Build Instant Apps and Bots with Messenger Web Views

March 14, 2017

by Luke Birdeau

There are many opinions about what distinguishes bots from apps. The most common is that apps are visual and bots are conversational. It's a nice shorthand, but it's important to not get caught up in strict dichotomies. Bots can be visual and apps can likewise be conversational. It's more about the **primary perspective** for each. And with bots the primary perspective is the message stream.

For those times when a bot requires custom visual interaction, Messenger provides a Web View (an embedded browser with HTML5 support). This component gives bot developers the visual control they need to meet nearly any application use case. This is critically important for bot developers as customers are savvy and will not tolerate even average usability. Do you really want to book a flight using an endless series of message bubbles?

Approximately half of deployed mobile apps are hybrid apps constructed using the same HTML5 container that underlies the Messenger Web View. This means that bot developers have at their disposal the same building block used to build apps like Uber and Evernote. With the tools provided by Messenger, you really can deliver capable visual solutions.

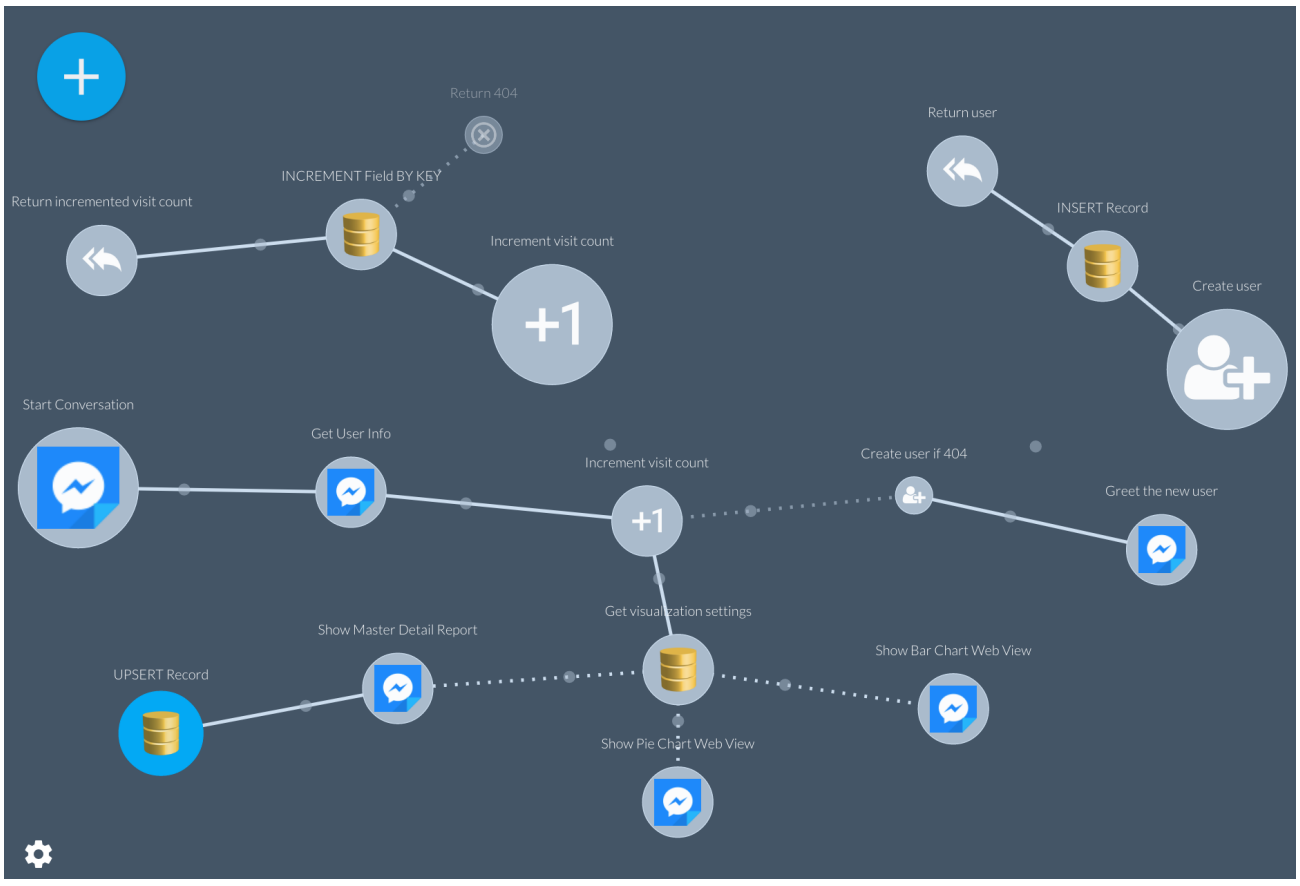
It requires organized effort to leverage the Messenger Web View when compared to other UI components. It's essentially a blank canvas and requires work to corral its features and make it behave with proper inputs, outputs, and faults. The remainder of this document will detail how we chose to augment the Web View at Intwixt to better leverage its rich abilities and use it interchangeably with the other UI controls provided by Messenger. These strategies include:

