I want to create a google chrome extension to help with filling out web forms. This extension should have a configuration file in the extension folder which contains slots for 10 strings. When the pinned icon on the toolbar is clicked, a popup list of these strings shown in buttons should be presented.

When one of the strings is clicked in the popup, the string should be placed into the currently selected field of the form which is displayed on the current webpage.

Some poorly designed forms require keyclicks or other user based input in the form even when the html data for the field has been set, add some simulated user behavior with keyclicks if possible, or transmit the data into the field by keyclicks or otherwise to stimulate javascript events that will mark the field as touched in this kind of application code.

Reasoned about Chrome extension design for 10 seconds >

Below is one way to implement such an extension. The idea is to have a JSON file with 10 preset strings, a popup that loads these strings as buttons, and a content script that, when triggered, simulates typing the selected string into the currently focused form field.

Below is an outline of the files you might include:

---

1. manifest.json

This file defines your extension. Note that we use Manifest V3 and request the activeTab and scripting permissions. A content script is injected into all pages so that it can receive messages from the popup.

json Copy

```json
{
  "manifest_version": 3,
  "name": "Form Filler Extension",
  "version": "1.0",
  "description": "Fill web forms with preset strings and simulate typing events.",
  "action": {
    "default_popup": "popup.html"
  },
  "permissions": ["activeTab", "scripting"],
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content_script.js"]
    }
  ]
}
```

---

2. config.json

Place this file in your extension folder. It simply contains an array of 10 strings (modify these as needed):

```json
{
  "strings": [
    "First preset text",
    "Second preset text",
    "Third preset text",
    "Fourth preset text",
    "Fifth preset text",
    "Sixth preset text",
    "Seventh preset text",
    "Eighth preset text",
    "Ninth preset text",
    "Tenth preset text"
  ]
}
```

---

3. popup.html

This is the popup that appears when the user clicks the pinned icon. It includes a container where the buttons will be dynamically created.

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Form Filler</title>
  <script src="popup.js"></script>
</head>
<body>
  <div id="buttonContainer"></div>
</body>
</html>
```

---

4. popup.js

This script loads your `config.json`, creates buttons for each string, and sends a message to the content script with the selected string when a button is clicked.

```js
document.addEventListener("DOMContentLoaded", async () => {
  // Load the configuration file from the extension folder.
  const response = await fetch(chrome.runtime.getURL('config.json'));
  const config = await response.json();

  const container = document.getElementById("buttonContainer");
  config.strings.forEach((str) => {
    const btn = document.createElement("button");
    btn.textContent = str;
    btn.style.display = "block";  // arrange buttons in a column
    btn.style.margin = "5px";
    btn.addEventListener("click", () => {
      // Query the active tab and send the selected string.
      chrome.tabs.query({active: true, currentWindow: true}, (tabs) => {
        chrome.tabs.sendMessage(tabs[0].id, { action: "fill", text: str });
      });
    });
    container.appendChild(btn);
  });
});
```

---

5. content_script.js

This script listens for messages from the popup. When it receives a request to “fill” a field, it checks for the currently focused element (like an `<input>` or `<textarea>`). It then simulates typing by iterating through each character and dispatching key events (keydown, keypress, input, and keyup). This approach should trigger any JavaScript listeners on the page that require key-based input.

```js
chrome.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.action === "fill") {
    const activeElement = document.activeElement;
    if (activeElement && (activeElement.tagName === "INPUT" || activeElement.tagName === "TEX
      activeElement.focus();
      // Optionally, clear any existing value:
      activeElement.value = "";

      // Simulate typing the string character by character.
      for (let i = 0; i < message.text.length; i++) {
        const char = message.text[i];
        // Append character to the current value.
        activeElement.value += char;

        // Create and dispatch keydown event.
        const keyDownEvent = new KeyboardEvent("keydown", { key: char, bubbles: true });
        activeElement.dispatchEvent(keyDownEvent);

        // Create and dispatch keypress event.
        const keyPressEvent = new KeyboardEvent("keypress", { key: char, bubbles: true });
        activeElement.dispatchEvent(keyPressEvent);

