Connecting Fiddler to Gemini via MCP

This document explains the bridge data flow, I followed one data path end to end to show the exact functions that move a captured web session from Fiddler's memory into the Gemini model.

I focused on a single tool to keep it simple session_headers but every other endpoint (response bodies, stats, timeline, etc.) is a variation on the same pattern. I got 7 tools implemented in total.

The MCP tools are simply server endpoints, accessed by unique PATHs just like websites on a HTTP server. I use python - flask server here for simplicity.

High level

1. **Fiddler** converts captured sessions → JSON → HTTP POST on a flask server.
2. **enhanced-bridge.py** stores JSON and exposes endpoints like /api/sessions/headers/
3. **5ire-bridge.py** provides MCP tools like fiddler_mcp__session_headers → REST call.
4. **Gemini client** calls the tool, injects the JSON obtained from bridge into a Gemini prompt, and the LLM explains the headers back to the analyst as per normal chat flow.

1. Fiddler captures traffic and publishes JSON

Fiddler keeps sessions in memory. To make them accessible outside the UI we need to push each completed session to the local HTTP server. The CustomRules script does this immediately after every response by calling `McpTryPost` using the javascript.

```
// CustomRules.js
static function McpTryPost(oSession: Session): void {
    try {
        // Skip tunnels or sessions with no HTTP response
        if ((oSession.oResponse == null) || (oSession.responseCode == 0)) return;

        var json: String = McpBuildSimpleJson(oSession);
        McpHttpPost(json);
    } catch (e) {
        FiddlerApplication.Log.LogString("MCP error: " + e.Message);
    }
}
```

Key idea: McpBuildSimpleJson condenses the session (request line, headers, body, risk flags) and McpHttpPost sends it to our staging server at http://127.0.0.1:8081/live-session.

2. enhanced-bridge.py stages the data

The staging HTTP server buffers the JSON in a ring buffer and exposes REST endpoints. For the header example, the GET route below looks up the session and returns its request/response headers.

```
# enhanced-bridge.py
@self.app.route('/api/sessions/headers/<session_id>', methods=['GET'])
def get_session_headers(session_id):
    """Get headers for specific session"""
    try:
        with self.session_lock:
            for session in reversed(self.live_sessions):
                if str(session.get('id', '')) == str(session_id):
                    return jsonify({
                        "success": True,
                        "session_id": session_id,
                        "request_headers": session.get('requestHeaders', {}),
                        "response_headers": session.get('responseHeaders', {}),
                        "found": True
                    })

        return jsonify()
```

Key idea: we keep everything accessible through plain HTTP so any tool (curl, MCP server, browser) can fetch the data like a normal web API.

3. 5ire-bridge.py exposes MCP tools

The MCP bridge translates model tool invocations into REST calls. The helper client calls the endpoint above.

```
 # 5ire-bridge.py (FiddlerBridgeClient helper)
def get_session_headers(self, *, session_id: str) -> Dict[str, Any]:
    try:
        data = self.request("GET", f"/api/sessions/headers/{session_id}")
    except BridgeConnectionError:
        return {
            "success": False,
            "error": "Cannot connect to real-time bridge",
            "bridge_status": "Disconnected",
        }
    except BridgeRequestError as exc:
        return {
            "success": False,
            "error": f"Header lookup failed: {exc}",
            "session_id": session_id,
        }

    if not isinstance(data, dict) or not data.get("success", False):
        return {
            "success": False,
            "error": data.get("error", "Headers not available") if isinstance(data, dict) else "Unexpected response",
            "session_id": session_id,
        }

    return {
        "success": True,
        "session_id": session_id,
        "request_headers": data.get("request_headers", {}),
        "response_headers": data.get("response_headers", {}),
        "notes": [
            "Use these headers to reason about authentication, caching, and security controls yourself.",
        ],
    }
```

That helper is wired to the actual tool definition that the LLM sees:

```
 # 5ire-bridge.py (MCP tool)
@mcp.tool()
def fiddler_mcp__session_headers(
    session_id: Annotated[str, Field(description="Session ID from live_sessions or sessions_search.")],
) -> Dict[str, Any]:
    """Fetch ONLY the HTTP headers (NOT the body content) for a captured session."""

    return client.get_session_headers(session_id=session_id)
```

Key idea: The tool name fiddler_mcp__session_headers is what shows up in tools/list. Calling it ultimately invokes the REST endpoint above.

4. gemini-fiddler-client.py hands data to the model

The Gemini client launches 5ire-bridge.py, calls the tool, and when the result arrives it builds a new prompt that contains the JSON payload. This is where the bridge output becomes part of the LLM conversation.

```python
 # gemini-fiddler-client.py (excerpt from chat loop)
tool_result = self.call_tool(tool_name, arguments)

# Add tool result to history so it can be reused
self.conversation_history.append({
    "role": "tool",
    "tool": tool_name,
    "content": json.dumps(tool_result, indent=2)
})

# Ask Gemini to analyze the tool result
analysis_prompt = f"""The tool '{tool_name}' returned this result:

{json.dumps(tool_result, indent=2)}

Please analyze this result and answer the user's original question: "{user_query}"

Provide a clear, concise analysis. If you need to call another tool, respond with the same JSON format."""

analysis_response = self.model.generate_content(analysis_prompt)
```

whatever the bridge returns (headers, bodies, stats) becomes part of the prompt that is sent to the Gemini API. Different tools simply plug different JSON snippets into the same pattern.