Globant

we are ready

Front End Engineer

Introduction to Backbone

TOPICS

Introduction to MV* Patterns
MVC vs MV vs MVVM vs MVP
Introduction to Backbone
Views
Models
Routes

Now, you can see:

01 Why use a framework

http://youtu.be/m1nq8wWVyZM

INTRODUCTION TO MV* PATTERNS

MVC (Model-View-Controller) is an architectural design pattern that promotes a clear separation of concerns.

The key point in this pattern is the separation between Model objects, which represent entities from our problem domain, and Presentation/View objects, which are responsible for showing data on the screen.

Many variation for the same pattern have emerged, but all of them share the same final goal: improve the separation of concerns.

MVC VS MVP VS MVVM

MV* comes in different flavours

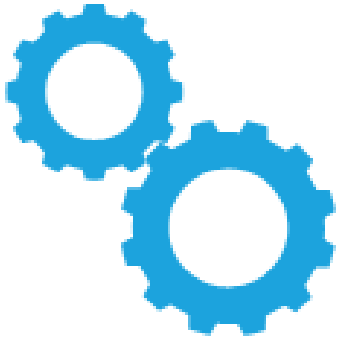
- MVC: popularized by the Smalltak-80 community. Defines View, Model, and Controller objects.
- MVP: popularized by IBM in the early 90'. Introduces a Presenter object which sits between the View, and the Model objects.
- MVVM: introduced by Microsoft for their WPF, and Silverlight technologies. In this variant, a ViewModel object act as a bridge between the View, and the Model objects.

- The main difference lies in the use of a Controller, Presenter, or ViewModel object.

MVC

Model
Represents data
or state

View
Presentation and
user controls

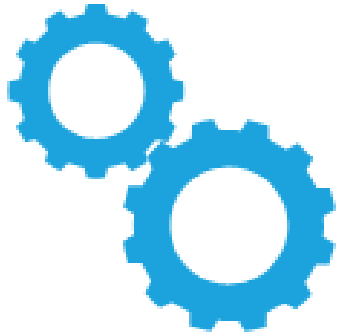


Controller
Logic between views
and models

MVP

Model
Represents data
or state

View
Presentation and
user controls

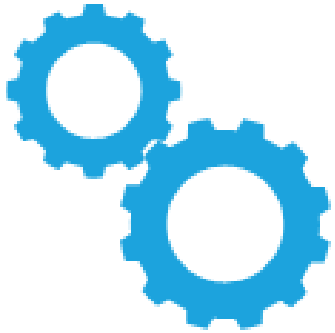


Presenter
Logic between views
and models

MVVM

Model
Represents data
or state

View
Presentation and
user controls



ViewModel
Logic between views
and models

Now, you can see:

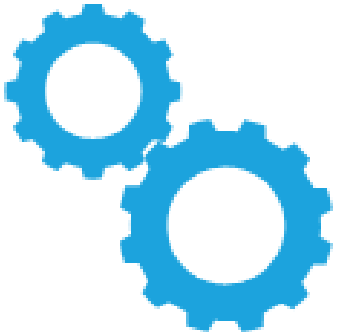
02 MVC is the MVP
03 Framework concepts
04 Finding the right solution

http://youtu.be/Fx1c7K9afc4
http://youtu.be/uESFUPqrLjQ
http://youtu.be/ehyEyfMStxI

MVC VS MVP VS MVVM

Model
Represents data
or state

View
Presentation and
user controls

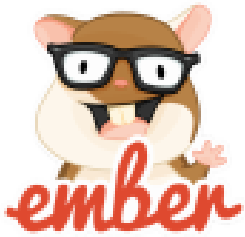


Now, you can see:

05 MVC vs MVP vs MVVM

http://youtu.be/ElHslw5wMWg

MV* FRAMEWORKS



SINGLE PAGE APPLICATIONS

Single Page Applications (SPA), are web applications running on the browser, completely written in Javascript.

