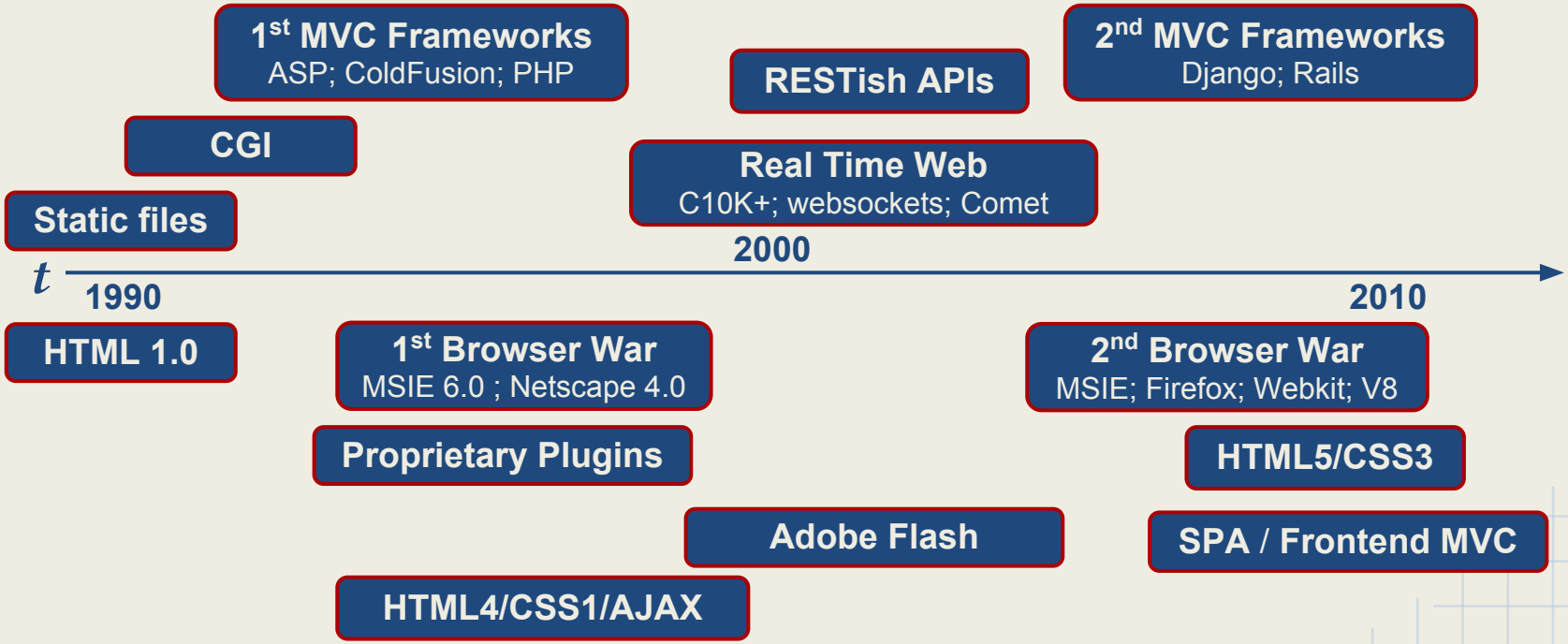
Modern Web Applications with Flask and Backbone.js

/Yaniv (Aknin|Ben-Zaken)/
February 2013

Web application? Modern?



MVC

- MVC is a design pattern
- Introduced in SmallTalk in the 70'
- Parts
  - Model - the data
  - View - the presentation of the data
  - Controller - user interaction
- Move javascript implementations are some variation on the pattern and not pure MVC

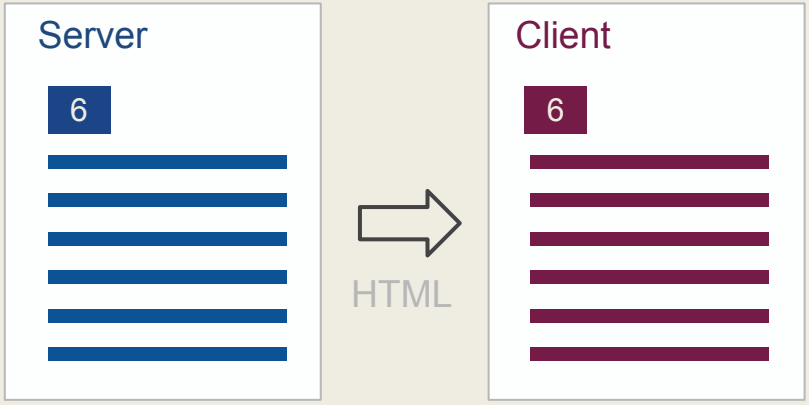
MVC

Why do we need a client side MVCish solution?

