JS AND PATTERNS

or: How I Learned to Stop Worrying and Love JS Frameworks

@drpicox

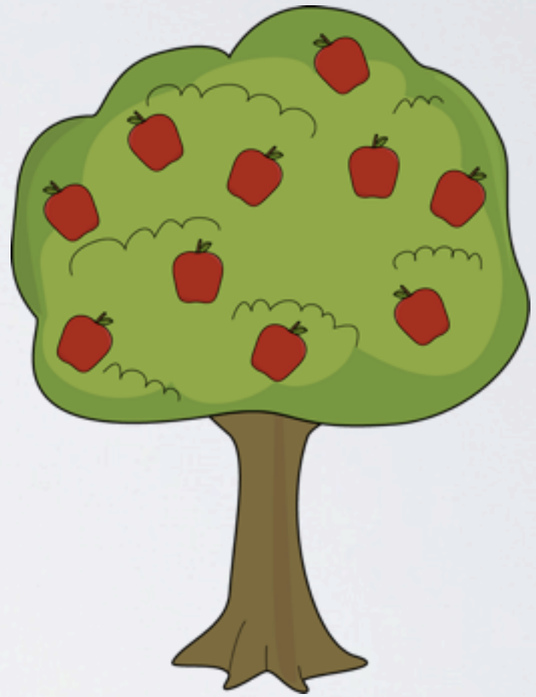
PATTERNS

Model View Controller

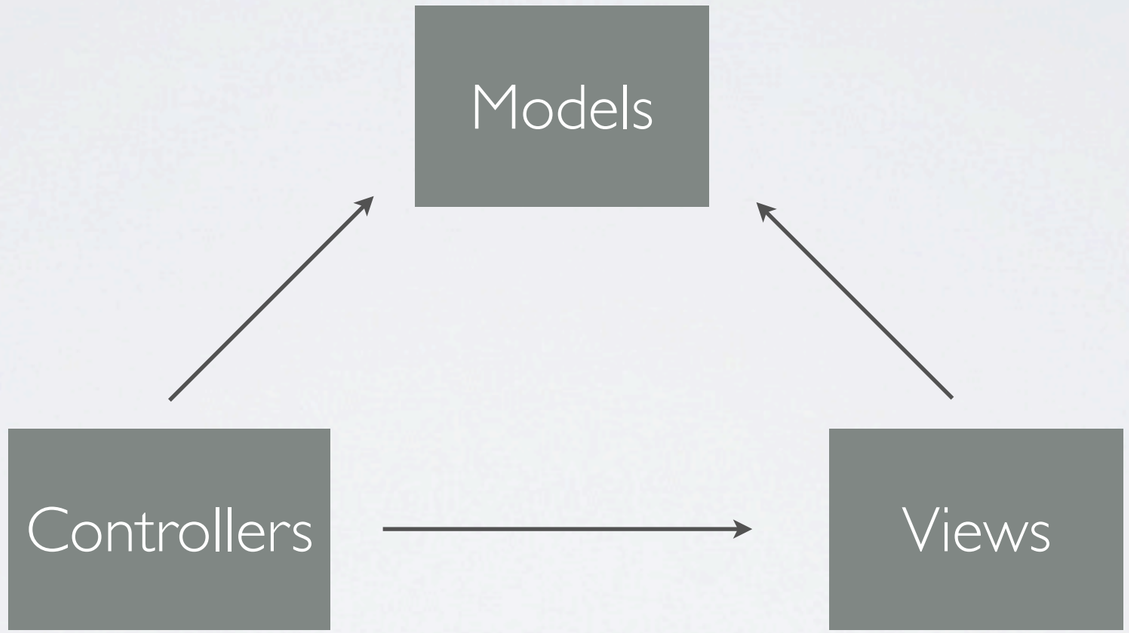
+

High Cohesion & Low Coupling

MODEL VIEW CONTROLLER



MODEL VIEW CONTROLLER



MODEL VIEW CONTROLLER

Model: is almost data

```
user = {
  id: ‘123’,
  name: ‘John’,
  surname: ‘Smith’,
  fullName: function() {
    return this.name+‘ ’+this.surname;
  },
  ...
};
```

