Building Collaborative Apps with Wookie

Collaborative Apps

- Use W3C Widgets packaging and widget object API with the Google Wave Gadgets API
- Runs in Wookie, no Wave server needed
- Requires plugins that can provide participant information using the Wookie REST API

Collaborative Widgets: APIs

- *State*
- *Participants*

State

- State is shared across widgets with the same IRI in the same shared context.
- State is propagated to related widgets using an event callback
- State is set by submitting deltas (as associative arrays) or single values

State example

```
wave.setStateCallback(stateUpdated);

stateUpdated = function(){
 var keys = wave.getState().getKeys();
 for (var i = 0; i < keys.length; i++) {
    alert(wave.getState().get(keys[i]));
 }
};

wave.getState().submitValue(“key”, “value”);
```

State change events

- State changes are automatically pushed to your widget instance, you don’t need to poll or check manually
  - So keep your state model as small as you can to reduce latency
- *Usability tip*: Consider whether to auto-refresh, or to prompt the user to refresh using a message. It can be confusing for users if the layout or order of items changes without them doing anything.

Participants

- Information about users accessing the widget
- Set by the plugin by calling Wookie's participants REST API
- Viewer is the current user object; participants is the set of users
- Good to have fallback as widget may be used in a "guest access" environment with no viewer (e.g. see Natter widget)

Participants

- Register callbacks with:
  wave.setParticipantCallback(myfunction);

- Methods*:
  - getViewer() - get the current user
  - getParticipants() - get map of all participants

- Model:
  - getId(), getDisplayName(), getThumbnailUrl()

**in future releases the getHost() method will also be supported*

Making a collaborative app

- This requires some more *planning*

1. Draw up a design
2. Create a working test page
3. Implement models, action handlers and event handlers
4. Create the config.xml, icon.png, zip it up and run it

Design

- Start with the view - what the widget looks like
- Create the model - what are the objects in the state model?
- Add the actions - how you interact with it
- Wire it all up…

Model

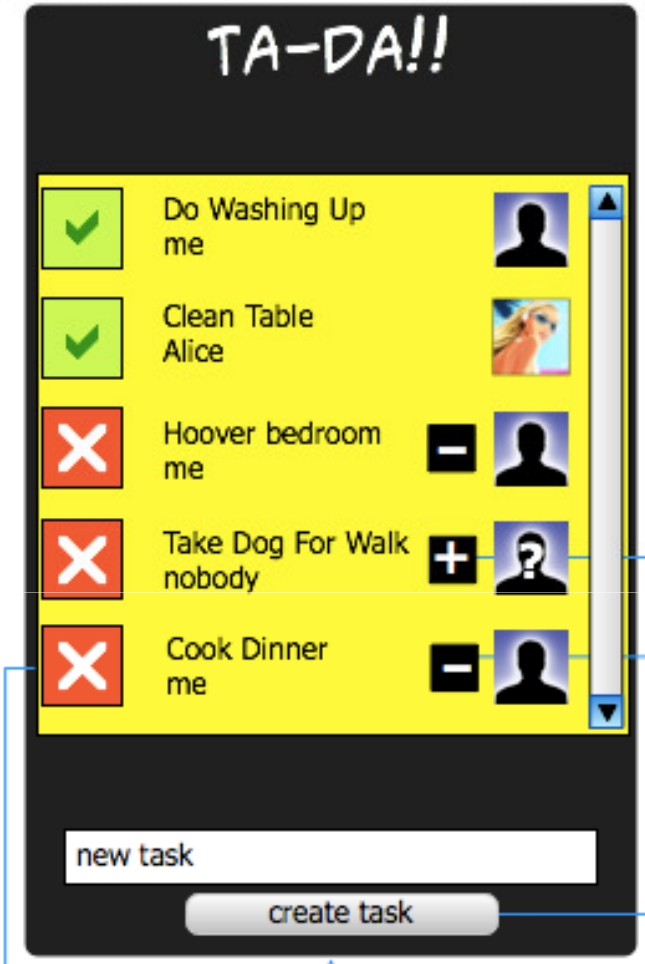
State

Task
task_id: String name: String status: String assigned_to: String
save() getUser(): participant

Participants

Preferences

View



Controller

Action Handlers

toggleTask()
newTask()
abandonTask()
claimTask()

Event Handlers

setStateCallback = stateUpdated()

setParticipantCallback = participantsUpdated()

Prototyping

- Make a regular html page to test out your view. Populate it with fake model, and don’t wire up the actions yet
- You can reuse this for the real widget - just take out your fake state info sections

Implementing

- Create a “Model” object for your state model
- Create a “Controller” object, and a method in it for each action
- Register the participant and state event handlers with an “update view” method that populates the view when running

