
 pz_find_reset, 59
 pz_find_reset(), 54, 58, 76, 103
 pz_font_bunny, 60
 pz_font_bunny(), 61, 97, 98
 pz_font_file, 60
 pz_font_file(), 60, 61, 97, 98
 pz_font_google, 61
 pz_font_google(), 60, 61, 97, 98
 pz_frame, 62
 pz_frame(), 26, 27, 62, 82–86, 99, 100
 pz_get_attr, 64
 pz_get_attr(), 64
 pz_get_count, 65
 pz_get_count(), 65
 pz_get_elements, 66
 pz_get_elements(), 66
 pz_get_html, 66
 pz_get_html(), 66
 pz_get_rect, 67
 pz_get_rect(), 66, 67, 69
 pz_get_style, 68
 pz_get_style(), 68
 pz_get_text, 69
 pz_get_text(), 69
 pz_get_title, 70
 pz_get_title(), 70
 pz_get_url, 71
 pz_get_url(), 71
 pz_get_value, 71
 pz_get_value(), 71
 pz_inspect, 72
 pz_js, 73
 pz_loc, 74
 pz_loc(), 6, 8, 10, 11, 14, 16, 18, 20, 21, 25, 26, 31, 33, 37–40, 42–48, 52–57, 62–69, 71, 72, 74, 75, 82, 85, 86, 90, 92, 99, 105
 pz_local_page (*pz_with_page*), 106
 pz_local_page(), 106
 pz_nav_back (*pz_nav_goto*), 75
 pz_nav_back(), 76
 pz_nav_forward (*pz_nav_goto*), 75
 pz_nav_forward(), 76
 pz_nav_goto, 75
 pz_nav_reload (*pz_nav_goto*), 75
 pz_open, 77
 pz_open(), 9, 29, 35, 36, 87–89, 104, 106
 pz_record, 79
 pz_record(), 82, 84
 pz_record_hold, 80
 pz_record_hold(), 27, 28, 82, 84
 pz_record_pause, 81
 pz_record_pause(), 27, 28
 pz_record_resume (*pz_record_pause*), 81
 pz_record_start, 82
 pz_record_start(), 23, 26, 29, 79, 84, 95
 pz_record_stop, 84
 pz_record_stop(), 23, 27, 28, 79, 84
 pz_screenshot, 85
 pz_screenshot(), 23, 72, 100
 pz_serve_quarto, 87
 pz_serve_quarto(), 77
 pz_serve_shiny, 88
 pz_serve_shiny(), 36, 77, 78
 pz_serve_static, 89
 pz_serve_static(), 9, 77
 pz_set_files, 90
 pz_set_files(), 4, 5, 7, 9, 11–13, 15, 17, 18, 93
 pz_set_shiny_input, 91
 pz_set_shiny_input(), 4, 5, 7, 9, 11–13, 15, 17, 18
 pz_set_value, 92
 pz_set_value(), 4, 5, 7, 9, 11–13, 15, 17, 18, 90

`pz_stage`, 93
`pz_stage()`, 4–10, 12–15, 17, 18, 31, 33,
34, 97, 99, 100, 106
`pz_stage_annotate`, 95
`pz_stage_annotate()`, 20, 22, 23, 95,
98–100
`pz_stage_fonts`, 97
`pz_stage_fonts()`, 20, 22, 23, 60, 61, 97
`pz_stage_frame`, 99
`pz_stage_frame()`, 62, 82–86, 95, 97, 99
`pz_wait`, 100
`pz_wait()`, 80, 83
`pz_wait_for_js`, 101
`pz_wait_for_js()`, 103, 105
`pz_wait_for_navigation`, 102
`pz_wait_for_navigation()`, 101, 105
`pz_wait_for_shiny_idle`, 103
`pz_wait_for_shiny_idle()`, 76, 91
`pz_wait_for_stable`, 104
`pz_wait_for_stable()`, 101, 103
`pz_with_page`, 106
`pz_with_page()`, 106

`shiny::runApp()`, 88

`tempdir()`, 78

`withr::defer()`, 106