AI Browser Extension

Paul Hanchett

Introduction

The AI Browser Extension (AIBE) is an extension for Chrome base browsers. It aims to provide two sets of functionality:

1. To enable an AI to act as an observer, watching human interactions with web pages, potentially recording the interactions to be used for training,
2. And, to enable an AI to act on the web page, via the browser, just as a human would.

Key to both these activities is that the observer and actor are both limited to what a human viewing the browser page could see, and to perform only the actions a human user could do. By limiting actions and viewability in this way, the browser becomes much simpler to control (keystrokes and mouse actions only), and the screen is simpler to view and interpret (only text and controls that are visible on the screen are reported, in the order they appear on the screen, with labels identifying targets for clicks and keystrokes).

Currently the control endpoint for the browser is a python based web server, with inbound (observer) events from the browser viewable. The Actor control channel back to the browser has also been implemented.

Installation

The AI Browser Extension consists of two components that work together:

1. **AIBE Server** - Python server (installed via PyPI)
2. **Chrome Extension** - Browser extension (downloaded from server)

PyPI Installation (Recommended)

System Requirements:

- Python 3.8 or higher
- Google Chrome browser
- Internet connection for PyPI download

Step 1: Install AIBE Server

Create a virtual environment (recommended to avoid conflicts):

```
# Create virtual environment
python -m venv .venv

# Activate virtual environment
# Windows:
.venv\Scripts\activate
# Linux/macOS/WSL:
source .venv/bin/activate

# Install from PyPI
pip install aibe-server
```

Step 2: Install Chrome Extension

Start the server and find the extension installation location:

```
# Start the server
aibe-server start

# Check where extension files are located
aibe-server status
# Note the installation path shown in the status output

# Visit the installation page
# Open Chrome and go to: http://localhost:3001/extension/install
```

Follow the on-screen instructions to download and install the Chrome extension in Developer Mode. The status command shows you exactly where the server is installed, which helps locate the extension files.

Platform-Specific Notes

Windows:

- Use Command Prompt or PowerShell
- Virtual environment activation: `.venv\Scripts\activate`
- Server runs as background process when started

Linux/macOS:

- Use Terminal
- Virtual environment activation: `source .venv/bin/activate`
- Server runs as detached background process

WSL (Windows Subsystem for Linux):

- Install in WSL environment using Linux commands
- Server localhost:3001 accessible from both WSL and Windows due to network mirroring
- Can manage server from either Windows or WSL terminals
- Chrome extension installation works from Windows Chrome browser

Development Installation

For developers wanting to modify the source code:

> **Note:** Source repository access requires permission. Contact paul@paulhanchett.com for contributor access.

```
# Clone repository (requires access)
git clone <repository-url>
cd ai-browser-extension

# Install server in development mode
cd server
python -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install -e .

# Load Chrome extension from local files
# Load unpacked extension from: server/extension/
```

Server Management

The AIBE server provides a complete command-line interface for managing the background server process.

Basic Commands

```
aibe-server start     # Start server in background
aibe-server status    # Check if server is running
aibe-server stop      # Stop the server
aibe-server restart   # Stop then start
aibe-server help      # Show command help
```

Command Details

Start Server:

```
aibe-server start
```

- Starts server on `http://localhost:3001`
- Runs in background (detached process)
- Shows server PID and installation path
- Detects if server already running

Check Status:

```
aibe-server status
```

- Shows server running status
- Displays installation path, PID, uptime

- Shows active sessions and event counts
- Indicates platform (Windows/Linux/WSL)

Stop Server:

```
aibe-server stop
```

- Gracefully shuts down server
- Works across platforms (Windows/Linux/WSL)
- Handles cross-platform process management

Restart Server:

```
aibe-server restart
```

- Stops existing server, then starts new instance
- Useful during development or configuration changes

Help:

```
aibe-server --help
# or
aibe-server help
```

- Shows complete command reference
- Includes usage examples

Cross-Platform Behavior

Same Machine, Different Environments:

- Server can run in Windows, WSL, or Linux
- Commands detect which environment server is running in
- Provides appropriate management instructions
- WSL network mirroring makes localhost:3001 accessible from both Windows and WSL

Process Isolation:

- Each platform maintains separate server processes
- Windows and WSL can each run their own server instance
- Status command indicates which platform server is running on
- Stop/restart commands only affect servers in the same environment

Configuration

Default Settings:

- Server Port: 3001
- Server Host: localhost (127.0.0.1)
- Log Level: INFO
- Background Mode: Enabled by default

Environment Variables:

```
AIBE_PORT=3001          # Change server port
AIBE_DEBUG=true         # Enable debug logging
AIBE_LOG_LEVEL=DEBUG    # Set logging verbosity
```

Verification

After installation, verify everything works:

1. **Server Status:**

   ```
   aibe-server status
   ```

   Should show "Server running" with PID and uptime.

2. **Web Interface:**
   Visit `http://localhost:3001/status` in browser - should display server information.

3. **Extension Status:**
   Click Chrome extension icon - should show "Connected" status.

4. **Event Capture:**
   Browse any website with extension active - events should be captured automatically.

Sessions

A user may have more than one tab open in their browser, and it would be confusing to mix them all together. So the server backend assigns a sessionID to each tab the user opens, and uses that to identify the source tab of each event.

Session Management Endpoints

The endpoints below are used by the extension injected script, content.js, to manage sessions. Use of these endpoints by the extension is automatic:

- `PUT /sessions/init` - Initialize new session with TabIdentity data
- `GET /sessions` - List all active sessions (returns array of SessionInfo objects)
- `POST /sessions/close` - Close session when tab is closed
- `POST /sessions/{session_id}/heartbeat` - Update session activity timestamp

Observer Channel

Load

Load is a navigation event, reporting that the browser is navigating to a new URL.


```
{
    "type": "event",
    "event": "load",
    "url": "http://localhost:3001/test-dropdown",
    "timestamp": "2025-07-03T18:23:25.395Z"
},
```


Screen Status

Every event (mouse clicks and key presses) *may* cause the screen to change. Screens are not reported until they change. Once they start to change, they won't be reported until they stabilize. These changes in the screen cause it to be reported as a **screen_status** event.

The entire screen is scanned for visible textual content and for visual elements (like input or anchor elements), and then sorted by position top to bottom, left to right, putting all elements in their X, Y screen relationship to one another. If an control has a label assigned, the label is added to the control. If not, an algorithm attempts to locate the visual element a human associate with the control on the screen.

Screen status reports the current URL of the page, which control has focus, the list of visible elements including text and controls, and the time that the screen stabilized as JSON:


```
{
    "type": "screen_status",
    "url": "http://localhost:3001/test-dropdown",
    "focus_label": null,
    "focus_id": null,
    "visible_elements": [
      {
        "label": "Country",
        "control_type": "INPUT_DROPDOWN",
        "tagName": "select",
        "type": "select-one",
        "current_selection": "Select a country",
        "current_value": "",
        "clickable_options": [
          "\"United States\" → usa",
          "\"Canada\" → canada",
          "\"United Kingdom\" → uk",
          "\"France\" → france"
        ],
        "is_open": false,
        "is_disabled": false,
        "href": null,
```

```
        "top": 441.146697998047,
        "left": 278.016510009766,
        "bottom": 474.989673614502,
        "right": 478.016510009766,
        }
      },
      {
        "label": "Skills (Hold Ctrl to select multiple)",
        "control_type": "INPUT_DROPDOWN",
        "tagName": "select",
        "type": "select-multiple",
        "current_selection": "(none selected)",
        "current_value": null,
        "clickable_options": [
          "\"JavaScript\" → js",
          "\"Python\" → py",
          "\"Java\" → java",
          "\"CSS\" → css",
          "\"HTML\" → html",
          "\"React\" → react"
        ],
        "is_open": false,
        "is_disabled": false,
        "href": null,
        "top": 618.347106933594,
        "left": 278.016510009766,
        "bottom": 738.347106933594,
        "right": 478.016510009766,
        }
      }
    ],
    "timestamp": "2025-07-03T18:23:25.698Z"
  },
```


Notice also that the controls reported are *not* HTML control names! To simplify the many variations on controls like **INPUT**, controls are named according to their semantic type.