A simple list and a counter example

No client side data manipulation

Getting the list



MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

No client side data manipulation

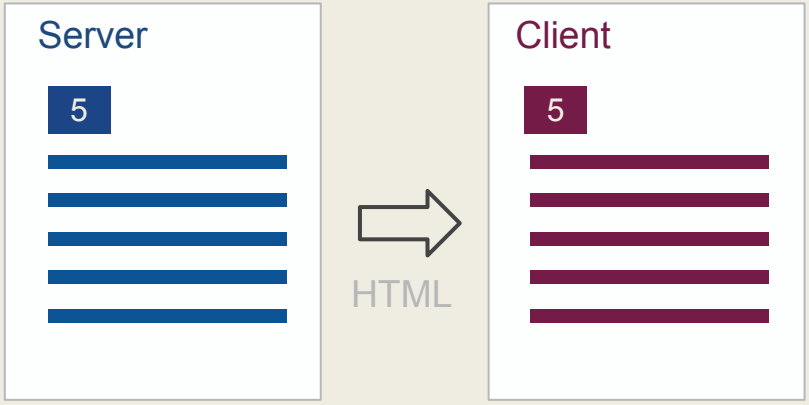
Sending a delete action

MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

No client side data manipulation

Getting an updated page - List and counter are synced

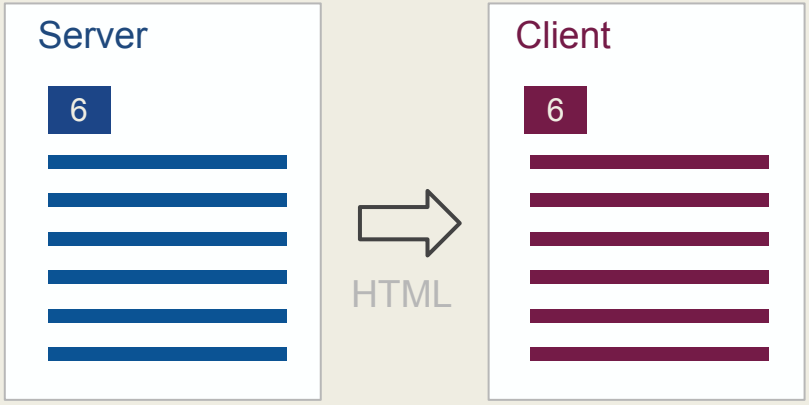


MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

Client side data manipulation by dom mutation

Getting the list



MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

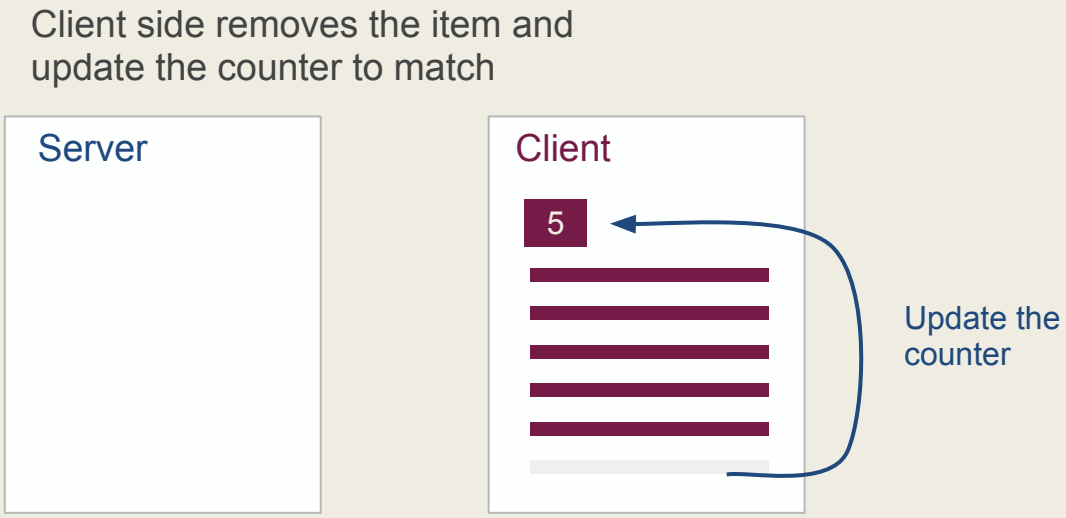
Client side data manipulation by dom mutation