SPA moves the application state from the server-side to the client-side. They generate DOM elements using client-side templates

- The server is not responsible of generating HTML, only to provide an API for the webapp to access data
- An object model resides in the client, and is synchronized with the server as the user interacts with the application

SPA & MVC

SPA usually follow an MV* architecture

- HTML templates define data-display
- JS View objects listen Model objects for changes, and update screen accordingly
- JS Models fetch data from the server, they are usually observables
- JS Controllers coordinate work among Views, and Models, and usually set up their dependencies

REST is

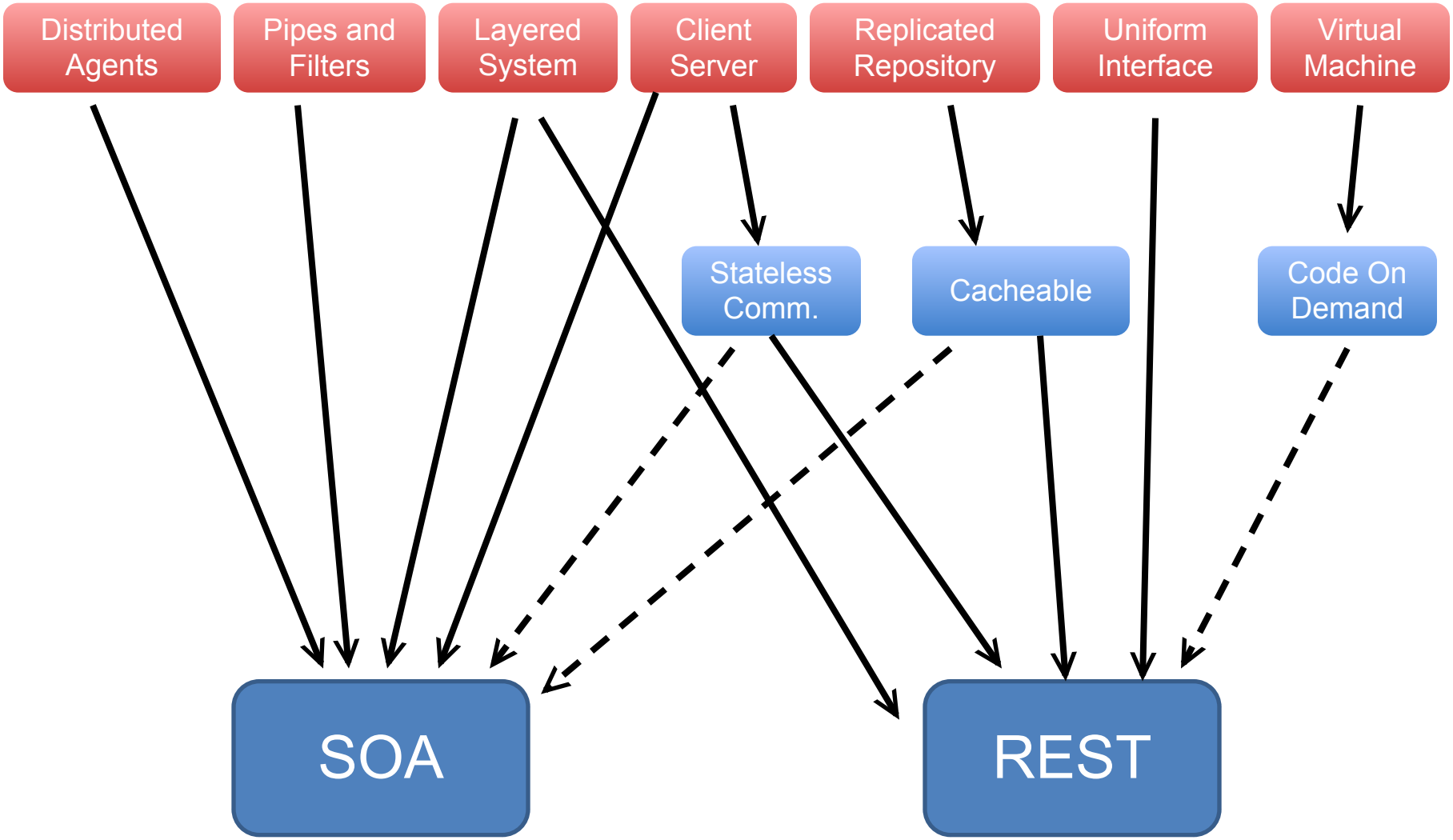
An acronym from: Representational State Transfer

