Symphony Client Side Architecture

In this document we will review the client side architecture of Symphony. In short, the client is a scalable, framework agnostic, modular architecture with its goal being to play nice with the abundance of javascript libraries available today. Below, I will review the four tenants of the client: the core, the extensions, the sandbox, and lastly the modules.

The core of the application is simple. It initializes extensions (common services used through the application), the sandbox (an event bus), and handles module life cycle (discreet units of user interaction). The lifecycle of a module includes registration, creating new instances, and destroying existing instances. This is the limit of the core's responsibilities.

Extensions are services provided for use throughout the application. A module uses an extension to supplement the UI. Extensions are initiated by the core when the application starts up. Let us review some of the most important extensions. First we have the `longpoll` which is responsible for handling incoming messages from the server. Another extension is the `dataStore` which stores arbitrary client data (either locally or to the server). The transport extension communicates with the server which uses jQuery AJAX requests behind the scenes. Note you only need to use the transport extension if you are interacting with Symphony APIs. Interaction with other 3rd services can be achieved with a plain AJAX requests. Other services such as user presence, user aliasing, logging, version checking, and layout are all achieved with extensions. Modules interact with extensions over the sandbox.

The sandbox provides a communication interface throughout the application. The sandbox sits between the modules and the core. If an extension is made available to the sandbox then it is automatically available to all modules. Two sandbox methods are incredibly useful: publish and

subscribe. The pubsub methods allow communication amongst modules, as well as from modules to the core (and therefore any extension). Publishing an event is as simple as

```
this.sandbox.publish('view:created', null, {…data…})
```

Now any module or any extension can listen to this event:

```
this.sandbox.subscribe('view:created', function(context, data){…})
```

This is how communication amongst any entity within the app is handled. Remember to unsubscribe from the event when your object is destroyed!

Now let us observe the lifecycle of an incoming message to demonstrate how the sandbox works. The `longpoll` extension has an active connection to the server. Now assume the server returns a message and the `longpoll` beings parsing. It finds the data to be of type "instant message," and calls the instant message handler which simply publishes a message:

```
this.sandbox.publish('incoming:message', null, message);
```

A few things listen to this particular type of event including the navigation, the `appBridge`, and the IM module. The left navigation uses this information to increment the unread badge. The IM module uses it to render a new message in the message list. The `appBridge` uses the message to render a desktop alert.

Modules are discreet units of user interaction. Examples include the header, the navigation, the IM widget, the search widget, or any application from 3rd party developers. Modules are destroyed when the module is removed from the grid/layout. There are a few rules to follow when writing a module: Modules only have knowledge of itself and sandbox. Therefore all the DOM interaction can only happen within the module itself. The ability to communicate with other modules (or to the core) is provided through the sandbox.

Persisting data to the server is straight forward. While the `dataStore` is essentially a freeform key/value store, using a particular key prefix will automatically save to the server. Any `dataStore` key that begins with “documents.” is automatically saved. Here is an example:

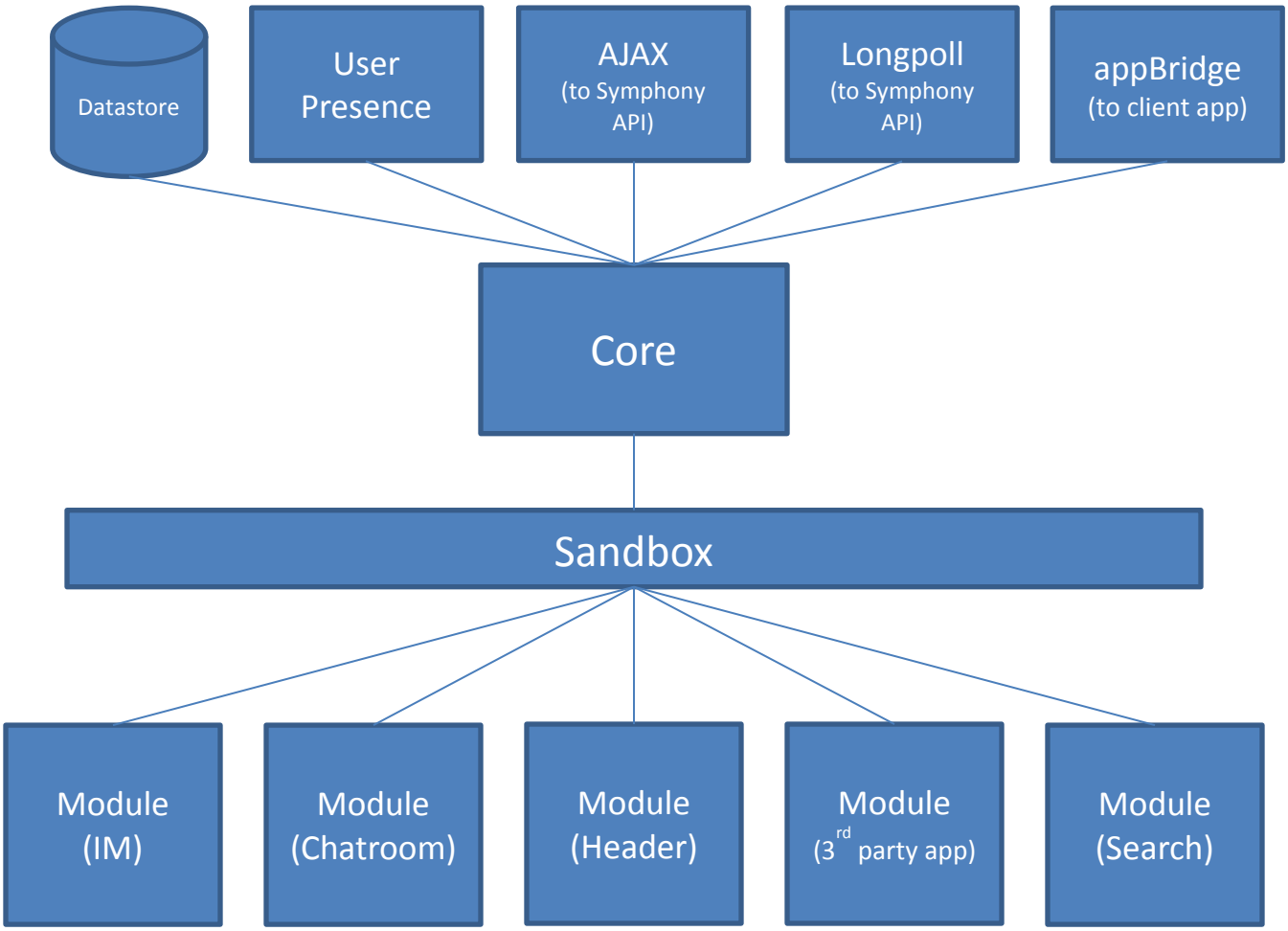
```
this.sandbox.setData('documents.apps, activeApps)
```

In this case the user’s `activeApps` object is being saved to the key “`documents.apps`”. To retrieve the data use the `getData` method from the sandbox. An example would be something like:

```
this.sandbox.getData('documents.apps);
```

Using the `setData` and `getData` methods, paired with the `documents.` prefix makes it easy to persist data to the server.

In a more general sense, the project makes use of Browserify, which allows for node style require statements to import modules. The ultimate test for the overall design is to run any random module on its own and it should function as expected. It should be noted the design was intended to be framework agnostic. You will find no (or very few) mentions of any javascript frameworks in the sandbox and above. That said, nearly all of the modules use Backbone JS. However adding support for other frameworks should be straightforward.


Datastore
User Presence
AJAX (to Symphony API)
Longpoll (to Symphony API)
appBridge (to client app)
Core
Sandbox
Module (IM)
Module (Chatroom)
Module (Header)
Module (3rd party app)
Module (Search)