1. Normalize the Web View development experience
2. Isolate message stream semantics
3. Take a data-first approach
4. Standardize common UI patterns
5. Allow for full customization

1. Normalize The Web View Development Experience

By way of background, our bot development platform is process based, so we encourage developers to orchestrate their Messenger bot by connecting a series of activities into a process diagram. Our runtime engine then executes each step to control the Messenger bot.

Our process flows read like a storyboard and detail the flow and purpose of the bot. For example, according to the following diagram, the bot first validates if it recognizes a user and then increments their visit count. If the user is new, the bot creates a new user record and then sends a welcome message. If the user is returning, the bot will query their settings to determine which report style to display. The bot will then load one of three different Web Views: Pie Chart, Bar Chart, or Master/Detail Report. The Master/Detail allows the user to modify a record, and is therefore followed by an upsert activity in the database. We believe that treating the Web View as just another component type and keeping it interchangeable is key to delivering a rich and unified experience. Whether the bot is sending a simple message or displaying a Web View it's the same development experience.

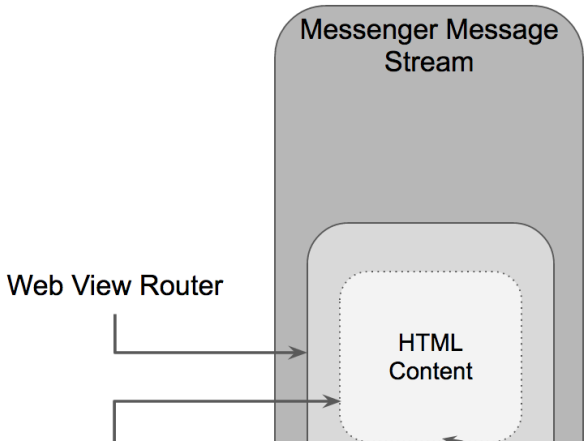


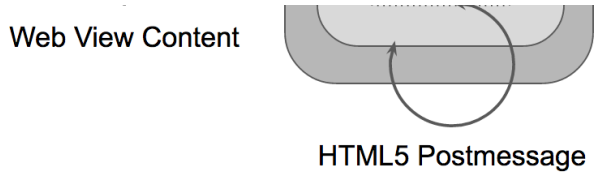
Bot Process Model

2. Isolate Message Stream Semantics

The central organizing principle for Messenger bots has traditionally been the message stream (instant apps will influence and evolve this over time). Users typically don't navigate deeply nested menus or follow successive links to reach content as is the case with traditional apps. Instead, they navigate by conversing with the bot, eventually interacting with a Web View, because it was appropriate to the use case and was injected into the message stream.

