ActionExecutorService

September 4, 2026

ActionExecutorService

Kind: Service

Source: `atloria-monorepo/apps/screenshot-worker/src/app/screenshot/services/action-executor.service.ts`

Service for executing pre-capture actions on Playwright pages

Supports 13 action types:

- click, dblclick: Click interactions
- type, fill: Text input
- press: Keyboard keys
- wait: Duration or selector waits
- scroll: Scroll to element or page bottom
- hover, focus, blur: Element state changes
- select: Dropdown selection
- navigate: Page navigation
- frame: Iframe context switching

And 5 wait condition types:

- selector, timeout, networkIdle, loadState, function

`ActionExecutorService` executes a configured list of **pre-capture actions** against a Playwright `Page` (or an active `Frame`) before a screenshot is taken. It provides a single, validated execution pipeline that maps high-level action definitions (click, type, wait, navigate, frame, etc.) into concrete Playwright calls. This service typically sits inside the screenshot worker flow, ensuring pages are in the desired state (navigation complete, elements ready, UI interacted with) prior to capture.

Methods

Method	Signature	Returns	Description
`executeActions`	`executeActions(page: Page, actions: ScreenshotActionDto[]	undefined)`	`Promise<void>`
`executeActionsWithHook`	`executeActionsWithHook(page: Page, actions: ScreenshotActionDto[]	undefined, onAfterEach: (action: ScreenshotActionDto, index: number) => Promise)`	`Promise<void>`
`executeAction`	`executeAction(page: Page, action: ScreenshotActionDto, index: number, context: ActionContext)`	`Promise<ActionContext	void>`
`executeWaitFor`	`executeWaitFor(page: Page, waitFor: WaitForDto	undefined)`	`Promise<void>`
`validateAction`	`validateAction(action: ScreenshotActionDto)`	`void`	Validates an action has all required fields
`captureDOMState`	`captureDOMState(page: Page)`	`Promise<CapturedDOMState>`	Capture the current DOM state from a page Used for AI verification and recovery

Where it refuses work

- `ActionExecutorService` stops the work with `Error` when `!ACTION_TYPES.includes(action.type as ActionType)`.
- `ActionExecutorService` stops the work with `Error` when `selectorRequiredActions.includes(action.type) && !action.selector`.
- `ActionExecutorService` stops the work with `Error` when `valueRequiredActions.includes(action.type) && !action.value`.
- `ActionExecutorService` stops the work with `Error` when `action.type === 'press' && !action.key` — “Key is required for press action”.
- `ActionExecutorService` stops the work with `Error` when `action.type === 'frame' && !action.frameAction` — “frameAction is required for frame action”.
- `ActionExecutorService` stops the work with `Error` when `action.type === 'wait' && action.duration === undefined && !action.selector` — “Either duration or selector is required for wait action”.

When something fails

- `ActionExecutorService` handles failure in 9 places: it discards it silently in 5, lets it reach the caller in 3, and turns it into a return value in 1.

Diagram

```
sequenceDiagram
  participant Orchestrator as Screenshot Orchestrator
  participant Executor as ActionExecutorService
  participant Page as Playwright Page
  participant Frame as Playwright Frame

  Orchestrator->>Executor: execute(page, actions[])
  loop for each action
    alt action.type == "frame"
      Executor->>Page: frameLocator()/frame()
      Page-->>Executor: Frame context
      Executor->>Frame: set active context
    else action targets frame context
      Executor->>Frame: perform action (click/type/wait/etc.)
    else action targets page context
      Executor->>Page: perform action (navigate/wait/scroll/etc.)
    end
  end
  Executor-->>Orchestrator: done (page ready for capture)
```

Usage

```
import { Injectable } from '@nestjs/common';
import { chromium, type Page } from 'playwright';
import { ActionExecutorService } from './services/action-executor.service';

@Injectable()
export class ScreenshotJobRunner {
  constructor(private readonly actionExecutor: ActionExecutorService) {}

  async run() {
    const browser = await chromium.launch();
    const page: Page = await (await browser.newContext()).newPage();

    // Example action list (shape may vary slightly based on your DTOs)
    const actions = [
      { type: 'navigate', url: 'https://example.com' },
      { type: 'wait', condition: { type: 'loadState', state: 'networkidle' } },
      { type: 'click', selector: 'text=Sign in' },
      { type: 'fill', selector: 'input[name="email"]', value: 'user@example.com' },
      { type: 'press', selector: 'input[name="password"]', key: 'Tab' },
      { type: 'type', selector: 'input[name="password"]', value: 'correct horse battery staple' },
      { type: 'click', selector: 'button[type="submit"]' },
      { type: 'wait', condition: { type: 'selector', selector: '[data-testid="dashboard"]' } },
      { type: 'scroll', to: 'bottom' },
      // Switch into an iframe, then interact within it
```

```
      { type: 'frame', selector: 'iframe#billing' },
      { type: 'select', selector: 'select#plan', value: 'pro' },
      { type: 'hover', selector: '[data-testid="plan-details"]' },
    ];

    await this.actionExecutor.execute(page, actions);

    // Now page is prepared for capture
    const screenshot = await page.screenshot({ fullPage: true });

    await browser.close();
    return screenshot;
  }
}
```

AI Coding Instructions

- Keep the action execution path **strictly deterministic**: validate action payloads up-front and route by `action.type` with a single dispatcher to avoid divergent behavior.
- When implementing waits, prefer explicit `wait` actions with supported condition types (`selector`, `timeout`, `networkIdle`, `loadState`, `function`) and avoid hidden implicit sleeps inside other actions.
- Be careful with **frame context**: `frame` actions should update the active execution target (Page vs Frame). Ensure subsequent selector-based actions operate in the intended context.
- Integrate new action types by mapping directly to Playwright primitives and preserving consistent error messages (include `action.type` and selector/url where applicable) to aid job debugging.

Referenced By

- `ScreenshotModule` (MODULE_PROVIDES)
- `BatchScreenshotService` (DEPENDS_ON)
- `FlowReplayService` (DEPENDS_ON)