MODEL VIEW CONTROLLER

Model: it might be also some view state

```
tabChooser = {
  tabs: [‘introduction’,‘advanced’],
  current: 0,
  next: function() {
    this.current += 1;
    this.current %= this.tabs.length;
  },
  ...
};
```

MODEL VIEW CONTROLLER

Model: knows nothing about Views and Controllers

(it is observed: callbacks, dirtycheck, ...)

```
user.addObserver(userFormView)
```

ex: look for Observer pattern for more info.

MODEL VIEW CONTROLLER

View: is almost html/css

```
<form name="user">
  <label for="name">Name:</label>
  <input name="name" type="text">
  <label for="surname">Surname:</label>
  <input name="surname" type="text">
  <input type="submit" value="Update">
</form>
```

MODEL VIEW CONTROLLER

View: + a little bit of JS

```
userFormView = {
  setUser: function(user) {
    user.addObserver(this);
    this.user = user;
    this.update();
  },
  update: function() {
    this.form.name = user.name;
    this.form.surname = surname;
  },
  ...
};
```

MODEL VIEW CONTROLLER

View: knows nothing about Controllers

(it is used: callbacks, interfaces, ...)

```
userFormView.onSubmit(controller.save)
```

MODEL VIEW CONTROLLER

Controller: is almost everything else

```
user = http.get('/user');
userFormView = new UserFormView();
userFormView.setUser(user);
userFormView.onSubmit(function() {
  http.post('/user', user);
});
viewport.setView(userFormView);
```

MODEL VIEW CONTROLLER

Controller: is almost everything else

```
user = http.get('/user');
userFormView = new UserFormView();
userFormView.setUser(user);
userFormView.onSubmit(function() {
  http.post('/user', user);
});
viewport.setView(userFormView);
```

Be Careful!

Persistence/Service Controller

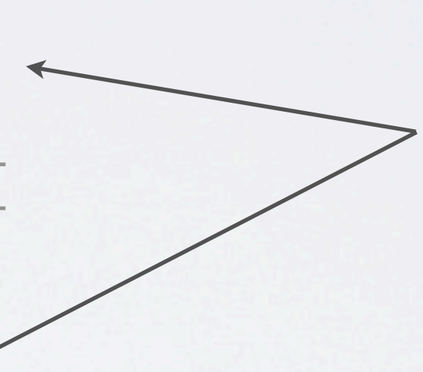
loads an stores data (may use identity map)

MODEL VIEW CONTROLLER

Controller: is almost everything else

Be Careful!

```
user = http.get('/user');
userFormView = new UserFormView();
userFormView.setUser(user);
userFormView.onSubmit(function() {
  http.post('/user', user);
});
viewport.setView(userFormView);
```



Route
Controller

ex: http://localhost:8080/#/user

MODEL VIEW CONTROLLER

Controller: is almost everything else

Be Careful!

```
user = http.get('/user');
userFormView = new UserFormView();
userFormView.setUser(user);
userFormView.onSubmit(function() {
  http.post('/user', user);
});
viewport.setView(userFormView);
```

View Controller

handle an specific view

MODEL VIEW CONTROLLER

Controller: detect controllers that you need

- Persistence/Service Controllers: load, store, manages models.
  Uses other persistence controllers and models

- View Controllers: prepare and observe views.
  Uses persistence controllers, views and (low) models.

- Route Controllers: select a view to show.
  Uses other controllers, (instance) views and (low) models.

MODEL VIEW CONTROLLER

, or Model View Adapter, or Model View ViewModel, or Model View Presenter

MVC Is NOT a Fashion!