Mouse and Keyboard events

Mouse clicks and keyboard presses are reported as events, and include the targets of those events. An event target will always be a visible (e.g., clickable) element on the screen.


```
{
    "type": "mouse",
    "event": "click",
    "target": {
      "control": "Country",
      "control_id": "country",
      "selected_text": "Select a country",
      "selected_value": "",
      "clicked_option": null,
```

```
      "modifiers": {
        "ctrl": false,
        "shift": false,
        "alt": false,
        "meta": false
      }
    },
    "target_id": "country",
    "y": 458,
    "x": 466,
    "button": 0,
    "buttons": 0,
    "timestamp": "2025-07-03T18:23:30.037Z"
  },
```

Information in the **target:** entry can be used to locate the element this click (or keypress) occurred on, as well as the final values set on the element as a result.

Note: It's not possible to completely identify editing actions on a control, so we've chosen to report only the control's final state.

Keyboard events are consolidated to reduce noise. Text input keystrokes are accumulated into single events containing arrays of keys, key codes, timestamps, and the final text value. Control keys (Enter, Space, Arrow keys, Tab) are sent immediately as navigation events. Consolidation automatically flushes when the user switches to a different input field or performs other actions.

```
{
  "type": "keyboard",
  "event": "text_input",
  "target": {
    "control": "Email Address",
    "control_id": "email",
    "selected_text": "user@example.com",
    "selected_value": "user@example.com"
  },
  "keystrokes": [
    {
      "key": "u",
      "key_code": "KeyU",
      "timestamp": "2025-01-08T10:15:30.100Z",
      "modifiers": { "ctrl": false, "shift": false, "alt": false, "meta": false }
    },
    {
      "key": "s",
      "key_code": "KeyS",
      "timestamp": "2025-01-08T10:15:30.180Z",
      "modifiers": { "ctrl": false, "shift": false, "alt": false, "meta": false }
    },
    {
      "key": "@",
      "key_code": "Digit2",
      "timestamp": "2025-01-08T10:15:30.350Z",
```

```
      "modifiers": { "ctrl": false, "shift": true, "alt": false, "meta": false }
    }
  ],
  "final_text": "user@example.com",
  "timestamp": "2025-01-08T10:15:35.500Z"
}
```

Observer Endpoints

Observer endpoints (require sessionId):

- `POST /sessions/{session_id}/events` - Submit new browser event to specific session
- `GET /sessions/{session_id}/events/recent?limit=50` - Get recent events for session
- `GET /sessions/{session_id}/events/consume` - Consume unprocessed events (FIFO queue)
- `GET /sessions/{session_id}/events/unprocessed` - View unprocessed events for session
- `GET /sessions/{session_id}/events/processed?limit=50` - View processed events for session

Actor Channel

The Actor channel is fully implemented and operational. It enables AI systems to control the browser by sending commands that are executed as if a human user performed them.

Events in the Actor channel tell the browser to *do* something; it's the inverse of the Observer channel. Only load, mouse and keyboard events have effect in the Actor channel. Since we'd like to listen to the Observer then play back what we 'heard' on the Actor channel, other recognized events will be ignored. Detected errors will be reported on the Observer channel.

For tracking, Actor events (load, mouse, keyboard) will be echoed to the Observer channel, but marked so they can be distinguished from browser-operator events.

Actor Command Types

The Actor channel supports three types of commands:

1. **Navigation Commands** (`load`): Navigate to a specific URL
2. **Mouse Commands** (`click`): Click on elements by their label
3. **Keyboard Commands** (`keyboard`): Type text into form fields

The Actor channel is the inverse path to the Observer channel. As such, it accepts any observer event without error (for example, screen_status events) but does not act on all of them.

