

Servo DevTools: Where Rust Meets JavaScript

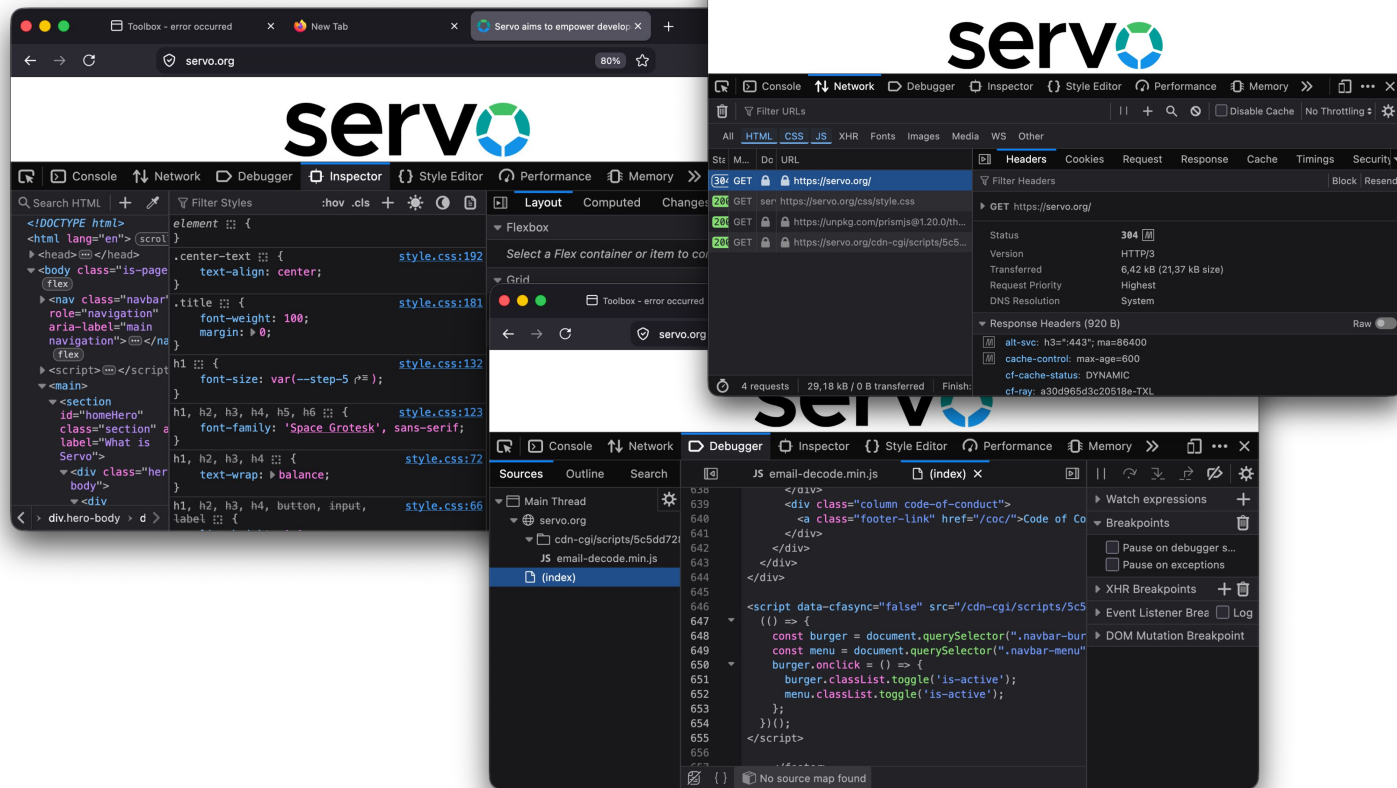
Rakhi Sharma (atbrakhi)

10.09.2026



I used Devtool for many years

- Inspector
- Console
- Debugger
- Network
- Performance/ memory



Fast forward: the tables turned



I was suddenly building DevTools

... inside a rendering engine written in Rust



Who am I?