Sending a delete request and getting confirmation

MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

Client side data manipulation by dom mutation



The down-side of the data being mixed with the representation is that each data manipulation code must be aware of all the existing ways it is shown, and update them.

MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

Client side data manipulation by dom mutation

Updating every other representation of this data

Suppose we want to have a thumbnail list widget in the corner, we'll have to update the delete code to update that as well.

MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

Client side data manipulation by dom mutation

Updating every other representation of this data

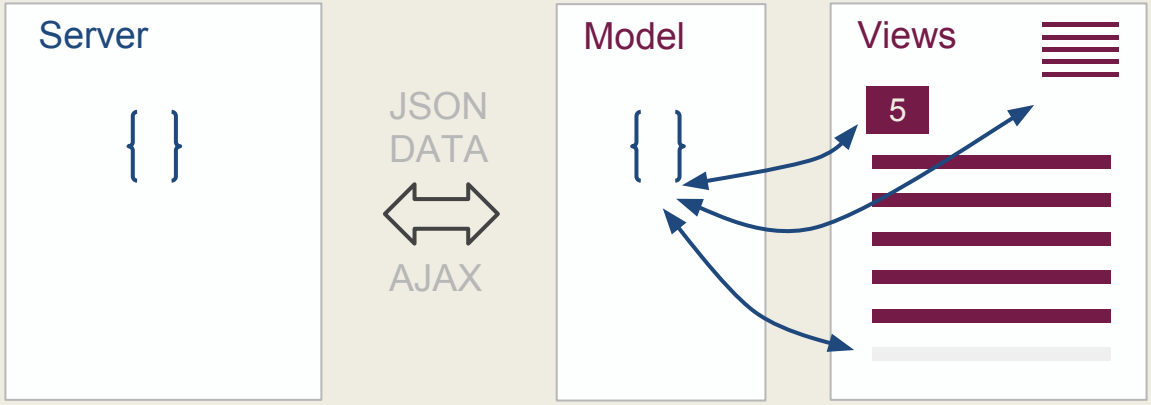
Suppose we want to have a thumbnail list widget in the corner, we'll have to update the delete code to update that as well.

MVC

Why do we need a client side MVCish solution?
A simple list and a counter example

Client side data manipulation using a model and mutiple views

The server sent only data, not html.
The html is built by the views.



After a delete, the model is changed.
The views Observe the model and update themselves after a change. The model isn't aware of the different views and their implementation.
Code is easier to maintain.