Actor API Endpoints

Actor endpoints (require sessionId):

- `POST /sessions/{session_id}/actor/send` - Queue command(s) for specific session
- `GET /sessions/{session_id}/actor/commands` - Poll pending commands for session (used by browser extension)
- `GET /sessions/{session_id}/actor/retrieved?limit=50` - View previously retrieved commands for session

- `GET /sessions/{session_id}/status` - Get session status including Actor command counts

Server status endpoints:

- `GET /status` - Server status including Actor command statistics

Commands are queued on the server and retrieved by the browser extension via polling every 100ms, creating a reliable command delivery system with session isolation.

Important: Browser Tab Visibility Requirement

Critical: The Actor channel requires the browser tab to be visible and active for proper operation. Chrome significantly reduces processing time allocated to hidden or background tabs, which causes Actor commands to hang or fail to execute.

When a tab is hidden, minimized, or in the background, Chrome throttles:

- JavaScript execution timing
- DOM updates and screen rendering
- Event processing and element detection

This browser-level performance optimization breaks the extension's ability to detect screen changes, locate elements, and execute commands with proper timing.

Always ensure the target browser tab is visible and active when using Actor channel commands or running automated tests.

Testing Infrastructure

The AI Browser Extension includes a comprehensive, unified testing infrastructure that validates both Observer and Actor channels through automated test suites.

Unified Architecture Achievement

After extensive development, we achieved **48/48 test success** using a unified codebase that works identically in both console and web environments. This eliminates the duplicate testing code that previously existed and provides a single, reliable testing foundation.

Core Testing Components

1. **TestingFramework.js** - Unified testing framework providing:
    - Session discovery and management
    - Navigation and element interaction methods
    - Observer stream processing and screen update detection
    - Actor command execution with proper timing
    - Configurable logging levels (ERROR, WARN, INFO, DEBUG)
2. **GenericElementTest.js** - Data-driven element testing system:
    - Specification-based element reading/writing
    - Support for all web control types (text, checkboxes, dropdowns, etc.)

    - Proper test isolation with navigation reset between tests
    - Error handling for missing elements and edge cases
3. **DataDrivenTestRunner.js** - Test suite execution engine:
    - Pure data-driven test configuration
    - Progress callbacks for real-time UI updates
    - Comprehensive suite and test-level reporting

Test Interfaces

Console Interface: `node data-driven-tests.js [--log-level=LEVEL]`

- Command-line execution with configurable verbosity
- Supports ERROR, WARN, INFO, and DEBUG logging levels
- Default INFO level reduces output from 1100+ lines to ~300 lines

Web Interface: `http://localhost:3001/test-runner`

- Real-time test execution with progress tracking
- Interactive logging level selection (persisted in localStorage)
- Visual test suite organization with expandable details
- Same 48/48 success rate using identical console infrastructure

Test Coverage

The unified test suite covers:

- **12 test suites** across different control types and scenarios
- **48 individual tests** covering form inputs, dropdowns, checkboxes, radio buttons, links, and buttons
- **Edge cases** including error handling, navigation isolation, and state management
- **Cross-browser compatibility** through the Generic User Framework approach

This testing infrastructure serves as both validation and demonstration of the AI Browser Extension's capability to interact with web interfaces exactly as humans do.

Testing Infrastructure Endpoints

The Python server provides comprehensive testing support through dedicated endpoints:

Test Pages:

- `GET /test-inputs` - Input fields test page for form validation
- `GET /test-controls` - Comprehensive controls test page with dropdowns, checkboxes, buttons
- `GET /test-result?action={action}` - Test result confirmation page

Test Runner Interface:

- `GET /test-runner` - Interactive web-based test runner interface
- `GET /test-suites-config` - JSON configuration data for 48 automated tests across 12 test suites

Testing Framework Libraries:

- `GET /TestingFramework.js` - Core testing framework for session management and UI interaction
- `GET /GenericElementTest.js` - Data-driven element testing system
- `GET /DataDrivenTestRunner.js` - Test suite execution engine with progress callbacks

Session Exploration:

- `GET /sessions/explorer` - Interactive session management and debugging interface

Note: Tests require browser tab visibility for proper execution (see Actor Channel section for details on Chrome's tab throttling behavior).