- Mostly known online as **@atbrakhi**
- Servo maintainer and TSC member
- Working at Igalia as a browser engineer
- Mostly hacking on Servo and Chromium



Servo Rendering Engine

- Web rendering engine
- Written in Rust
- Desktop, mobile, and embedded applications
- WebGL and WebGPU support
- Now with DevTools



Servo DevTools

- Inspector
- Console
- Debugger
- Network
- Style Editor (basic support)
- ~~Performance and Memory~~ (not yet)



Servo DevTools ← RDP → Firefox DevTools

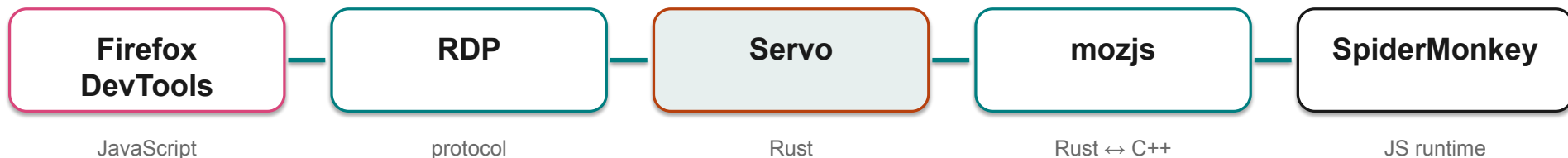
**Servo
DevTools**



**Firefox
DevTools**



Servo Devtools: Where Rust Meets JavaScript



A breakpoint request crosses all of these layers



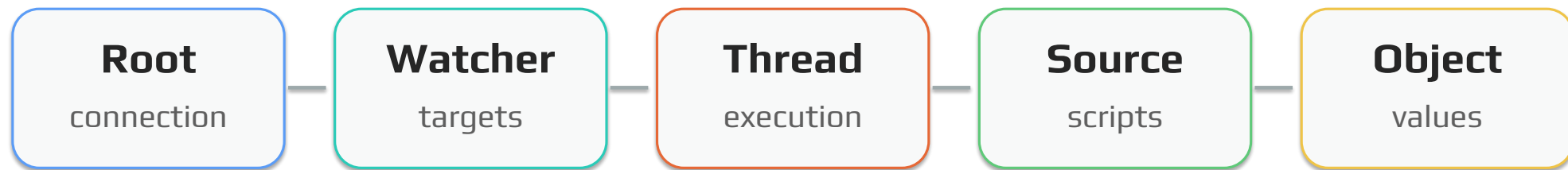
Servo Devtool reuse Firefox DevTool UI



Servo implements the protocol the Firefox DevTools client already understands



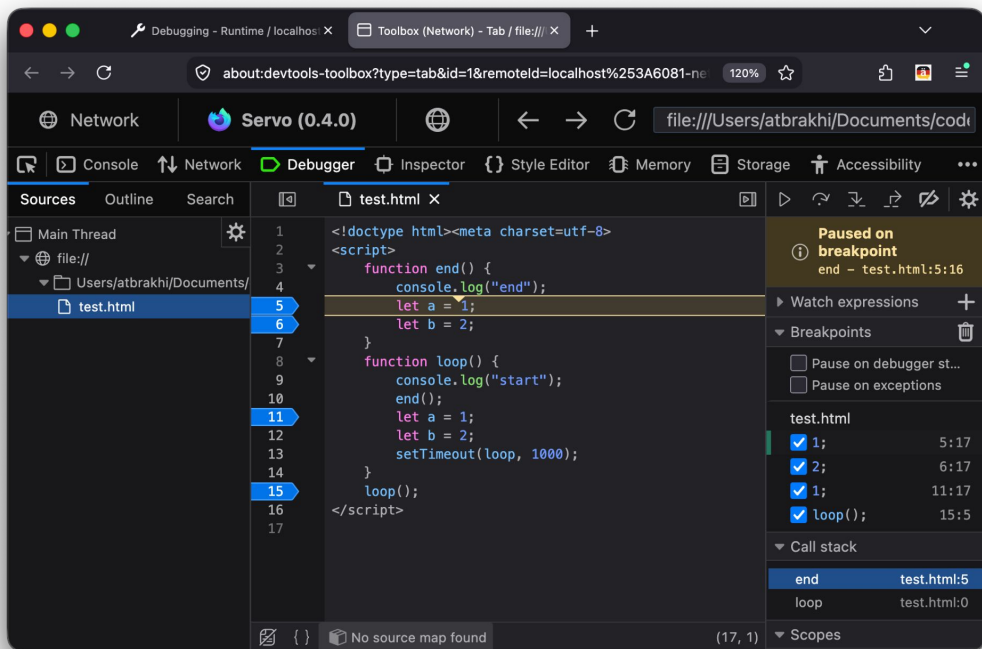
RDP is organized around actors



Different parts of Firefox DevTools send requests to different actors.