Web Application: concepts

Backend

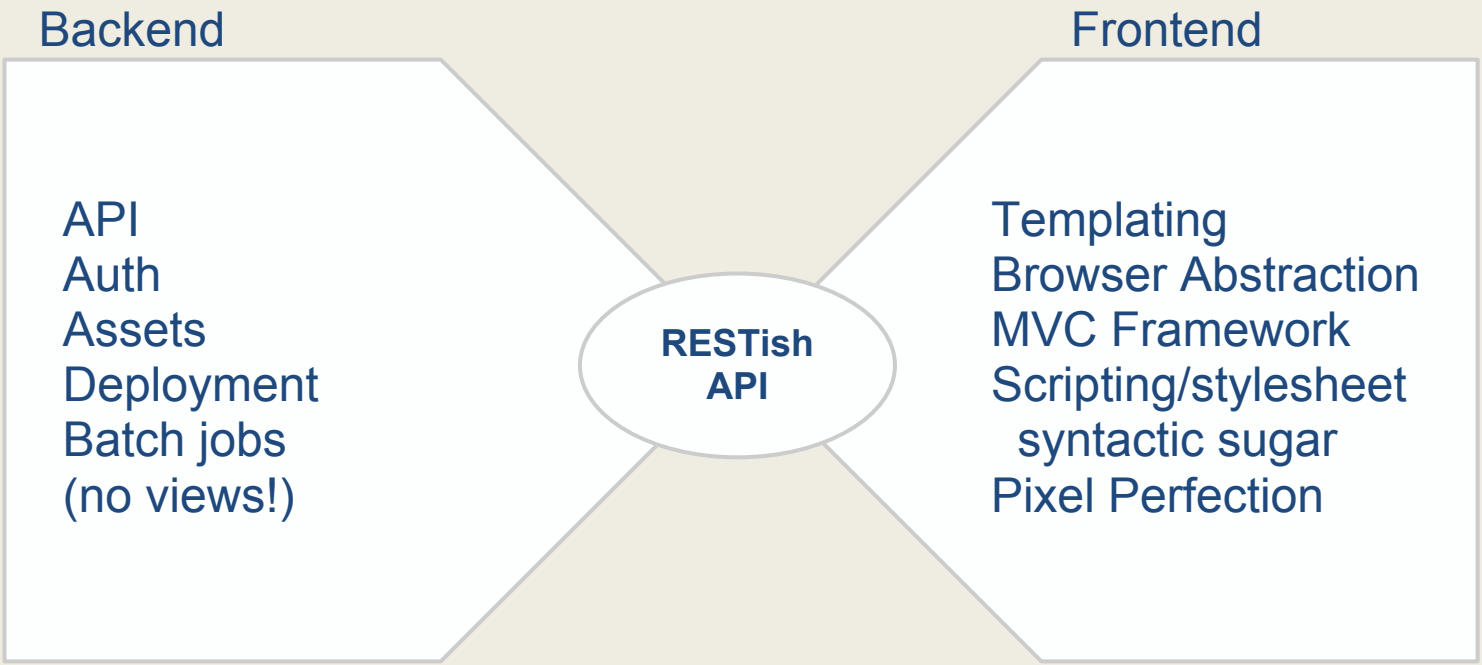
Persistence Logic
(Resource State)
Pages and Assets
Misc. Legacy resources

Frontend

UI
Layout
Validation
Non persistent logic
(Session state)

Easy to get started - challenges are typically with deployment, scale, monitor, secure, etc

Web Application: roles



RESTish

- REST: Architectural style pioneered by Roy Fielding
- Often misunderstood and misimplemented
- Strict adherence to constraints defines RESTfulness
- RESTish is what we call "REST inspired" services
- Common principles
  - client-server over HTTP
  - resource oriented & uniform interface
  - stateless (or stateleast)
  - layered and cachable
  - not really RESTful...

Crash course: FLSK 101

```python
#!/usr/bin/env python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(debug=True)
```

Things to keep in mind

- Flask is surprisingly thin
- Read the source, Luke
- Get familiar with Werkzeug
- Brilliant context locals are brilliant
- Flask in under ten lines:

```
class Flask(_PackageBoundObject):
    ...
    def wsgi_app(self, environ, start_response):
        with self.request_context(environ):
            try:
                response = self.full_dispatch_request()
            except Exception, e:
                rv = self.handle_exception(e)
                response = self.make_response(rv)
            return response(environ, start_response)
```

Crash course: FLSK 102

```python
#!/usr/bin/env python
import os
from httplib import ACCEPTED, FORBIDDEN
from flask import Flask, request
app = Flask(__name__)

@app.route('/', methods=['DELETE'])
def reboot():
    if request.values.get('password') == 'secret':
        os.system('sudo shutdown now')
        return 'ok', ACCEPTED
    return 'no', FORBIDDEN

if __name__ == '__main__':
    app.run()
```