How do we think this will be used?

1. The first use of this extension will be to record normal browser sessions for analysis and evaluation. Possibly eventually to become AI training material.
2. The Observer capability could also be used to teach a nascent AI the basics of browser navigation.
3. Eventually, the Actor capacity could be used to close the loop providing success feedback to the learning AI.
4. The AI could be trained to read text on the screen and surmise the screen's purpose, projecting what's needed to complete the screen, or what screen to seek to accomplish some purpose.
5. With enough exposure, an AI might be able to predict 'I need to find these kinds of websites, to complete my quest.'

The above is very hand-wavey because it posits how an entity might discover how to do a job. The above is pretty linear and AI's famously are not, but the extension will give AI an entre no available today.

Research Foundations: Linguistic Structure and Function Discovery

AIBE is built on research findings that reveal how web interaction patterns can be understood through structural analysis. These observations enable AI systems to learn browsing behavior through pattern recognition rather than complex semantic analysis.

Linguistic Structure Observation

The user can go to a web site, click on certain things on the web page, enter data into fields, and the browser screen can update with new information. Those actions create events that we can see in the Observer channel of AIBE.

But how can we make sense of all this? There is structure to all these events, whether a user is immediately aware of it or not:

- Individual actions (clicks, keystrokes) form "words" with screen updates serving as breaks between words
- Interactions on a single page combine into "sentences," terminated by navigation to another page
- Site workflows become "paragraphs" - sequences of sentences within a single site
- Complete task sequences become "stories" - from start of task to task completion

So, a task solution consists of a Story, made up of one to many Paragraphs. A paragraph contains one or more Sentences, and Sentences are composed of Words.

Analysis reveals that user activity has a natural linguistic structure, without even considering what the user is trying to accomplish or the data they are entering (the *semantic* content!)

This understanding helps an AI observer assign meaning to web actions without regard to semantics, and to form hypotheses about how they are joined together to accomplish tasks.

Further, by looking at headings and titles on individual pages (Paragraphs), we can infer generally what the page is for, and narrow what the user could be doing on that page. Then we can look at the 'Words' they are choosing, and make an even better assessment of their goals.

By reviewing the analyzed Paragraphs of the Story we see the arc from where they started to where they ended. If we can guess the problem they were trying to solve (from the final Sentence of the last Paragraph), we can infer how they developed a solution to go from what they had (data they entered into fields) to what they were looking for (possibly the universe of all information that was displayed to them, but most likely what was shown on the final page at the last Sentence.)

Function Model Observation

By looking at each page and what was entered onto the page, we learn required inputs for a Sentence. From data displayed but not entered on the page, we learn the potential results of a Sentence. In essence we learn that if we want a certain output result, we can get it if we have this sentence and its required inputs.

This reveals that web pages function like programming functions - they accept inputs, transform them, and produce outputs. Having this for a number of Sentences (think subroutines) I can extrapolate how to go from the output I want to the inputs I have, and the sequence of Sentences between input and output.

Key insights from the Function Model:

- Cross-site pattern recognition and transfer learning
- Hypothesis-driven exploration of unknown information sources
- Path planning through function composition

Functions, inputs and outputs become a toolkit that can possibly be assembled to get desired outputs from the available inputs.

All this, without being able to name precisely what the user is doing!

In the bigger picture, once I know the universe of Sentences (functions) available from various websites, I could generate a path from some goal to required inputs, and the navigation path from input to output.

Which is precisely what a skilled operator learns to do with multiple websites, that they are familiar with!