Because of the importance of the message stream, we made an early design decision to isolate all data exchange and Web View lifecycle events. Each time we render a Web View we actually use two HTML pages, nesting one inside the other. The outer page (*Web View Router*) knows how to acquire the data via Ajax from the message stream context, how to load the inner page, and how to close and return data to the message stream when the Web View session is complete. The inner page knows nothing of the message stream semantics and is a vanilla HTML page. It communicates cross-frame with the outer page using HTML5 postMessage. This pattern is also useful when repurposing existing HTML assets and using them within a Messenger bot. I will cover this more in section 5.





Using this approach, the HTML content developer focuses on styling, layout, and behaviors. This serves to greatly simplify the development and maintenance of the Web View micro UIs. All data exchange and lifecycle complexity is replaced with a factory call to **newRouterClient** to get the initial data set from the router and a call to **emitData** to close the session and return the session output to the message stream when done. Here is the boilerplate HTML for a valid *Web View Content* page.

```
<!DOCTYPE html>
<html>
<head>
<script src="/static1/webviews/js/router_client.js">
</script>
<body>

  <script>
  var myRouter =
Intwixt.newRouterClient(function(data) {
    //use your favorite MVC framework to render the
JSON data
  });
  </script>

  <button
onclick="myRouter.emitData({cancel:true})">Close Web
View</button>
</body>
```

I've included a **Close Web View** button in the above example to show how to end the Web View session. If the Web View is configured to return data, the process works the same, calling **emitData** to route the output to the

Web View Router container. The router will then manage the boilerplate interactions like making Ajax calls to deliver the response data to the message stream and closing the Web View.

3. Take A Data-First Approach

When a Web View loads, its initial content is strongly influenced by the upstream message context. And when dismissed, the Web View output likewise affects downstream context. The Web View both influences and is influenced by the information stream in which it lives. Critically, this means that the Web View Micro UIs that get developed behave very much like functions with defined inputs, outputs and errors.

When our team decided to augment the Messenger Web View and corral it more formally, we determined that we would need to include a formal schema specification to define the contract for our Web Views. We needed a robust mechanism for defining both data and metadata (such as validation constraints, and whether or not a given field of data was required). We decided upon JSON as the encoding format for our data and then developed a custom in-house schema format similar to JSON Schema. When a developer chooses to use a Web View, they define input, output, and fault messages using the schema to ensure that the information flows predictably from the message stream to the Web View and back to the message stream.

Not all data exchange must be brokered by the router in the outer HTML container.

The router exists to provide a bridge to the message stream context, but there are situations where the embedded Web View must make Ajax calls or use a WebSocket. This is OK and should be encouraged for a better user experience. For example, if the Web View is used for a flight reservation bot, it could make REST calls in the background using Ajax as the user changes their arrival and departure times, updating flights and prices to match.

If after searching flights the user does choose a specific flight, their selection would become the output for the Web View session and would

be sent to the containing Web View Router for use by downstream messaging activities (perhaps to pay for the chosen flight).

4. Standardize Common UI Patterns

I've built UI frameworks on desktop, mobile and Web. There are many principles that apply universally and are good rules of thumb when approaching UI development. My favorite is:

> "Every interface is either a list of items or a single item."

The user is either viewing and picking items from a homogeneous list or they are targeting an item in the list. It's incredibly simple, but it serves to remind that most interfaces fall into one of only a few core design patterns. And it forms the basis for our core Web View templates that we provide out of the box, namely:

- **Radiogroup**: Choose a single item from a list
- **Checkgroup**: Choose one or more items from a list
- **Form**: Schema-generated HTML form and elements

The following sections will detail these various template types and how they are used to broadly support the most common interaction patterns.