        // Create and dispatch input event to indicate change.
        const inputEvent = new Event("input", { bubbles: true });
        activeElement.dispatchEvent(inputEvent);

        // Create and dispatch keyup event.
        const keyUpEvent = new KeyboardEvent("keyup", { key: char, bubbles: true });
        activeElement.dispatchEvent(keyUpEvent);
      }

      // Optionally dispatch a change event at the end.
      const changeEvent = new Event("change", { bubbles: true });
      activeElement.dispatchEvent(changeEvent);
    } else {
      alert("Please select an input or textarea element before using the form filler.");
    }
  }
});
```

Explanation

- **Configuration & Popup:**
  The `config.json` holds your strings. The popup (popup.html and popup.js) fetches these strings and creates a list of buttons. When a button is clicked, a message is sent to the content script in the active tab.
- **Content Script & Event Simulation:**
  The content script listens for the "fill" message. It then verifies that the active element is an input or textarea. For each character in the selected string, the script simulates:
    - **Keydown and keypress events:** These events mimic the physical act of pressing a key.
    - **Input event:** Dispatching this event informs the page that the field's value has changed.
    - **Keyup event:** This simulates releasing the key.

  This sequence is intended to trigger any event listeners that the webpage might have (especially in cases where simply setting the value directly would not be enough).

Further Considerations

- **Typing Speed:**
  For a more natural simulation, you might consider adding a delay between characters (using `setTimeout` or `async/await` with a delay) so the events don't fire too rapidly. This can help with pages that expect a human-like pace of input.
- **Element Verification:**

If your target fields are more diverse (e.g., contentEditable elements), you might need to adjust the checks in the content script.

- **Error Handling:**
  You might want to improve user feedback (for example, if no suitable field is focused).

This should give you a complete starting point for your extension. You can then package your extension (using Chrome's Developer mode) and test it on various webpages to ensure the simulated keystrokes trigger the necessary events.

Nothing is happening in the particular form when the text button in the popup is selected. Can you make the process that fills in the field more robust or use multiple techniques to set the content text to improve reliability for different types of forms.

Reasoned about improving field filling reliability for 13 seconds >

Below is an updated version of the content script that uses multiple techniques to insert text into the active form field. This version does the following:

- **Simulated Typing:**
  It "types" the text character-by-character with delays while dispatching keyboard events (keydown, keypress, input, keyup) to mimic human input.
- **Direct Assignment:**
  After simulated typing, it uses a direct assignment method (including use of the native `setRangeText` if available) to update the field's value. This fallback helps trigger listeners on forms that require both methods to mark the field as "touched."
- **Support for Different Elements:**
  It works for `<input>`, `<textarea>`, and contentEditable elements.

Below is the improved `content_script.js`:

```js
chrome.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.action === "fill") {
    const activeElement = document.activeElement;
    if (activeElement && (activeElement.tagName === "INPUT" || activeElement.tagName === "TEX
      activeElement.focus();

      // Utility function to dispatch a keyboard event for a given type and character.
      function dispatchKeyboardEvent(type, char) {
        const event = new KeyboardEvent(type, { key: char, bubbles: true, cancelable: true })
        activeElement.dispatchEvent(event);
      }

      // Approach 1: Simulate typing by dispatching events per character.
      function simulateTyping(text, index = 0) {
        if (index < text.length) {
          const char = text[index];
        // Dispatch keydown and keypress events.
        dispatchKeyboardEvent("keydown", char);
        dispatchKeyboardEvent("keypress", char);

        // Use setRangeText if available for a more native behavior.
        if ("setRangeText" in activeElement && typeof activeElement.setRangeText === "funct
          activeElement.setRangeText(char, activeElement.selectionStart, activeElement.sele
        } else if ("value" in activeElement) {
          activeElement.value += char;
        } else if (activeElement.isContentEditable) {
          activeElement.innerHTML += char;
        }

        // Dispatch an input event after updating the value.
        activeElement.dispatchEvent(new Event("input", { bubbles: true }));
        dispatchKeyboardEvent("keyup", char);