You set a breakpoint in the Debugger tab



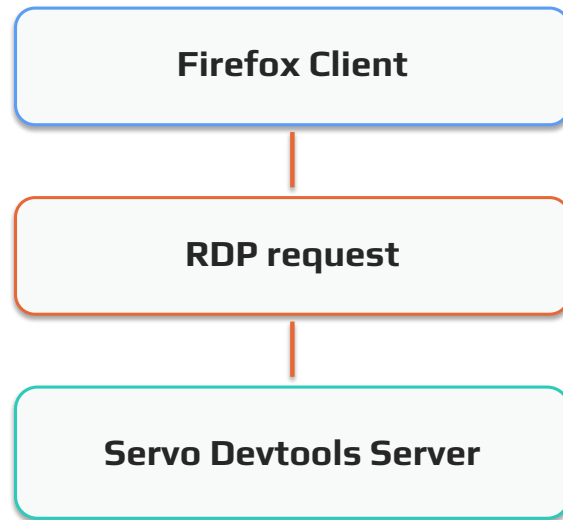
- What do we see in the Debugger tab after hitting the breakpoint? Scopes, the call stack, file names, pause status, and more.
- The DevTools client needs to know where the user is allowed to set a breakpoint. How?
- See the lines where breakpoints are disabled? Why are they disabled?



SetBreakpoint: What happens behind the scenes?

- Firefox sends a packet via RDP.
- It is addressed to the relevant actor—in this case, the “breakpoint-list” actor
- Servo receives it, processes it, and responds via RDP.

```
{  
  "to": "BreakpointListActor19",  
  "location": {  
    "sourceUrl": "test.html",  
    "line": 8,  
    "column": 14  
  },  
  "type": "setBreakpoint"  
}
```



Let's look at the packet the server receives from the client

to

the breakpoint-list actor

location

URL + line + column

type

setBreakpoint

```
{  
  "to": "BreakpointListActor19",  
  "location": {  
    "sourceUrl": "test.html",  
    "line": 8,  
    "column": 14  
  },  
  "type": "setBreakpoint"  
}
```



Servo receives a packet. What happens next?

01

Find the Source

02

Try to resolve line +
column → script ID +
offset

03

send "SetBreakpoint"
to the script thread

```
components/devtools/actors/breakpoint.rs · BreakpointListActor

1 let source_url = source_url.ok_or(ActorError::Internal)?;
2 let browsing_context_actor =
3     registry.find:<BrowsingContextActor>(&self.browsing_context_name);
4 let thread_actor =
5     registry.find:<ThreadActor>(&browsing_context_actor.thread_name);
6 let source = thread_actor
7     .source_manager
8     .find_source(registry, &source_url)
9     .ok_or(ActorError::Internal)?;
10
11 if let Some((script_id, offset)) = source.find_offset(line, column) {
12     source
13         .script_sender
14         .send(DevtoolScriptControlMsg::SetBreakpoint(
15             source.spidermonkey_id,
16             script_id,
17             offset,
18         ))
19         .map_err(|_| ActorError::Internal)?;
20 }
```



Now the script thread has to ask the JavaScript engine to set the breakpoint

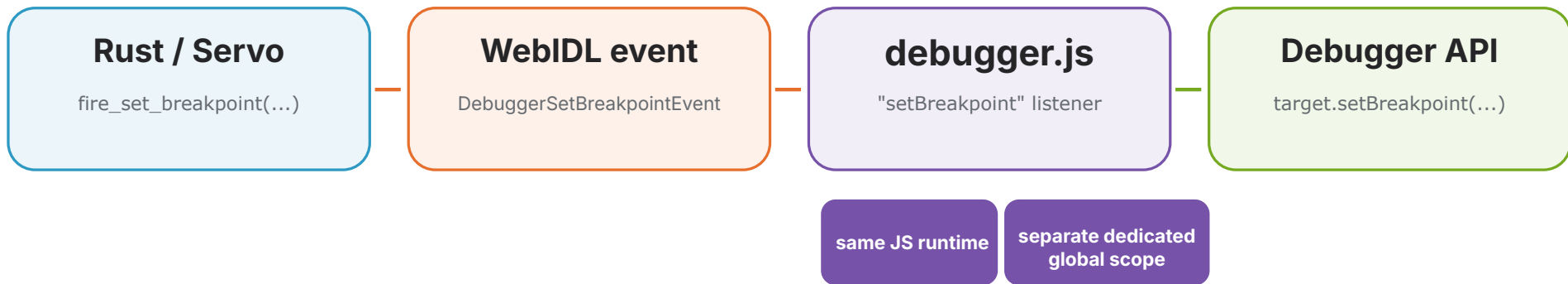
Servo has two different JavaScript integration paths:

1. Native SpiderMonkey APIs through mozjs / JSAPI, Console features uses this.
2. Debugger features use SpiderMonkey's Debugger API through debugger.js. (Debugger tab does this)



debugger.js is the bridge for the Debugger tab

Servo does not call the SpiderMonkey Debugger API directly.