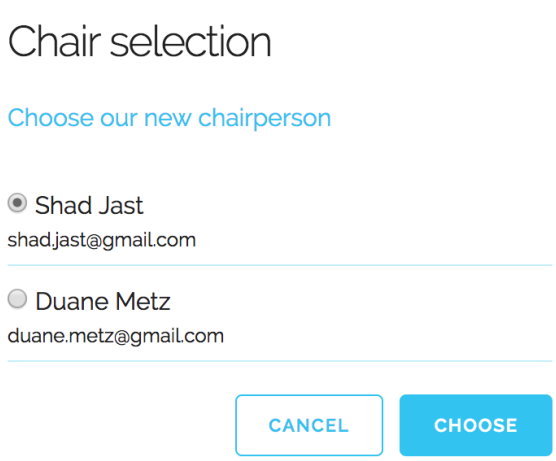
Radiogroup and Checkgroup Templates

The *Radiogroup Template* is used whenever the end user user must choose a single item from a list while the *Checkgroup Template* is used to choose one or more items. These Web Views are configured using JSON to describe the configuration (such as the page title and submit button label) and the data.

```
{
```

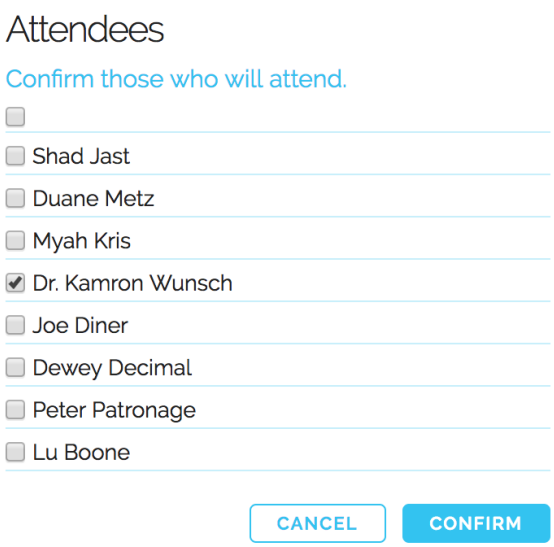
```
  "data": {
    "sjast": {
      "title": "Shad Jast",
      "subtitle": "shad.jast@gmail.com",
      "id": "sjast"
    },
      "dmetz": {
        "title": "Duane Metz",
        "subtitle": "duane.metz@gmail.com",
        "id": "dmetz"
      }
  },
  "config": {
    "title_label": "Chair Selection",
    "instructions_label": "Choose our new
chairperson",
    "cancel_label": "Cancel",
    "submit_label": "Choose"
  }
}
```

When rendered in the Web View, the checkgroup (or radiogroup as is shown below) renders each JSON data item as an item in the list.



Of course, a simple picklist of options isn't a very compelling

augmentation. Messenger already comes with its own picklist controls, including quick replies and carousels. But things get interesting once you start to apply custom stylesheets. Consider the following examples. The first uses a “compact” stylesheet while the second shows even greater customization with the radio options rendered as color swatches--all done entirely with CSS.



We strongly encourage developers to look for clever ways to reuse the basic template patterns (radiogroup, checkgroup, and form) by using only custom CSS. Stylesheets are a great way to deliver custom micro UIs, and they can be served cross-domain from any CDN. The latest semantics for CSS behaviors, attribute selectors, and animations are rich and performant.

