Web Agents

dglazkov@
last updated: 2017/5/11
status: draft

Web Agent is an implementation pattern for high-level browser features implemented on top of the Web Platform: e.g. Translate, Autofill, Password Manager, etc.

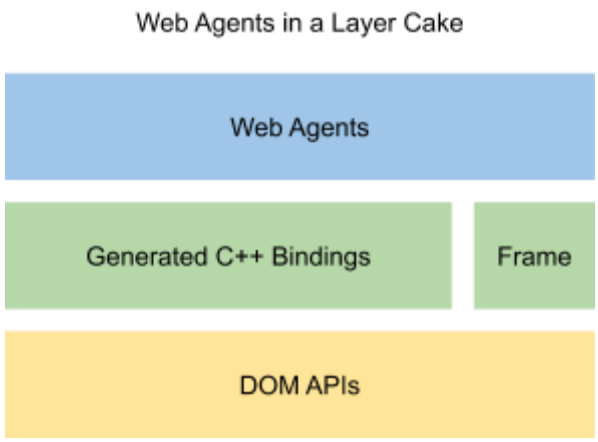
The term *Web Agents* is used interchangeably to represent both the pattern and the actual features implemented using the Web Agent pattern: Translate Web Agent, Autofill Web Agent, etc.

The development of this pattern is motivated by the following **goals**:

- **Improve schedulability and interruptibility**. The Web Agent pattern is designed to give the rendering engine highly granular control over when (and whether) the Web Agents code gets to run. In other words, the Web Agents pattern makes high-level browser features schedulable (and eventually interruptible), which improves the engine responsiveness.
- **Improve layering**. The Web Agents pattern enables better layering, cleanly separating Web Platform features (those features that are required for a Web Platform implementation) from higher-level browser features. Similarly, replacing hand-built wrappers for DOM interfaces in public/web and Source/web with generated C++ DOM bindings provides a clear separation of concerns and reduces the risk of adding logic in the bindings layer.
- **Drive Web Platform improvements**. Using generated C++ DOM bindings encourages more faithful use of the Web Platform APIs and fosters better understanding of edge cases and deficiencies in Web Platform. At the limit, Web Agents are just Web apps with extra capabilities.

The **Web Agents project** aims to:

- Design the Web Agents pattern and implement required infrastructure;
- Apply the pattern consistently by converting all existing code that fits the pattern to Web Agents.



Web Agents can be viewed as four interdependent parts:

1. **Web Agents API**. A set of generated from IDL thin wrappers around DOM classes. Web Agents will be using these wrapper classes to access DOM APIs, similar to how Javascript uses JS bindings to access DOM APIs.

2. **The Frame class**, which will be the access point to the DOM wrapper classes (for example, it will have an accessor to the Document wrapper class instance) Frame will be passed to a Web Agent upon instantiation. It will act as an EventTarget and provide a set of events to listen to. These events will roughly represent the use cases that motivated all virtual dispatch callbacks in RenderFrameObservers. Additionally, the Frame class will provide facilities for registering and requesting mojo interfaces.

3. **The infrastructure for dispatching events**. The main opportunity for better schedulability and interruptibility lies in making smarter decisions around dispatching events that the Web Agents listen to.

4. **The actual Web Agent base class**, which builds on the RenderFrameObserver pattern. This work includes design patterns and guidance for making robust Web Agents (for example, setting an expectation that events may never be called because this particular Web Agent is paused).

The Web Agents introspect or mutate the state of the associated frame (and its subtree) in response to events inside of the frame or requests from embedder, and communicate the outcomes of these actions back to the embedder:

- To introspect or mutate the state of the frame and its document, Web Agents rely in on Web Agent API.

- To react to events inside of the frame, Web Agents register event listeners using Web Agent API and Frame.
- To let embedder make requests, Web Agents register mojo interfaces with the associated frame (most commonly, they implement the mojo interface)
- To talk back to the embedder, Web Agents obtain and use mojo interfaces from the associated frame.

Most of the code that uses Web* APIs today will eventually transition to become Web Agents.

Details/Specifics

An outline of the modules design can be seen in https://codereview.chromium.org/2591803002. Here are some highlights.

Web Agents use the event listener pattern that is typical for Web platform APIs. The constructor is passed a Frame:

```
namespace web {

class SampleAgent : public Agent,
                    public mojom::SampleAgent {
 public:
    SampleAgent(Frame*);
  // ...
};

}  // namespace web
```

In constructor, the agent is able to register events, Mojo interfaces, and set up:

```
namespace web {

SampleAgent::SampleAgent(Frame* frame) : Agent(frame) {
  frame->AddEventListener(FrameEvents::DidCommitProvisionalLoad,
      base::Bind(&SampleAgent::DidCommitProvisionalLoad,
                 base::Unretained(this));
  frame->AddEventListener(FrameEvents::DidAssociateFormControl,
      base::Bind(&SampleAgent::DidAssociateFormControl,
                 base::Unretained(this)), PASSIVE);
  frame->AddEventListener(FrameEvents::FocusedNodeChanged,
      base::Bind(&SampleAgent::FocusedNodeChanged,
```

```
                base::Unretained(this)), CLEAN_LAYOUT);
  // ...
  frame->GetInterfaceRegistry()->AddInterface(
      base::Bind(&SampleAgent::BindRequest, base::Unretained(this)));
  // ...
}

}
```

The generated Web Agents API code has to balance faithful representation of Web IDL semantics and idiomatic C++:

```
namespace web {

class Document : public Node {
 public:
  virtual ~Document() = default;

  // ...
  //  [CallWith=EnteredWindow, CEReactions, CustomElementCallbacks,
  //      RaisesException]
  //    void write(DOMString... text);
  template<typename... Args>
  void Write(Args... args) {
    String array[] = { args... };
    Vector<String> vector(sizeof...(args));
    for (auto item : array)
      vector.push_back(item);
    document()->write(document()->domWindow(),
                      vector, ASSERT_NO_EXCEPTION);
  }

  // typedef
  //   (HTMLScriptElement or SVGScriptElement) HTMLOrSVGScriptElement;
  // readonly attribute HTMLOrSVGScriptElement? currentScript;
  using CurrentScript =
    WTF::Variant<blink::Member<HTMLScriptElement>,
                 blink::Member<SVGScriptElement>>;
  Optional<CurrentScript> GetCurrentScript() const;

  // ...
};

}
```

Implementation Stages

Here's a list of possible implementation stages of this project:

- Switch to Event Dispatch Model
    - Blocked on:
        - ?
    - Comments/questions:
        - Can be done incrementally
- Web Agents API Code Generator
    - Blocked on:
        - ?
- Port RenderFrameObservers to Web Agents
    - Blocked on:
        - ?
- Remove Web* API
    - Blocked on:
        - Web Agents API Code Generator
-

Open Questions

- How do we decide what is the priority between agents? Will need to look at current cases
- Agents commonly collaborate (translate and phishing detector, autofill and password manager), so we need a way for them to talk to each other. Probably via custom events?