Servo dispatches events to debugger.js.
debugger.js calls the SpiderMonkey Debugger API and invokes WebIDL methods to communicate back with Rust.

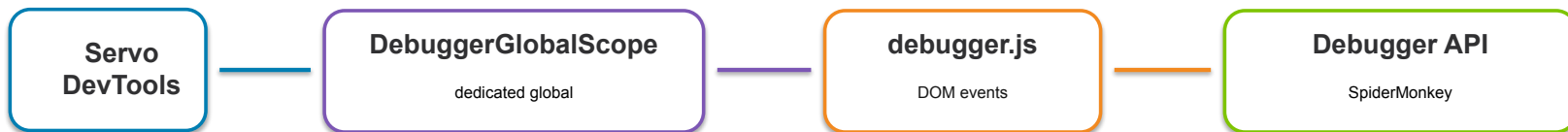


debugger.js is the RPC layer

RUST ↔ JAVASCRIPT

```
components/shared/embedder/resources.rs  
  
131 /// RPC script for the Debugger API on behalf of devtools.  
132 DebuggerJS,
```

```
components/script/dom/debugger/debuggerglobalscope.rs  
  
143 pub(crate) fn execute(&self, cx: &mut JSContext) {  
144     let mut realm = enter_auto_realm(cx, self);  
145     let cx = &mut realm.current_realm();  
146  
147     let _ = self.global_scope.evaluate_js_on_global(  
148         cx,  
149         resources::read_string(Resource::DebuggerJS).into(),  
150         "",  
151         None,  
152         None,  
153     );  
154 }
```



Before setting the breakpoint, Servo asks which locations are breakable

RUST → DEBUGGER.JS

```
components/devtools/actors/source.rs

265 pub fn find_offset(&self, line: u32, column: u32) -> Option<(u32, u32)> {
266     let (tx, rx) = channel()?;
267     self.script_sender
268         .send(DevtoolScriptControlMsg::GetPossibleBreakpoints(
269             self.spidermonkey_id,
270             tx,
271         ))
272         .ok()?;
273     let result = rx.recv().ok()?;
274     for location in result {
275         // Line number are one-based. Column numbers are zero-based.
276         // FIXME: the docs say column numbers are one-based, but this appears to be incorrect.
277         // <https://firefox-source-docs.mozilla.org/devtools/backend/protocol.html#source-locations>
278         if location.line_number == line && location.column_number - 1 == column {
279             return Some((location.script_id, location.offset));
280         }
281     }
282     None
283 }
```

```
components/script/event_loop/script_thread.rs

2146 DevtoolScriptControlMsg::GetPossibleBreakpoints(spidermonkey_id, result_sender) => {
2147     self.debugger_global.fire_get_possible_breakpoints(
2148         cx,
2149         spidermonkey_id,
2150         result_sender,
2151     );
2152 },
```

GetPossibleBreakpoints

SourceActor blocks on the reply, then matches line + column.



debugger.js walks the Debugger.Script tree

```
components/default-resources/resources/debugger.js

416 addEventListener("getPossibleBreakpoints", event => {
417     const {spidermonkeyId} = event;
418     const script = sourceIdsToScripts.get(spidermonkeyId);
419     const result = [];
420     walkScriptTree(script, (currentScript) => {
421         for (const location of currentScript.getPossibleBreakpoints()) {
422             location["scriptId"] = currentScript.sourceStart;
423             result.push(location);
424         }
425     });
426     getPossibleBreakpointsResult(event, result);
427 });
```

One source can contain multiple Debugger.Script objects.

getPossibleBreakpoints()



scriptId + offset

Those are the coordinates Servo can use to set the breakpoint.