Crash course: FLSK 103

```
@app.route('/')
def show_entries():
    cur = g.db.execute(SELECT_SQL)
    ent = [dict(ttl=r[0], txt=r[1]) for r in cur.
fetchall()]
    return render_template('show_entries.html', ent=ent)

@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(UNAUTHORIZED)
    g.db.execute(INSERT_SQL, [request.form['title'],
                 request.form['text']])
    g.db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

Crash course: FLSK 103

```
@app.route('/')
def show_entries():
    cur = g.db.execute(SEL      SQL)
    ent = [dict(ttl=r[0],      r[1]      r in cur.
fetchall()]
    return render_template(      ries.html', ent=ent)

@app.route('/add', methods      )
def add_entry():
    if not session.get(      ed_
        abort(UNAUTHORIZ      )
    g.db.execute(INSERT_SQL, [request.form['title'],
                 request.form['text']])
    g.db
    flas                                    )
    return redirect(uri_for(
```

This is server side MVC!
(read: not our cup of tea)

Crash course: FLSK 103

```
@app.route('/user/<int:id>')
def user_page(id):
    User.query.get_or_404(id)
    return render_template('user.html',
        user_id = id,
    )
..............................................................
{% extends "webapp.html" %}

{% block head %}
    {{ super() }}
    <script type="text/javascript">
       window.xx.context.user_id = {{ user_id|tojson|safe }};
        $(function() { UserPage({el: $('.user_page')}); });
    </script>
    <script type="text/javascript" src="{% asset "user.js" %}"></script>
    <link rel="stylesheet" href="{% asset "user.css" %}"/>
{% endblock %}

{% block body %}
{{ super() }}
<div class="user_page">
    <div id="header_bar"><div class="logo"></div></div>
</div>
{% endblock %}
```

Crash course: FLSK 103

```
@app.route('/user/<int:id>')
def user_page(id):
    User.query.get_or_404(id)
    return render_template('user.html',
        user_id = id,
    )
.................................................................
{% extends "webapp.html" %}

{% block head %}
    {{ super() }}
    <script type="text/javascript">
       window.xx.context.user_id = {{ user_id|tojson|safe }};
        $(function() { UserPage({el: $('.user_page')}); });
    </script>
    <script type="text/javascript" src="{% asset "user.js" %}"></script>
    <link rel="stylesheet" href="{% asset "user.css" %}"/>
{% endblock %}

{% block body %}
{{ super() }}
<div class="user_page">
    <div id="header_bar"><div cl
</div>
{% endblock %}
```

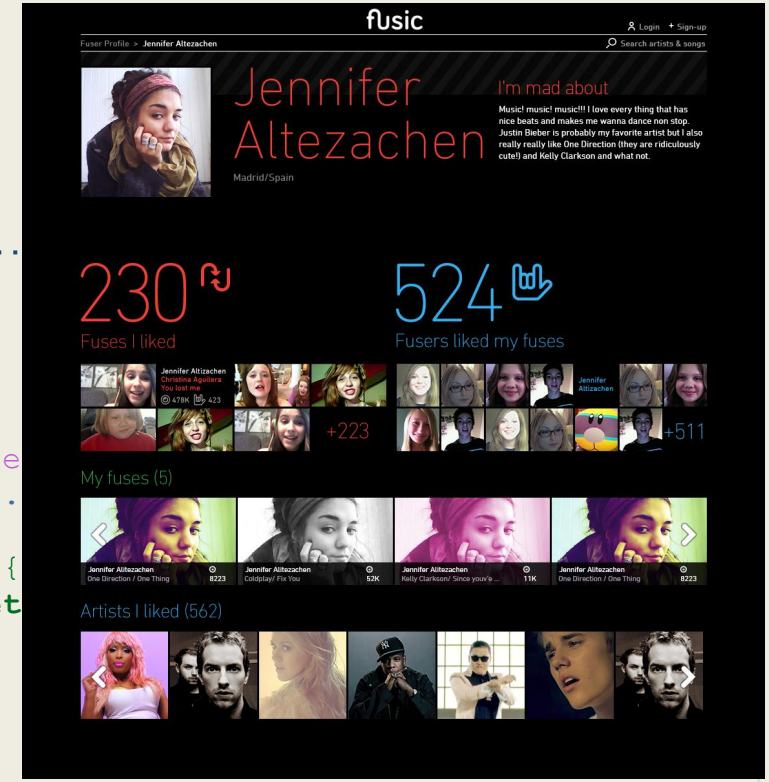
Complete View & Template of user page
(you're not seeing much here - that's the point!)

Crash course: FLSK 103