        // Delay before typing the next character to simulate a more natural pace.
        setTimeout(() => simulateTyping(text, index + 1), 50);
      } else {
        // After finishing simulated typing, dispatch a change event.
        activeElement.dispatchEvent(new Event("change", { bubbles: true }));
      }
    }

    // Approach 2: Directly assign the full text as a fallback.
    function directAssignment(text) {
      if ("value" in activeElement) {
        activeElement.value = text;
      } else if (activeElement.isContentEditable) {
        activeElement.innerHTML = text;
      }
      activeElement.dispatchEvent(new Event("input", { bubbles: true }));
      activeElement.dispatchEvent(new Event("change", { bubbles: true }));
    }

    // Clear the field before starting.
    if ("value" in activeElement) {
      activeElement.value = "";
    } else if (activeElement.isContentEditable) {
      activeElement.innerHTML = "";
    }
    activeElement.dispatchEvent(new Event("input", { bubbles: true }));

    // Start with simulated typing.
    simulateTyping(message.text);

    // After a delay based on the text length, perform direct assignment as a fallback.
    setTimeout(() => {
      directAssignment(message.text);
    }, message.text.length * 60); // delay scales with text length

  } else {
    alert("Please select an input, textarea, or contentEditable element before using the fo
  }
  }
});
```

Explanation

1. **Dispatching Keyboard Events:**
   The `dispatchKeyboardEvent` function creates and dispatches keyboard events (keydown, keypress, keyup) for each character.
2. **Simulated Typing (Approach 1):**
   - The `simulateTyping` function loops over the string and simulates typing one character at a time.
   - It uses `setRangeText` if available to mimic native editing behavior.
   - Between each character, a short delay (50 ms) is introduced so the events occur at a more human-like pace.
   - After typing is done, an extra change event is fired.
3. **Direct Assignment (Approach 2):**
   - After simulated typing, a direct assignment method sets the field's value or innerHTML with the entire string.
   - This method also dispatches input and change events, ensuring that listeners that require a full value change are triggered.
4. **Field Clearing:**
   The script clears the existing content before starting the simulated input.

By combining these methods, the extension is more likely to interact successfully with different kinds of forms—even those that need extra "nudge" via events to register that a field has been updated.

Test this updated content script with your extension. Depending on the behavior of the target webpage, you might tweak the delays or add further event dispatches.

This still isn't working i

Finished reasoning >

I think focus is lo n the main form when the popup in the tool tip is displayed

Reasoned for 8 seconds >

The issue is likely that when you click the extension's popup, it steals focus from the page, so by the time the content script receives the message, the active element isn't what you expected. One workaround is to have the content script track the last focused element on the page. Then, when the popup sends the fill command, you use that stored element instead of relying on document.activeElement.

Below is an updated version of the content script that:

1. **Tracks the Last Focused Element:**
   A global variable (`lastFocusedElement`) is updated via a capturing focus event so that even if focus is lost when the popup opens, you still have a reference.
2. **Uses Multiple Insertion Techniques:**
   It simulates typing (with keyboard events) and then falls back to a direct assignment.
3. **Supports Input, Textarea, and contentEditable Elements:**
   It works for various editable fields.

Updated content_script.js

```js
// Global variable to store the last focused editable element.
let lastFocusedElement = null;

// Listen for focus events in the capturing phase.
document.addEventListener(
  "focus",
  (event) => {
    const target = event.target;
    if (
      target &&
      (target.tagName === "INPUT" ||
        target.tagName === "TEXTAREA" ||
        target.isContentEditable)
    ) {
      lastFocusedElement = target;
    }
  },
  true
);

chrome.runtime.onMessage.addListener((message, sender, sendResponse) => {
  if (message.action === "fill") {
    // Use the stored lastFocusedElement if available.
    const activeElement =
      lastFocusedElement ||
      (document.activeElement &&
        (document.activeElement.tagName === "INPUT" ||
         document.activeElement.tagName === "TEXTAREA" ||
         document.activeElement.isContentEditable)
        ? document.activeElement
        : null);

    if (activeElement) {
```

```
if (activeElement) {
  // Re-focus the element.
  activeElement.focus();

