An MVP guide to JavaScript – Model-View-Presenter

In a previous post I described the benefits of MVP architecture (you can see this post). Now I will try to explain how to implement that design pattern, inspired from Google's GWT, with JavaScript. *I would assume, for this demo, that you are familiar with the basics of jQuery.*

What is MVP?

Model-View-Presenter is a design pattern which separates the code for a specific widget/functionality to three sections:

Model

In which the data model for the widget is defined.

View

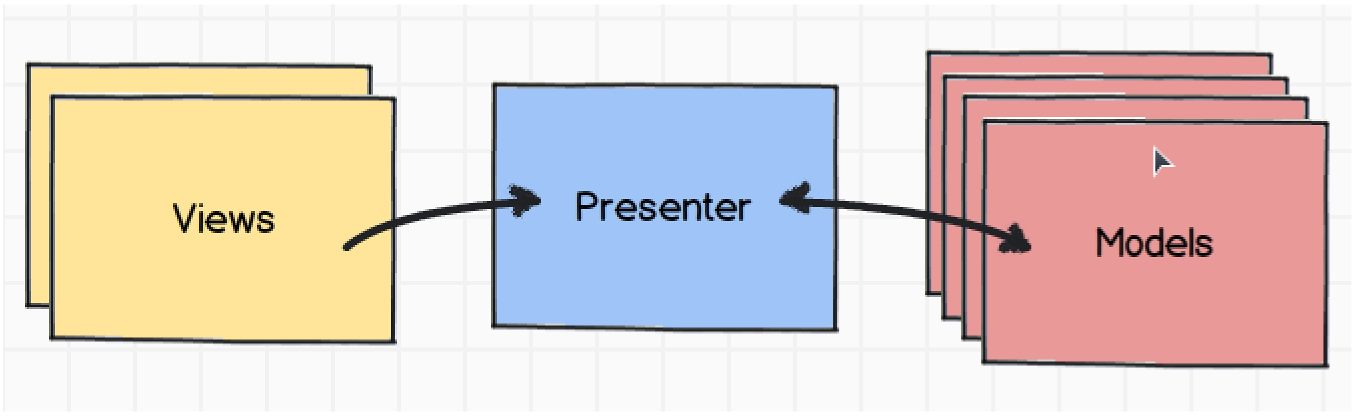
In which the logic behind the UI is handled, with UI events, data visualization and other UI centric logic.

Presenter

Where the logic behind the functionality of the widgets rests, such as data manipulation, data storing and loading, application events, etc…

So how does it work?

When working with MVP, the presenter is your main guy, this object carries the logic behind your widget and thus comprises the bulk of your code.



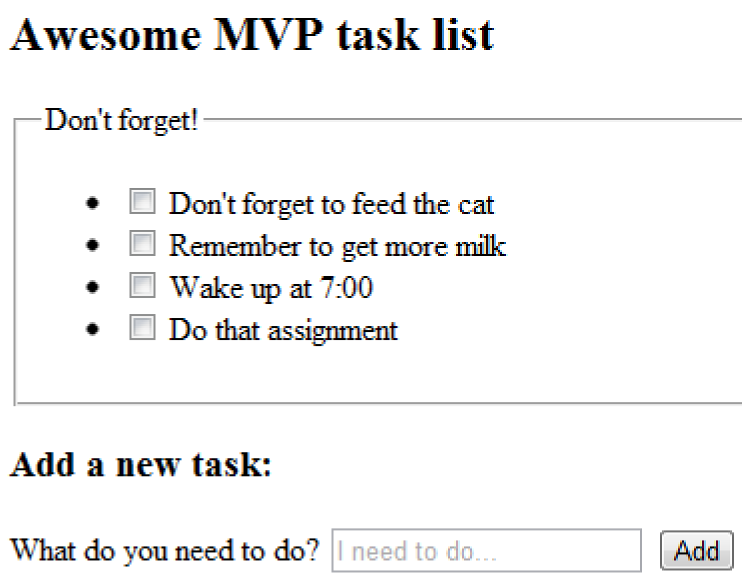
The presenter requires the corresponding view, and a model. When the presenter gets the model, it updates the view with different handlers, and the view will then update the UI.

The presenter will register handlers in the view for any UI events that require some logic.

A major benefit with MVP is that you can create **several views**, which may look and behave differently, but share the same presenter. Meaning that they share the same functionality, but the UI looks and behaves differently.

Show me how it's made!

First, for this demo, I will define the requirements, *the application:*


Awesome MVP task list

Don't forget!

- Don't forget to feed the cat
- Remember to get more milk
- Wake up at 7:00
- Do that assignment

Add a new task:

What do you need to do? I need to do... Add

A small task list application.

How can this be implemented in Model-View-Presenter?
I will need to split the defined app into two widgets; a Single Task Widget and a Task List Widget.