Form Template

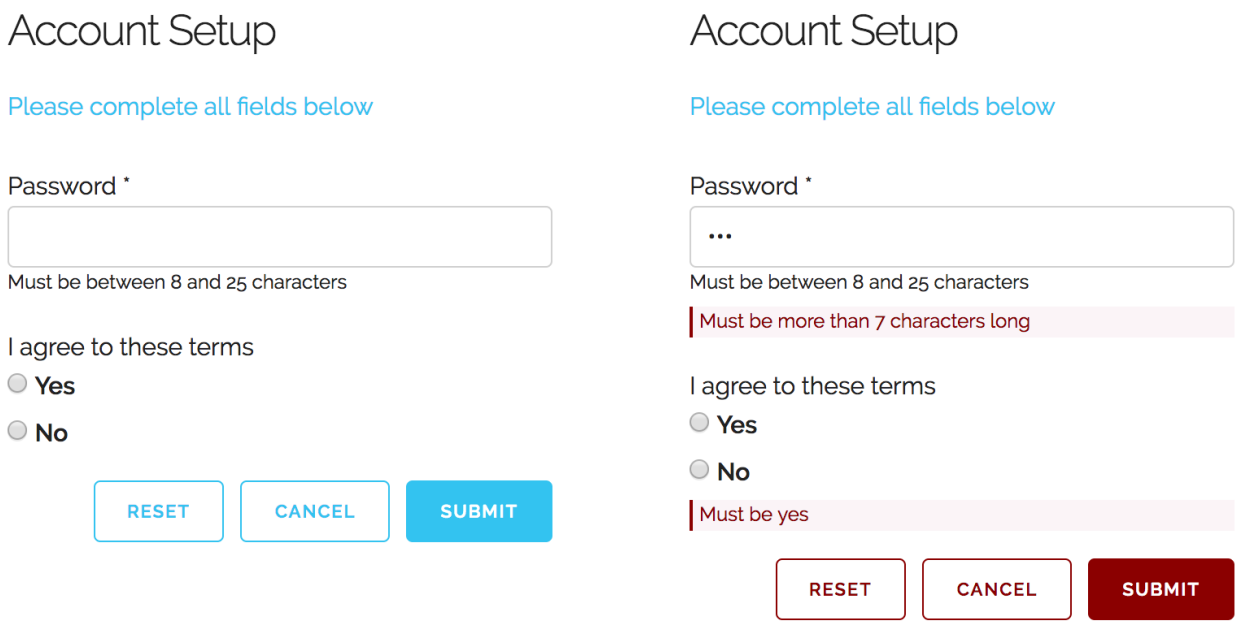
The *Form Template* Web View is declared using JSON, similar to the *Checkgroup Template* and *Radiogroup Template*. However, in the case of the Form Template, each form field is declared using a variant of JSON schema. This allows for form fields with highly customized behaviors and properties, including complex state validation.

By way of example, the following JSON will generate a form with password and radio fields. The password is required and must be between 8 and 25

characters long. The radio button selection must be “yes”. Other configuration options are shown including the button labels and page titles. And critically, a custom CSS file can be included when the Web View is configured to fully customize the user experience.

```
{
  "model": [
    {
      "id": "password",
      "title": "Password",
      "tip": "Must be between 8 and 25 characters",
      "data_type": "string",
      "field_type": "password",
      "required": true,
      "constraints": { "$gtlen": 7, "$ltlen": 26 }
    },
    {
      "id": "agreement",
      "title": "I agree to these terms",
      "field_type": "radio",
      "data_type": "string",
      "data": [
        { "id": "yes", "title": "Yes" },
        { "id": "no", "title": "No" }
      ],
      "constraints": { "$eq": "yes" }
    }
  ],
  "config": {
    "title_label": "Account Setup",
    "instructions_label": "Please complete all fields
below",
    "submit_label": "Submit",
    "reset_label": "Reset",
    "cancel_label": "Cancel"
  }
}
```

Here is the Web View rendered in both its initial and error states. Note how error messages and style changes are automatically injected. And the submission is halted if the form fails validation.



5. Allow For Full Customization

Standard templates are advantageous, because they're configured solely through JSON and CSS. However, it is critical that developers be able to construct Web Views of arbitrary complexity. Because we isolate the message stream semantics using the nested page strategy, we can embed any HTML content as the inner page as long as that content includes our Web View Router library.

```
https://con2.intwixt.com:3017/static1/webviews/js/router_client.js
```

The page must then call the factory method, **newRouterClient**. If there is

any data provided by the message stream it will arrive via this channel.

```
// instance the router to get the initial data
var myRouter = Intwixt.newRouterClient(function(data)
{
  //render the data with your favorite UI toolkit
});
```

Finally, the custom HTML page can complete the Web View session by calling **emitData** on the router instance. This will cause the Web View output to be sent to the message stream (if there is output data) and the Web view will be dismissed.

