

Arbitrary Extensions to Apache Shindig's Data Model

Introduction

By definition, the [OpenSocial spec supports extensions](#) to its data model. In Apache Shindig, implementing data model extensions requires familiarity with Java and Shindig's back-end and requires continued maintenance as extensions evolve. This article documents how to enable arbitrary extensions in Apache Shindig's data model, circumventing the need for explicitly hardcoding and maintaining extensions.

For example, the following is an Activity from the Activity Streams API with arbitrary extensions:

```
{
  id: "activity123",
  ...
  object: {
    ...
    replies: [{...}, {...}, {...}]    // This field isn't hardcoded in Shindig, but it will handle it
  },
  extensions: {
    dynamicExtension: "Another extension that was not predefined"
  }
}
```

How to Enable Arbitrary Extensions

Currently, Activity Streams support arbitrary extensions in Apache Shindig. Additional data models may support arbitrary extensions with a very quick and simple change: the data model's interface must extend `ExtendableBean`, and the interface's implementing class must extend `ExtendableBeanImpl`.

For example, the following shows how to enable arbitrary extensions with Activity Streams.

ActivityEntry.java interface definition:

```
@ImplementedBy(ActivityEntryImpl.class)
@ExportableBean
public interface ActivityEntry extends Comparable<ActivityEntry>, ExtendableBean { . . . }
```

ActivityEntryImpl.java class definition:

```
public class ActivityEntryImpl extends ExtendableBeanImpl implements ActivityEntry { . . . }
```

XML Limitations

I recommend using the JSON API when handling extensions. The JSON API fully supports all CRUD operations with extensions "out-of-the-box" by simply extending `ExtendableBean`. XML serialization requires a couple of extra steps and has a number of limitations due to Shindig's design and use of `xstream`. Specifically:

- To serialize extensions to XML, the extension's field name must be predefined in the parent POJO. For example, to properly serialize the 'extensions' bucket in the Introduction example, the 'extension' property must be hardcoded in `ActivityEntry.java` (and the implementing class). Fortunately, once the property is added, nested extensions within the bucket are supported to an arbitrary depth.
- Very limited XML POST support. This has been a longstanding limitation of Shindig. Only the most basic XML POST requests succeed (only string key/value pairs, no Lists or Maps).
- Serialization of extensions do not follow all OpenSocial conventions. The following illustrates the serialization of a JSON extension which includes simple string key/value pairs, a list of strings, and a list of objects:

```

--- JSON SERIALIZATION ---
extension: {
  key1: "value1",
  key2: "value2",
  stringList: [
    "element1",
    "element2",
    "element3"
  ],
  mapList: [
    {
      key1: "value1"
    },
    {
      key2: "value2"
    }
  ]
}

--- XML SERIALIZATION ---
<extensions>
  <key1>value1</key1>
  <key2>value2</key2>
  <stringList>
    <java.lang.String>element1</java.lang.String>
    <java.lang.String>element2</java.lang.String>
    <java.lang.String>element3</java.lang.String>
  </stringList>
  <mapList>
    <map>
      <entry>
        <key>key1</key>
        <value>value1</value>
      </entry>
    </map>
    <map>
      <entry>
        <key>key2</key>
        <value>value2</value>
      </entry>
    </map>
  </mapList>
</extensions>

```

How It's Implemented

Here are some high-level implementation details of how arbitrary extensions are supported in Shindig. It should be a helpful starting point for modifying or expanding this work.

First, `ExtendableBean` and `ExtendableBeanImpl` are nothing more than wrappers for `Map` and `HashMap`, respectively. Thus, when a data object extends from `ExtendableBean`, it itself is a `Map`.

Next, when a JSON POST is received, the `BeanJsonConverter` is called to convert the raw JSON string to the appropriate target object type. A simple check was added within `BeanJsonConverter` to determine if the target class type extends `ExtendableBean`. If it does, the `BeanJsonConverter` will treat the object as a `Map` to initialize the `Map`'s key/value pairs to represent the JSON object in addition to initializing the object normally (using its getters and setters).

When the data object is persisted on disk, the object is stored as a `Map` to retain all properties of the initial POST request, and vice versa.

To support XML serialization for GET requests, I wrote a custom converter and registered it with `xstream`. The converter is named `ExtendableBeanConverter`, and it is registered with `xstream` within `XStream081Configuration.java`.

For more detail, check out the relevant source files for more comments: `ExtendableBean.java`, `ExtendableBeanImpl.java`, `BeanJsonConverter.java`, `ExtendableBeanConverter.java`