  // Utility function to dispatch keyboard events.
  function dispatchKeyboardEvent(type, char) {
    const event = new KeyboardEvent(type, {
      key: char,
      bubbles: true,
      cancelable: true,
    });
    activeElement.dispatchEvent(event);
  }

  // Approach 1: Simulated typing.
  function simulateTyping(text, index = 0) {
    if (index < text.length) {
      const char = text[index];

      dispatchKeyboardEvent("keydown", char);
      dispatchKeyboardEvent("keypress", char);

      if ("setRangeText" in activeElement && typeof activeElement.setRangeText === "funct
        activeElement.setRangeText(
          char,
          activeElement.selectionStart,
          activeElement.selectionEnd,
          "end"
        );
      } else if ("value" in activeElement) {
        activeElement.value += char;
      } else if (activeElement.isContentEditable) {
        activeElement.innerHTML += char;
      }

      activeElement.dispatchEvent(new Event("input", { bubbles: true }));
      dispatchKeyboardEvent("keyup", char);

      setTimeout(() => simulateTyping(text, index + 1), 50);
    } else {
      activeElement.dispatchEvent(new Event("change", { bubbles: true }));
    }
  }

  // Approach 2: Direct assignment fallback.
  function directAssignment(text) {
    if ("value" in activeElement) {
      activeElement.value = text;
    } else if (activeElement.isContentEditable) {
      activeElement.innerHTML = text;
    }
    activeElement.dispatchEvent(new Event("input", { bubbles: true }));
    activeElement.dispatchEvent(new Event("change", { bubbles: true }));
  }
```

```
      // Clear existing content.
      if ("value" in activeElement) {
        activeElement.value = "";
      } else if (activeElement.isContentEditable) {
        activeElement.innerHTML = "";
      }
      activeElement.dispatchEvent(new Event("input", { bubbles: true }));

      // Start with simulated typing.
      simulateTyping(message.text);

      // Fallback direct assignment after a delay.
      setTimeout(() => {
        directAssignment(message.text);
      }, message.text.length * 60);
    } else {
      alert("No valid input field is focused. Please click on a form field first.");
    }
  }
});
```

How It Works

- **Tracking Focus:**
  The script listens to all focus events (in the capturing phase) on the page and stores any element that is an `<input>`, `<textarea>`, or is contentEditable. This way, even if the popup steals focus, you still have a reference to the last focused element.
- **Simulated Typing:**
  The `simulateTyping` function loops through the provided text, dispatching keydown, keypress, input, and keyup events for each character. This simulates natural typing.
- **Direct Assignment Fallback:**
  After simulated typing (with a delay that scales with text length), the script also directly assigns the text to the field. This ensures that if simulated t[illegible]g doesn't trigger the expected event listeners (or if the field resets), the text is still set.
- **Refocusing the Field:**
  On receiving the fill command, the script calls `activeElement.focus()` on the stored element to attempt to bring focus back to the correct field.

Additional Considerations

- **UI Alternatives:**
  If focus loss continues to be a problem, consider using an in-page overlay (injected via a content script) rather than a separate popup. This can prevent the main window from losing focus.
- **Delays and Timing:**

Adjust the delay values (e.g., 50 ms per character) as needed to suit the form's behavior.

Try updating your extension with this revised content script. This approach should improve reliability by ensuring the intended form field is targeted, even if the popup temporarily steals focus.

That worked although I had to ctrl-r hard reload the web page when the extension was reloaded so that the state tracking would work. please explain how the state tracking only occurs on freshly loaded pages.

Reasoned about focus state tracking for 11 seconds >

When an extension is reloaded, the new version of your content script isn't automatically injected into pages that are already open. Instead, the content script that was originally loaded when the page first loaded continues running in that page's context. That script set up the state tracking (using focus event listeners) and stored the last focused element in a global variable.

When you hard reload (Ctrl+R) the page, the browser re-injects the content script, meaning that the new version of your focus event listener is attached and the tracking state starts fresh. This is why the state tracking works reliably only on pages loaded after the extension (or the page) is reloaded.

Ask anything

+ Search Deep research

ChatGPT can make mistakes. Check important info.

?