Models

- Models can be implemented in typical “bean” fashion
- Save/Find methods need to access wave state
- Can use JSON (e.g. with json.js) or just plain strings for storage

```
function Task(id,name,status,assigned){
    this.task_id = id;
    this.name = name;
    this.status = status;
    this.assigned_to = assigned;
}
Task.prototype.save = function(){
wave.getState().submitValue(this.task_id, JSON.stringify(this));
}
```

Static model methods

```
Task.create = function(json){
        var obj = JSON.parse(json);
       var task = new Task(obj.task_id,
        obj.name,obj.status,obj.assigned_to);
      return task;
}

Task.find = function(task_id){
      var keys = wave.getState().getKeys();
      for (var i = 0; i < keys.length; i++) {
         var key = keys[i];
         if (key == task_id){
            return Task.create(task_id, wave.getState().get(key));
         }
      }
      return null;
}

Task.findAll = function(){
   var tasks = {};
      var keys = wave.getState().getKeys();
      for (var i = 0; i < keys.length; i++) {
      var key = keys[i];
      var task = Task.create(key, wave.getState().get(key));
      tasks[key] = task;
   }
   return tasks;
}
```

- Typically need methods to turn state strings back into model instances
- Also finder methods to get a particular model object
- This isn’t the only way to do this, but is an OK pattern

Controllers

```
/**
 * The Controller object
 * This is used to wire up the view and model with actions
 */
var Controller = {

    // Action handlers

    // Toggle task state between completed and to-do
    toggleTask: function(task_id){
    },

    // Create a new task
    newTask: function(){
    },

    // Abandon task for someone else to do
    abandonTask: function(task_id){
    },

    // Claim a task (assign to self)
    claimTask: function(task_id){
    }
}
```

- Methods for each action, making changes to the model
- You usually don't need to do any code that interacts with HTML, as the event handlers should do that

Event Handlers

```
// Update the view when state has been updated
  stateUpdated: function(){
     var tasks = Task.findAll();
     if (tasks && tasks != null){
        var tasklist = "";
        for (key in tasks) {
           var task = tasks[key];
           tasklist += // the task stuff to show
        dwr.util.setValue("tasklist", tasklist, {
     escapeHtml:false });
        var objDiv =
     document.getElementById("tasklist");
        objDiv.scrollTop = objDiv.scrollHeight;
     }
  },
  participantsUpdated: function(){
     Controller.updateUser();
  }
```

These fire whenever the state or participants are updated (e.g. by another instance).

Event handlers need to be registered like so:

```
wave.setStateCallback(Controller.stateUpdated);
wave.setParticipantCallback(Controller.participantsUpdated);
```

Also useful to have these methods called from onLoad() in an init() function to create the initial view

You can import JQuery if you like for setting the view content, or do it using DOM

Packaging

You need to add this to your config.xml to tell Wookie to include the Wave Gadget API methods:

```
<feature name="http://wave.google.com"
  required="true"/>
```

Fallback behaviours

Viewer handling with fallback

```
    /**
 * Setup user information
 */
user: {},
updateUser:function(){
if (wave.getViewer() != null){
    Controller.user.id = wave.getViewer().getId();
    Controller.user.username = wave.getViewer().getDisplayName();
    Controller.user.thumbnail  = wave.getViewer().getThumbnailUrl();
}
if (Controller.user.thumbnail == "" || Controller.user.thumbnail == null)
                Controller.user.thumbnail = "anon.png";
if (Controller.user.username == null || Controller.user.username == ""){
    Controller.user.username = "anonymouse";
    Controller.user.id = "anonymous";
}
}
```

Viewer handling with fallback: alternative approach

```
if (wave.getViewer() != null){
        username = wave.getViewer().getDisplayName();
        thumbnail  = wave.getViewer().getThumbnailUrl();
}
if (thumbnail == "" || thumbnail == null)
                thumbnail = "Images/default_thumbnail.png";
if (username == null || username == ""){
        username = "natterer" + rnd_no(9999);
}
```

Single-user mode

Could your app also work in a single-user environment with no Wave API?

e.g. test for existence of “wave” object, and adapt if its not present - e.g. store data in preferences not state

More faff, but makes your widget work in more situations (e.g. mobile)

```
<feature name="http://wave.google.com"
 required="false"/>
```

Spot the (semi) deliberate mistakes!

- There are a few problems with the ToDo widget - can you identify possible improvements?
  - Removing Wookie-specific code
  - Improved usability
  - Missing functionality

Uploading, debugging and testing

- You need a collaborative environment to test your widget properly
- Its useful though to test using Wookie's built in "demo" mode to check it still works OK in a "guest access" or anonymous user setup
- E.g. a Moodle or an Elgg installation

Other stuff…

- AJAX

If you want to call external sites from within the widget, call myurl = widget.proxify(url) first to call it via proxy. Also need to add the site to the server whitelist.

- Camera access

There is experimental support for BONDI camera capture API (will be checked in soon)