ScriptThread turns SetBreakpoint into a debugger.js event

components/script/event_loop/script_thread.rs

```
2153 DevtoolScriptControlMsg::SetBreakpoint(spidermonkey_id, script_id, offset) => {  
2154     self.debugger_global  
2155         .fire_set_breakpoint(cx, spidermonkey_id, script_id, offset);  
2156 },
```

ScriptThread

DevtoolScriptControlMsg::SetBreakpoint

components/script/dom/debugger/debuggerglobalscope.rs

```
232 pub(crate) fn fire_set_breakpoint(  
233     &self,  
234     cx: &mut JSContext,  
235     spidermonkey_id: u32,  
236     script_id: u32,  
237     offset: u32,  
238 ) {  
239     let event = DomRoot::upcast::<Event>(DebuggerSetBreakpointEvent::new(  
240         cx,  
241         self.upcast(),  
242         spidermonkey_id,  
243         script_id,  
244         offset,  
245     ));  
246     assert!(  
247         event.fire(cx, self.upcast()),  
248         "Guaranteed by DebuggerSetBreakpointEvent::new"  
249     );  
250 }
```

DebuggerSetBreakpointEvent



Pausing JavaScript must not pause DevTools

NATIVE BRIDGE + NESTED LOOP

```
components/script/dom/debugger/debuggerglobalscope.rs

557 let pause_reason = devtools_traits::PauseReason {
558     type_: pause_reason.type_.clone().into(),
559     on_next: pause_reason.onNext,
560 };
561 if let Some(chan) = self.upcast::GlobalScope().devtools_chan() {
562     let msg =
563         ScriptToDevtoolsControlMsg::DebuggerPause(pipeline_id, frame_offset, pause_reason);
564     let _ = chan.send(msg);
565 }
566
567 with_script_thread(|script_thread| {
568     script_thread.enter_debugger_pause_loop();
569 });
```

```
components/script/event_loop/script_thread.rs

2187 /// Enter a nested event loop for debugger pause.
2188 /// TODO: This should also be called when manual pause is triggered.
2189 pub(crate) fn enter_debugger_pause_loop(&self) {
2190     self.debugger_paused.set(true);
2191
2192     #[allow(unsafe_code)]
2193     let mut cx = unsafe { js::context::JSContext::from_ptr(js::rust::Runtime::get().unwrap()) };
2194     while self.debugger_paused.get() {
2195         match self.receivers.devtools_server_receiver.recv() {
2196             Ok(Ok(msg)) => self.handle_msg_from_devtools(msg, &mut cx),
2197             _ => {
2198                 self.debugger_paused.set(false);
2199                 break;
2200             },
2201         }
2202     }
2203 }
```

DebuggerPause
message

enter_debugger_pause_loop()

receive DevTools
messages



debugger.js sets the actual breakpoint

```
atbrakhi, 5 months ago • devtools: implement listing all frames (#43015)...
addEventListener("setBreakpoint", event => {
  const {spidermonkeyId, scriptId, offset} = event;
  const script = sourceIdsToScripts.get(spidermonkeyId);
  const target = findScriptById(script, scriptId);
  if (target) {
    target.setBreakpoint(offset, {
      // setBreakpoint(offset, handler) in <https://firefox-source-docs.mozilla.org/devtools-user/debugger-api/debugger.script/index.html#function-properties-of-the-debugger-script-prototype-object>
      // The hit handler receives a Debugger.Frame instance representing the currently
      // executing stack frame.
      hit: (frame) => handlePauseAndRespond(frame, {type_: "breakpoint"})
    });
  }
});
```



The actor replies to the client

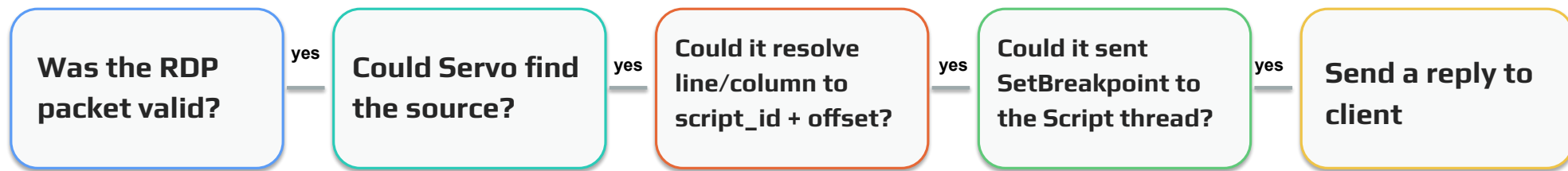
```
components/devtools/actors/breakpoint.rs  
  
77 let msg = EmptyReplyMsg {  
78     from: self.name().into(),  
79 };  
80 request.reply_final(&msg)?
```

```
{ "from" : "BreakpointListActor19" }
```

Empty means “no result payload.” It acknowledges the RDP request. Reaching this would also mean no earlier operation returned `ActorError`.