```
@app.route('/user/<int:id>')
def user_page(id):
    User.query.get_or_404(id)
    return render_template('user.html',
        user_id = id,
    )
.................................................
{% extends "webapp.html" %}

{% block head %}
    {{ super() }}
    <script type="text/javascript">
       window.xx.context.user_id = {{ use
        $(function() { UserPage({el: $('.
    </script>
    <script type="text/javascript" src="{
    <link rel="stylesheet" href="{% asset
{% endblock %}

{% block body %}
{{ super() }}
<div class="user_page">
    <div id="header_bar"><div cl
</div>
{% endblock %}
```



Complete View & Template of user page
(you're not seeing much here - that's the point!)

flask-assets crash course

- Flask extension wrapping the webassets package
- Declaratively define your pages' assets
- Assets are filtered during deployment / on the fly (development)

```
from flask.ext.assets import Environment, Bundle

from .app import app

assets = Environment(app)
coffee = Bundle('js/lib/lib.coffee', 'js/lib/auth.coffee',
                filters="coffeescript", debug=False,
                output="gen/base.coffee.js")
js = Bundle('js/vendor/underscore.js',
            'js/vendor/backbone.js', coffee
            output="gen/base.js", filters="yui_js")
assets.register('base.js', js)
```

flask-assets crash course

- Flask extension wrapping the webassets package
- Declaratively define your pages' assets
- Assets are filtered during deployment / on the fly (development)

```
from flask.ext.assets import Environment

from utils.flaskutils import register_assets
from .app import app

spec = {
    "base.js": ('js/lib/lib.coffee',
                'js/lib/auth.coffee',
                'js/vendor/underscore.js',
                'js/vendor/backbone.js')
}
register_assets(assets, spec)
```

flask-restful crash course

- *"Flask-RESTful provides the building blocks for creating a great REST API"*
  - Read: *"...for easily creating a decent RESTish API"*
  - Intensely down-to-business, not coupled to anything (db, etc)
  - Highly recommended, still under development

```
from myproject import app
from flask.ext import restful

api = restful.Api(app)

class HelloWorld(restful.Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')
```

flask-sqlalchemy crash course

- Flask extension wrapping the incredible SQLAlchemy package
- Every ORM I saw trades power for simplicity - SA trades very little power yet stays decently simple
- Great layered approach from high level ORM to declarative API for SQL

```
from flask.ext.sqlalchemy import SQLAlchemy
from .app import app
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    def __init__(self, username):
        self.username = username
    def __repr__(self):
        return '<User %r>' % self.username
```

Two great tastes that taste great together!

```
class Marshallable(Resource):
    method_decorators = (marshal_with(self.fields),)

class Entity(Marshallable):
    def build_query(self):
        return self.model.query
    def get(self, id):
        return self.build_query().get_or_404(id)

class UserMixin(object):
    fields = {"username": String}
    model = models.User
class User(UserMixin, Entity):
    pass

api.add_resource(User, '/api/users/<int:id>',
                 endpoint='api_user')
```

Suggested project layout

```
$ find * -maxdepth 2 | vi -
```

```
manage.py
requirements.txt
runcommands.sh
backend/
    api/
    app.py
    assets.py
    auth.py
    models.py
    static@
    templates/
    views.py
config/
    settings.py
frontend/
    index/
        core.coffee
        core.scss
        hello_world.jst
    <more pages>
utils/
<more app specific code>/
```

Suggested project layout

```
$ find * -maxdepth 2 | vi -
```

```
manage.py
requirements.txt
runcommands.sh
backend/
    api/
    app.py
    assets.py
    auth.py
    models.py
    static@
    templates/
    views.py
```

```
config/
    settings.py
frontend/
    index/
        core.coffee
        core.scss
        hello_world.jst
    <more pages>
utils/
<more app specific code>/
```

Bonus slide: authentication

- Make sessions a resource
- PUT for login, DELETE for logout, GET for whoami
- Easy to implement with flask-login and flask-restful