Do not create ternaries of ModelX-ViewX-ControllerX just to follow the pattern. Create models, views and controllers because they make sense, and reuse them!

Use them wisely

COMMON SENSE PATTERNS

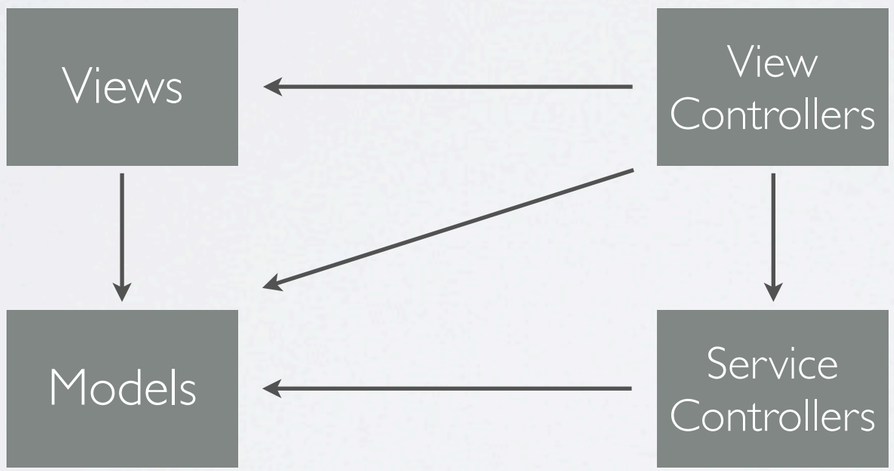
High Cohesion: put related things in the same place

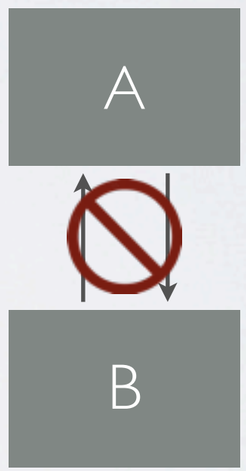
- Persistence Controllers: load/store models of one kind,
- Route Controllers: prepare a view to show,
- View: shows a related information to a user (not a model!),
- Model: describes an entity,
- ...

COMMON SENSE PATTERNS

Low Coupling: minimize what each part knows of other

- Create direction: A knows B, but B must not know A





COMMON SENSE PATTERNS

High Cohesion & Low Coupling

They are the most basic pattern to follow,
many other patterns come from them,
even Model View Controller.

Sunday 25 May 14

MODEL VIEW CONTROLLER

Model: is not html/dom

Please, don't

~~<input name="name" value="John">~~

~~<img class="status" src="success.jpg">~~

Model: is a Javascript object

```
user = {name: "John"};
status = "success";
```

SO...

Model View Controller: use it wisely

High Cohesion and Low Coupling: make it your common sense

Thanks

Code examples:

https://github.com/drpicox/tutorial-js-patterns

Deeper in code

DEEPER CODE

Persistence Controller: remote data accesses (v1)

```
userService = {
  get: function() {
    return http.get('/user');
  },
  save: function(user) {
    http.post('/user', user);
  },
};
```

Simple get and save, of data.

DEEPER CODE

Persistence Controller: remote data accesses (v2)

```
userService = {
  get: function() {
    return $http.get('/user').
    then(function (response) {
      var user = new User(response.data);
      return user;
    });
  },
  save: function(user) {
    return $http.post('/user', user);
  },
};
```

Recover data and create model instances.

DEEPER CODE

Persistence Controller: remote data accesses (v3)

```
    instance = null;
userService = {
  get: function() {
    if (instance) { return instance; }
    return instance = $http.get('/user').
    then(function (response) {
      var user = new User(response.data);
      return user;
    });
  },
  ...
};
```

Keep track of recovered instances. Not double loading, always same instance for same data. (have key? Identity map)

DEEPER CODE

View Controller: it makes the view alive

```
function UserFormController(view, user) {
  view.user = user;
  view.save = function() {
    userService.save(user);
  };
}
```

For Java programmers: imagine that view satisfies a interface.
For AngularJS programmers: view is indeed $scope.