At the end flow looks like this



At a high level, many DevTools request/response flows follow a similar pattern.

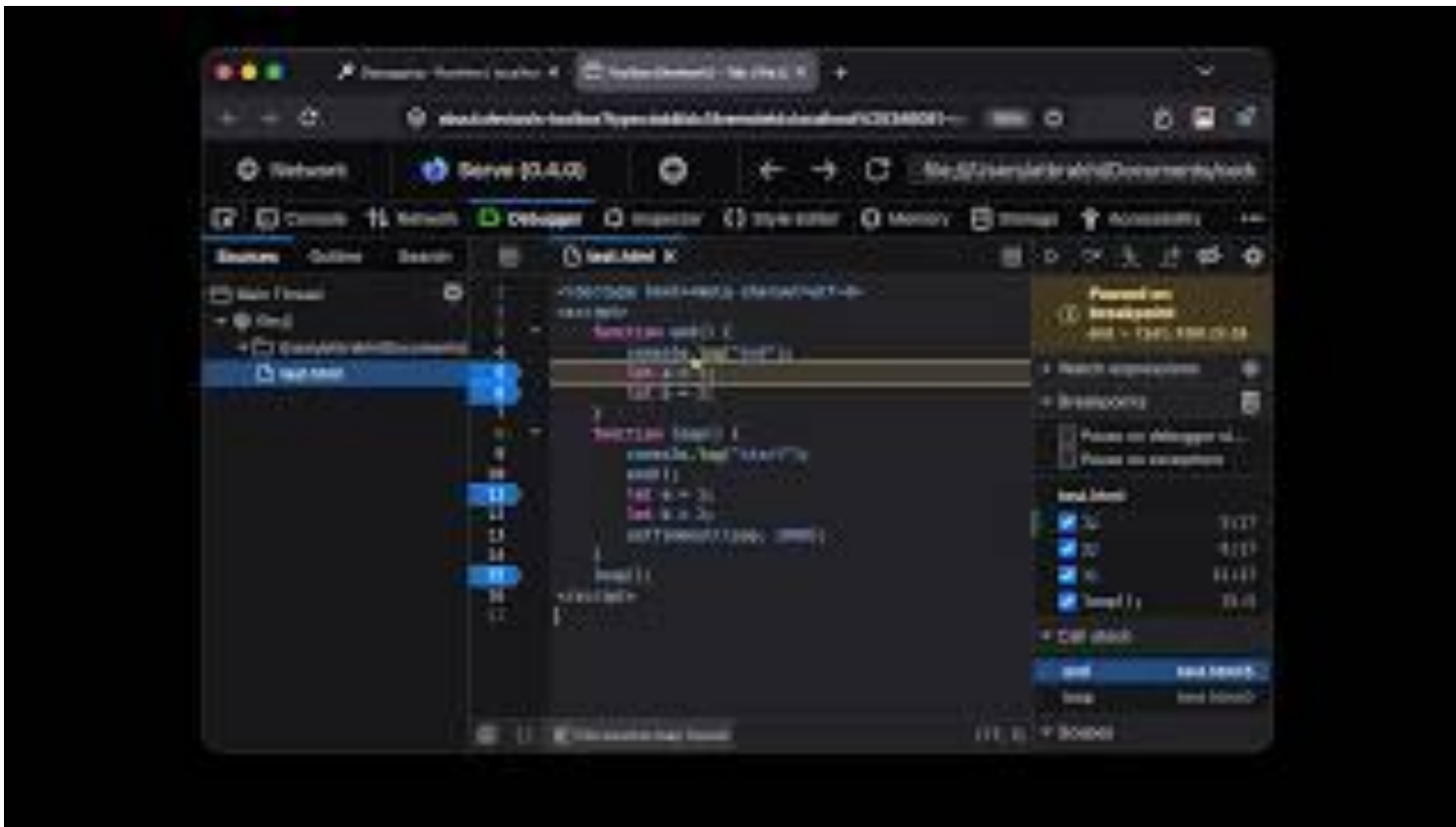


Some interesting facts

- `console.log(a + b)` is not the same as evaluating `a + b`
- Pausing JavaScript cannot mean pausing DevTools
- Before setting a breakpoint, DevTools first has to discover where a breakpoint is possible
- A line of JavaScript is not automatically a valid breakpoint location
- One JavaScript file is not one debugger script
- Console, Debugger, Inspector are different tabs, but they share lots of logic behind the scene



Demo



Thank you <3