```
class Session(Resource):
    fields = dict(basic_user_fields)
    def marshal(self, user):
        return marshal(user, self.fields)
    def get(self):
        if current_user:
            return self.marshal(current_user), OK
        return None, NO_CONTENT
    def delete(self):
        if not current_user:
            return None, NO_CONTENT
        logout_user()
        return None, RESET_CONTENT
    @parse_with(Argument('kind', required=True, choices=backends))
    def put(self, params):
        try:
            user, created = backends[params.kind]()
            return self.marshal(user), CREATED if created else OK
        except InvalidAuth, error:
            return {"message": error.msg}, error.status
```

Backbone

- Provides small amount of useful building blocks and the rest is up to you
- Has a strong following and a lot of extensions are available
- Main parts:
  - Model
  - View
  - Collection
  - Router
  - Events - All of the parts can be used with a pub/sub pattern using backbone events methods
  - (*Templates) - Backbone doesn't come with templating, you can use any js templating solution. We will use underscore's templates.

Backbone

The files which make up our example's frontend app:

```
AppView.coffee
Router.coffee
TodoCollection.coffee
TodoModel.coffee
TodoView.coffee
app.coffee
css
img
jst
```

Backbone - Model

- Used for data
- You get listen to change events
- Can be have a url and be synced with the backend

```
window.jj.TodoModel = class TodoModel extends Backbone.Model
    defaults:
        title: ''
        completed: false
    toggle: ->
        @save { completed: !@get('completed') }
```

Backbone - Collection

- A collection is a list of models
- Useful events for a model add, remove, change, etc
- Also can be fetched or sent as whole to the server

```
window.jj.TodoCollection = class TodoCollection extends Backbone.Collection
    model: jj.TodoModel
    url : "/api/todos/"
    completed: ->
        return @filter((todo) ->
            return todo.get('completed')
        )
    remaining: ->
        return @without.apply @, @completed()
    nextOrder: ->
        if not @length
            return 1
        return @last().get('order') + 1
    comparator: (todo) ->
        return todo.get('order')
```

Backbone - View - single item

- Handles a dom element or sub-views
- Open for interpretation, and has many ways to use
- Usually will listen to a Model's changes and re-render itself when needed

```
window.jj.TodoView = class TodoView extends Backbone.View
    tagName:  'li'
    template: _.template jj.jst["todo/jst/item.jst"]
    events:
        'click .toggle': 'toggleCompleted'
        'dblclick label':     'edit'
        'click .destroy':     'clear'
        'keypress .edit':     'updateOnEnter'
        'blur .edit':         'close'
    initialize: ->
        @listenTo @model, 'change', @render
        @listenTo @model, 'destroy', @remove
        @listenTo @model, 'visible', @toggleVisible

    render: ->
        @$el.html @template(@model.toJSON())
        @$el.toggleClass 'completed', @model.get('completed')
        @toggleVisible()
        @$input = @$('.edit')
        return @
```

User interaction

Listens to model changes

Render the element

Backbone - View - single item

```
    toggleVisible: ->
        @.$el.toggleClass 'hidden',  @isHidden()

    isHidden: ->
        isCompleted = @model.get('completed')
        return (!isCompleted and (jj.app?.TodoFilter is 'completed')) or (isCompleted
and (jj.app?.TodoFilter is 'active'))

    toggleCompleted: ->
        @model.toggle()

    edit: ->
        @$el.addClass 'editing'
        @$input.focus()

    close: ->
        value = @$input.val().trim()
        if value
            @model.save { title: value }
        else
            @clear()
        @$el.removeClass 'editing'
    updateOnEnter: (e) ->
        if e.which is jj.ENTER_KEY
            @close()

    clear: ->
        @model.destroy()
```

Update the model

Remove the model

Backbone - View - items list

- We can see the view manages item views
- Also listens to the router for url/hash changes and updates state