DEEPER CODE

Route Controller: it shows what user wants (v1)

```
routeManager.when('/user', function() {
  var user = userService.get();
  var view = new UserFormView();
  var controller = new UserFormController(view, user);
  viewport.setView(view);
});
```

DEEPER CODE

Route Controller: it shows what user wants (v2)

```
routeManager.when('/user', function() {
  userService.get().
  then(function (user) {
    var view = new UserFormView();
    var controller = new UserFormController(view, user);
    viewport.setView(view);
  });
});
```

Wait for it :-)

DEEPER CODE

View: jQuery version

```
// this{ user, save }
function UserFormView() {
  var view = this;
  view.template = ‘userForm.tpl.html’;
  view.link = function(dom) {
    $(‘input[name=“name”]’).attr(‘value’,view.user.name);
    $(‘form’).submit(function() {
      view.save();
    });
  };
  ...
}
```

DEEPER CODE

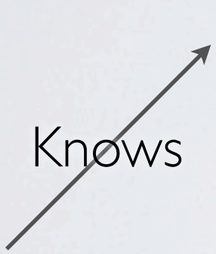
View: AngularJS version

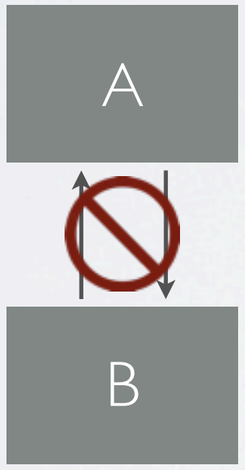
```
<form ng-submit="save()">
  <label>Name:</label>
  <input ng-model="user.name" type="text">
  <label>Surname:</label>
  <input ng-model="user.surname" name="surname" type="text">
  <input type="submit" value="Update">
</form>
```

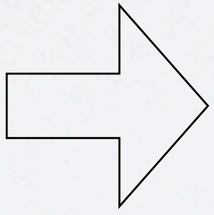
COMMON SENSE PATTERNS

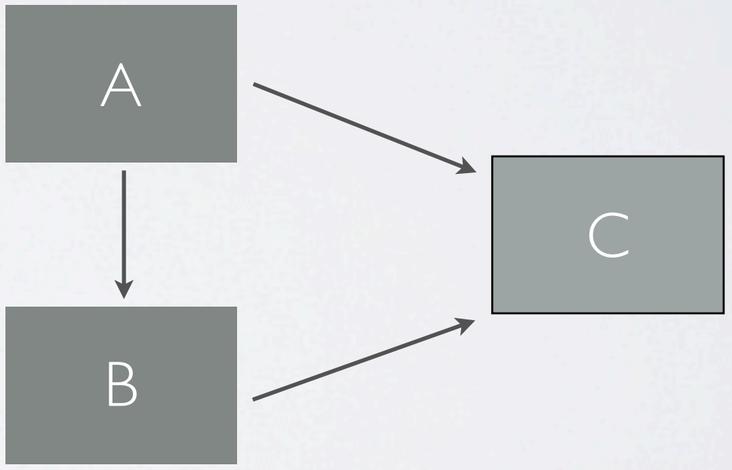
Low Coupling: minimize what each part knows of other

- Create direction: A knows B, but B must not know A









No circles!

@drpicox

When to use them?