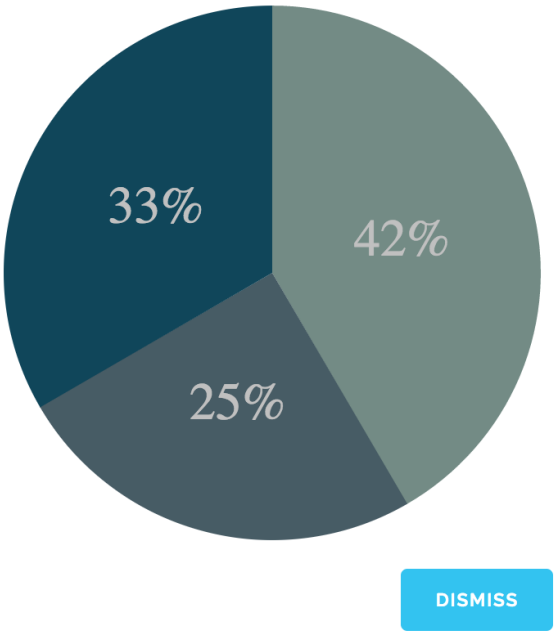
```
//end the session (send output to the router)
function closeSession(data) {
  myRouter.emitData(data);
}

//cancel the session (don't send output to the router)
function cancelSession() {
  myRouter.emitData({cancel:true})
}
```

Using the nested HTML page approach to the Web View significantly reduces the interface friction between a standard HTML page and the Messenger Message stream. Importantly, this means that if there are existing assets (perhaps authored as part of a hybrid Web app), it is possible to repurpose the content and include as a custom Web View. The one catch is that the custom content should load quickly to ensure a consistent user experience. The more your Web View behaves like the underlying message stream and native Messenger controls (including performance), the more consonant your overall user experience will feel.

We recommend that your custom Web Views be of limited complexity. Use a CDN to deliver static content and in general use strong HTTP cache settings for static assets like CSS and JavaScript. The total transfer size should be less than 100KB for the HTML page and its referenced assets. This ensures excellent load times and consistent usability for your custom Web View.

Consider the following custom Web View that renders a pie chart. It renders in real time, using a lightweight, single-page design requiring less than 25KB for all libraries and content.



Here is the underlying HTML for the pie chart. Notice how sample JSON data is provided to the router client when the factory method is called. This aids in development, allowing the Web View author to test and iterate their custom HTML in isolation of the Messenger event stream. The Intwixt router library knows when the HTML page is running within Messenger and will initialize the page with the sample data set when the page is run stand-alone. This reduces the complexity of development dependencies as the HTML page author is only concerned with behaviors, layouts, and styles. The JSON document becomes the contract between those who manage the message stream and those who author the rich UIs.

```
<!DOCTYPE html>
<html>
 <head>
 <title>Pie Chart Web View</title>
 <meta name="viewport"
       content="user-scalable=no, initial-scale=1,
maximum-scale=1,
       minimum-scale=1, width=device-width,
height=device-height"/>
 <link rel="stylesheet" href="/chartist.min.css">
 <script src="/chartist.min.js"></script>
 <script src="/static1/webviews/js/router_client.js">
</script>
</head>

<body class="container">
 <div class="ct-chart"></div>
 <script>

   var sample_data = { data: { series: [5, 3, 4] } };

   function on_data(data) {
     new Chartist.Pie('.ct-chart', data.data);
   }

   var myRouter = Intwixt.newRouterClient(on_data,
sample_data);

 </script>
 <button
onclick="myRouter.emitData({cancel:true})">Dismiss</bu
tton>
</body>
</html>
```

Conclusion

Web Views are a powerful addition to the Messenger control set. Their flexibility enables developers to build visual, instant apps as easily as more traditional conversational chatbots. It's a powerful component just waiting to be extended.

Posted in how to *and*
tagged with bot, chatbot, web view, messenger

Newer / Older

TERMS OF SERVICE PRIVACY POLICY VIDEOS

email info@intwixt.com
phone 415.570.8502

Disclaimer: Intwixt is a general integration platform, capable of connecting third party services through standard Web APIs. Logos, titles, and descriptions do not imply affiliation or partnership between Intwixt and the respective owner unless otherwise specified.