Creating a Task Widget

The Model

I'll start by defining a simple model for the task object:

models.js

```
function TaskModel(_text){
   var ID = (new Date()).getTime()
   this.getID = function(){
   return ID;
   }
   var Text = _text;
   this.getText = function(){
      return Text;
   }
   this.setText = function(value){
      Text = value;
   }
}
```

My preference is to make a getter and setter for each property in the model, it allows me to make changes to values when getting or setting them in a central location.

The Presenter

taskPresenter.js

```
function TaskPresenter(_view){

   var view;
   var model;

   function init(){
      view = _view;
      view.addCheckedHandler(function(){
         view.remove();
      });
   }

   var public = {
      getView: function(){
         return view;
      },
      setModel: function(_model){
         model = _model;
         view.setModel(model);
      }
   }

   init();
   return public;
}
```

First thing you will notice is that the presenter object gets its view (the UI) object in the constructor.
In a private function I can setup the object's logic, and the public functions define what the presenter is actually allowed to do.

As you can see, the `getView()` public function returns the view object which was supplied in the constructor, every presenter must expose this function.

Inside the `init()` function I registered a handler with the view that will fire when a task has been checked, this handler will tell the view to remove itself from the UI.

*As you can see the **presenter does not care** how 'checking the task' was achieved, but it knows what to do once that action happened.*

The View

taskView.js

```
function TaskView(){
     var html;
     function init(){
        html = $("<input type='checkbox'/><label></label></li>");
    }
     var public = {
        getHtml: function(){
            return html;
        },
        setModel: function(model){
            html.find("input").attr("id", model.getID());
            html.find("label").attr("for", model.getID());
            html.find("label").html(model.getText());
        },
        addCheckedHandler: function(handler){
            html.find("input").click(handler);
        },
        remove: function(){
            html.remove();
        }
    }
    init();
    return public;
}
```

The view contains an `html` property which is the HTML representation of this widget, a template.
All views must returns a public `getHtml()` function which will return the HTML object, the rendered template.

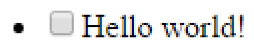
The public functions are those that I used in the presenter setup function, those functions represent the functionality of the view.

So I created a model, a presenter and a view for a single Task widget.

To use it, I need to do the following:

```
var model   = new TaskModel("Hello world!");
var task    = new TaskPresenter(new TaskView());
task.setModel(model);
$("body").append(task.getView().getHtml());
```

And this is the result:



Creating a list widget for the tasks

listPresenter.js

```
function ListPresenter(_view){
   var view;
   var model;
   function init(){
      view = _view;
      view.addCreateTaskHandler(function(taskTitle){
         var model   = new TaskModel(taskTitle);
         var task    = new TaskPresenter(new TaskView());
         task.setModel(model);

         view.addTask(task.getView());
      });

   }
   var public = {
      getView: function(){
         return view;
      }
   }
   init();
   return public;
}
```

The list presenter object registers a handler in the view when a new task is created, which then it will create a new task object and pass it’s view to the list view.

A presenter always handles with other presenter and a view always handles with other views.

listView.js

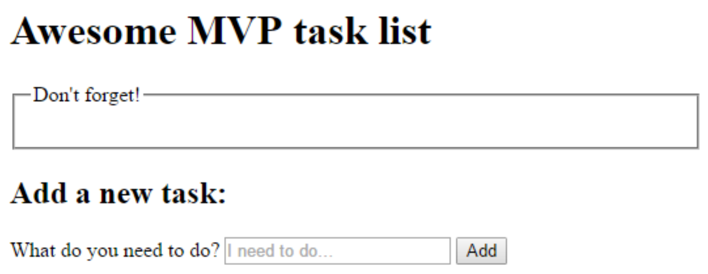
```
function ListView(){
   var html;
   function init(){
      html = $("<div>"+
            "<h1>Awesome MVP task list</h1>"+
               "<fieldset><legend>Don't
forget!</legend>"+
                  "<ul id='tasklist'></ul>"+
               "</fieldset>"+
            "<h2>Add a new task:</h2>"+
            "What do you need to do? <input
id='taskinput' placeholder='I need to do…'/>
<input id='submittask' type='submit'
value='Add'/>"+
                "</div>");
    }
    var public = {
        getHtml: function(){
            return html;
        },
        addCreateTaskHandler: function(handler){
            html.find("#submittask").click(function(){
                var newTaskTitle =
html.find("#taskinput").val();
                html.find("#taskinput").val("");
                handler(newTaskTitle);
            });
        },
        addTask: function(taskView){

html.find("#tasklist").append(taskView.getHtml());
        }
    }
    init();
    return public;
}
```

Since this is the main view for the application I included all the rest of the UI here, this isn't necessary, it's just for this demo's purpose.

The view's functionality is to create a task, and add the created task to the UI.
Now what's left is simply instantiate this object and add it to the HTML:

```
var list = new ListPresenter(new ListView());
$("body").append(list.getView().getHtml());
```

And the result is:



Download the example here:

MVP task list example