Multiple Page Application

- PHP/JSP/RoR/... controls flow
- SEO oriented
- Server delivers anything
- Application session is in Server
- ~~Just jQuery~~

Single Page Application

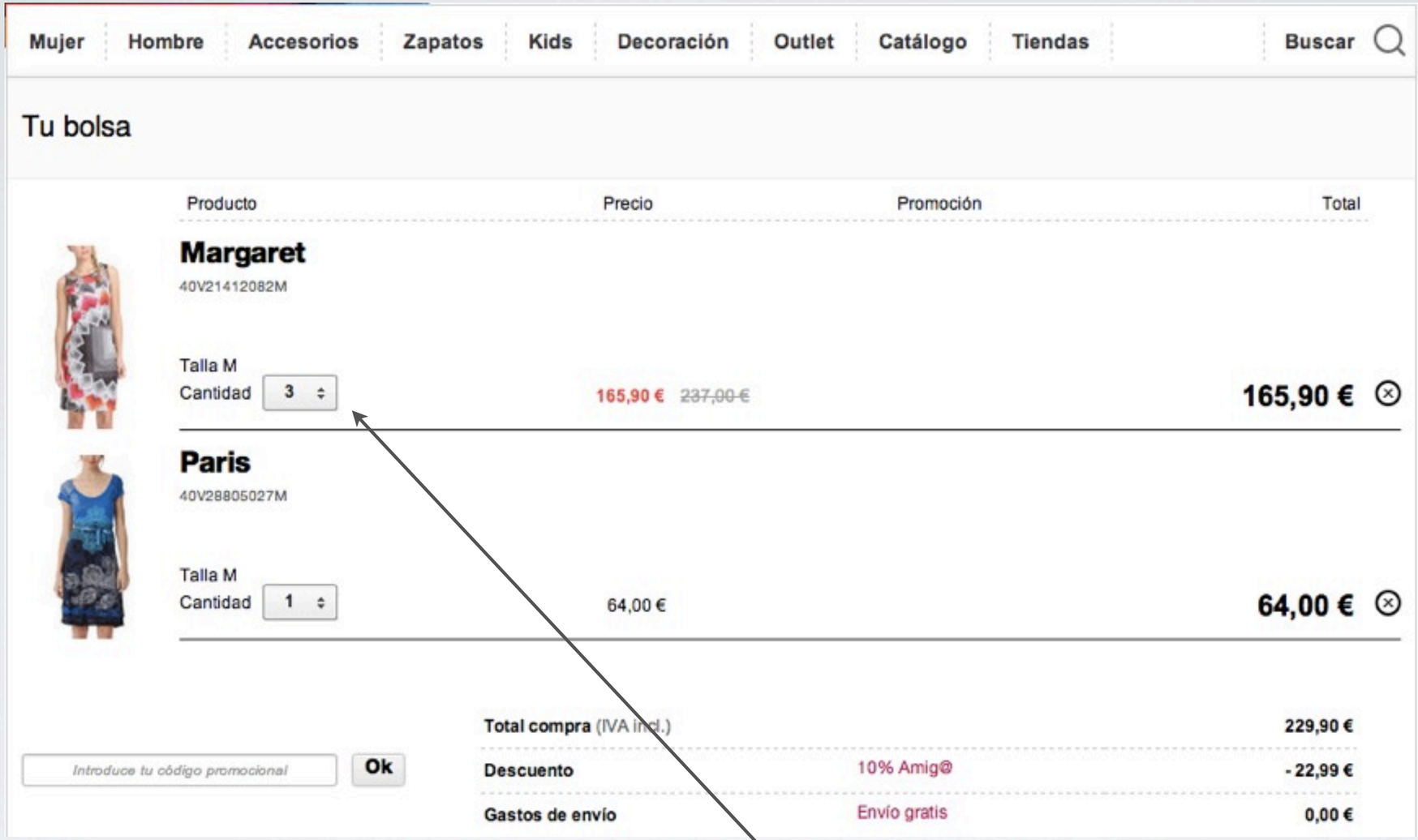
- JS controls the application flow
- SEO requires pre-render
- Server delivers assets and API
- Application session is in JS
- JS framework recommended

Multiple colors and sizes to choose

www.desigual.com

Price and other info may change

www.desigual.com



No one wants to wait here

So...

Multiple Page Applications also requires some JS framework