```
window.jj.AppView = class AppView extends Backbone.View
    template: _.template jj.jst["todo/jst/app.jst"]
    statsTemplate: _.template jj.jst["todo/jst/stats.jst"]
    events:
        'keypress #new-todo': 'createOnEnter'
        'click #clear-completed': 'clearCompleted'
        'click #toggle-all': 'toggleAllComplete'
    initialize: ->
        @buildElement()
        @allCheckbox = @$('#toggle-all')[0]
        @$input = @$('#new-todo')
        @$footer = @$('#footer')
        @$main = @$('#main')

        @listenTo @collection, 'add', @addOne
        @listenTo @collection, 'reset', @addAll
        @listenTo @collection, 'change:completed', @filterOne
        @listenTo @collection, 'all', @render

        @initRouter()

        @collection.fetch()

    initRouter : ->
        @router = new jj.Router()
        @listenTo @router, 'route:filterSet', @updateFilter
```

User interaction events

Listens to collection changes

Listens to the router

Backbone - View - items list

```
    buildElement: ->
        @.$el.html @template()
    render: ->
        completed = @collection.completed().length
        remaining = @collection.remaining().length
        if @collection.length
            @$main.show()
            @$footer.show()
            @$footer.html @statsTemplate({
                completed: completed
                remaining: remaining
            })
            @$('#filters li a').removeClass('selected').filter('[href="#/' + ( jj.app?.TodoFilter or '' )
+ '"]').addClass('selected')
        else
            @$main.hide()
            @$footer.hide()
        @allCheckbox.checked = !remaining

    updateFilter : (param) ->
        jj.app.TodoFilter = param.trim() or ''
        @filterAll()

    addOne: (todo) ->
        view = new jj.TodoView({ model: todo })
        $('#todo-list').append view.render().el

    addAll: ->
        @$('#todo-list').html('')
        @collection.each(@addOne, @)
```

...

Building the main element

Managing visibility according to filters

Managing the list items rendering

Backbone - Launching it all

- We initialize our main view and pass to it the dom element it is responsible for and the collection
- In general it is easier to maintain views that don't expect dom elements to be already present or models that are on a expected global (principle of least privilege)

Existing DOM:

```
    <section id="todoapp"></section>
    <div id="info">
        <p>Double-click to edit a todo</p>
        <p>Based on the work of <a href="https://github.
com/addyosmani">Addy Osmani</a></p>
        <p>Part of <a href="http://todomvc.com"
>TodoMVC</a></p>
        <p>Modified by Yaniv Ben-Zaken (@cizei)</p>
    </div>
```

app.coffee code:

```
$ ->
    jj.app = new jj.AppView
        el : $('#todoapp')
        collection : new jj.TodoCollection

     Backbone.history.start()
```

This all the HTML and JS you have in order to launch the app. All the rest is in templates which get rendered after by code this 4 js lines launch.

Backbone - Template

- Backbone come with it's own templating system
- There are many libraries for js-templates you can plug in and use, in most of them basically you convert a string to function which will accept a key value pairings as context and give you back an html string

Item template example (using underscore.js templates**):**

```
<div class="view">
    <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
    <label><%- title %></label>
    <button class="destroy"></button>
</div>
<input class="edit" value="<%- title %>">
```

Web Application: packages

Backend

flask-restful
flask-sqlalchemy
flask-login
flask-assets
flask-script
rq
(postgres, redis,
gunicorn, gevent)

RESTish API

Frontend

Backbone.js
jQuery
underscore.js
CoffeeScript
SCSS

Additional resources

- **flask-todo: "modern webapp" based on real code**
  https://github.com/fusic-com/flask-todo

- **In Flask we Trust** (great Flask primer)
  http://ua.pycon.org/static/talks/davydenko.pdf

- **Thoughts on RESTful API Design** (don't do REST without this!)
  https://restful-api-design.readthedocs.org/en/latest/

- **Fielding on poor REST implementations** (read the links, too)
  http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
  http://roy.gbiv.com/untangled/2008/specialization

- **Addy Osmani's TodoMVC**
  http://addyosmani.github.com/todomvc/

- **Coverage of MVC history and client side solutions** http://addyosmani.com/blog/digesting-javascript-mvc-pattern-abuse-or-evolution/

Thank you!

yaniv@aknin.name (`@aknin`)
me@yaniv.bz (`@cizei`)

(pssst: All that stuff sounds cool? fusic is hiring! Talk to us!)