An architectural style that is derived by applying a set ***constraints*** to the elements of a system so the system exhibits certain ***properties*** such as loose coupling and scalability

Defined by Roy T. Fielding, PhD in his *PhD dissertation, Chapter 5 Representational State Transfer (REST)*

Data-orientated, “Unlike the distributed object style, where all data is encapsulated within and hidden by the processing components, the nature and state of an architecture's data elements is a key aspect of REST”

SOA vs. REST


Distributed Agents
Pipes and Filters
Layered System
Client Server
Replicated Repository
Uniform Interface
Virtual Machine
Stateless Comm.
Cacheable
Code On Demand
SOA
REST

When can I use REST?

For Web Services

- build your web service using the REST style
- alternative to some of WS-*, not a replacement for WS-*

Clients interfacing to public REST APIs

- e.g. Amazon S3 REST API, Google Data APIs
- 63 percent public APIs have a REST *like* interface

From Rich Internet Applications (RIAs)

- client sends AJAX requests to a REST interface using a JavaScript library e.g. jQuery, Dojo (JsonRestStore) or a framework like JavaFX or Silverlight
- response (JSON, XML etc) is displayed on the client

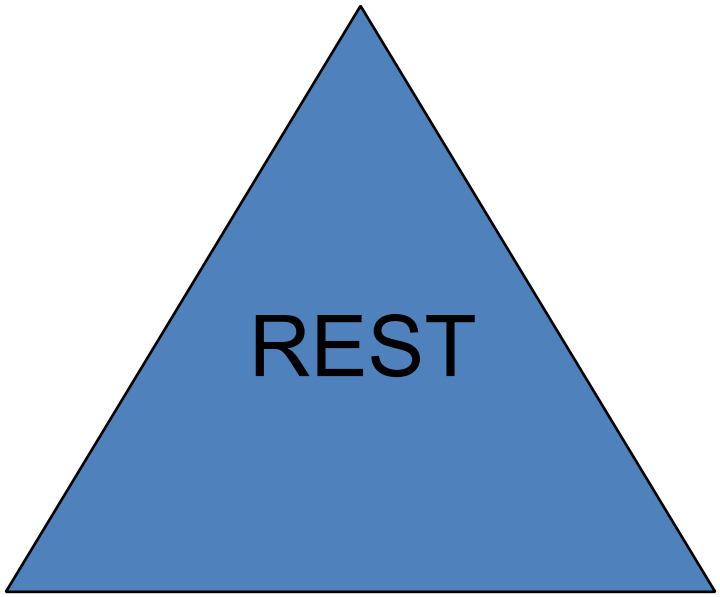
Now, you can see:

06 Intro to REST

http://youtu.be/UGTQaaZW6K4

Main Concepts

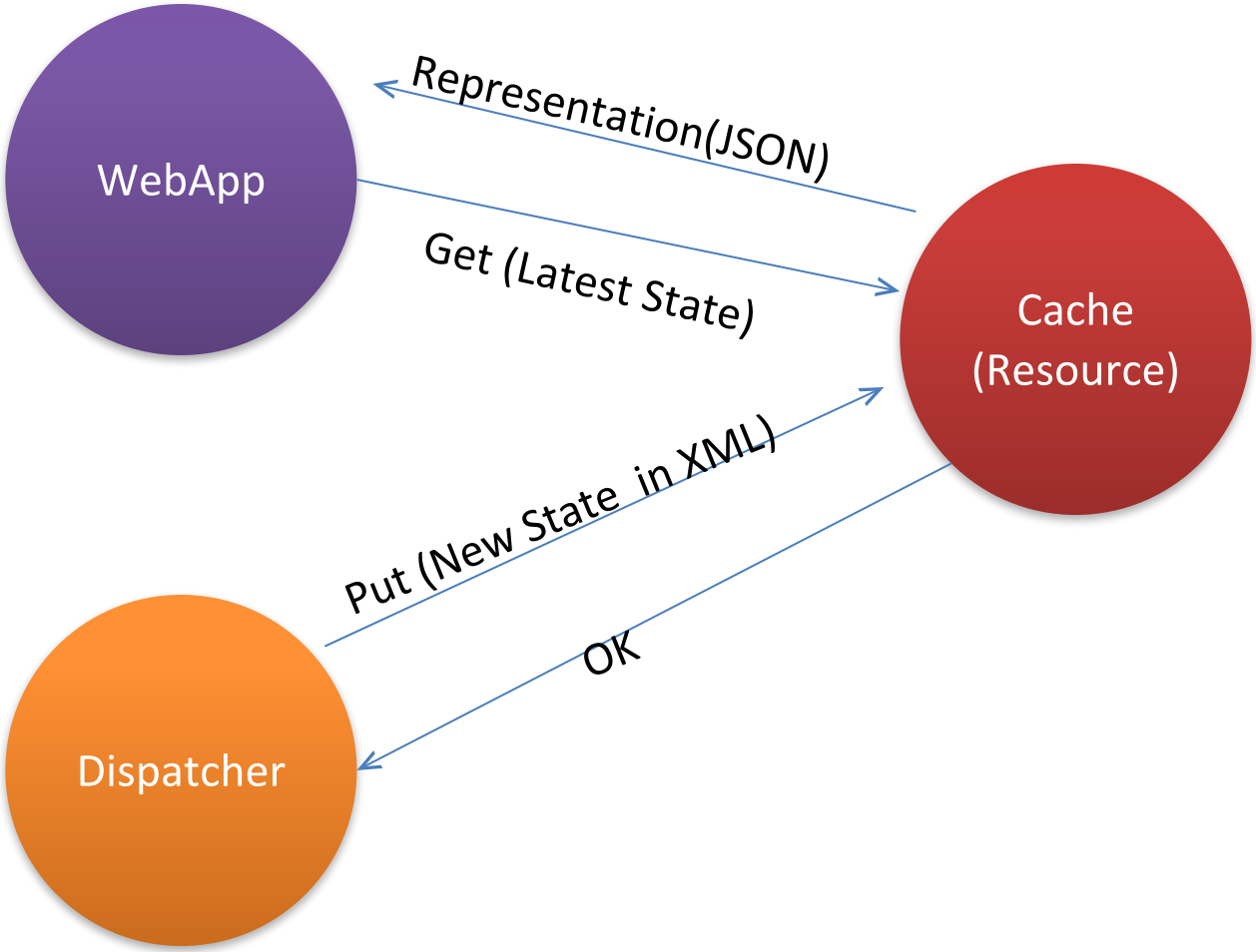
Nouns (Resources)
unconstrained
i.e., http://example.com/employees/12345



Verbs
constrained
i.e., GET

Representations
constrained
i.e., XML

REST Representation State Transfer



Resources

The key abstraction of information in REST is a resource.

A resource is a conceptual mapping to a set of entities

Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on

Represented with a global identifier (URI in HTTP)

Naming Resources

REST uses URI to identify resources

http://localhost/books/
http://localhost/books/ISBN-0011
http://localhost/books/ISBN-0011/authors

http://localhost/classes
http://localhost/classes/cs2650
http://localhost/classes/cs2650/students

As you traverse the path from more generic to more specific, you are navigating the data

VERBS

Represent the actions to be performed on resources

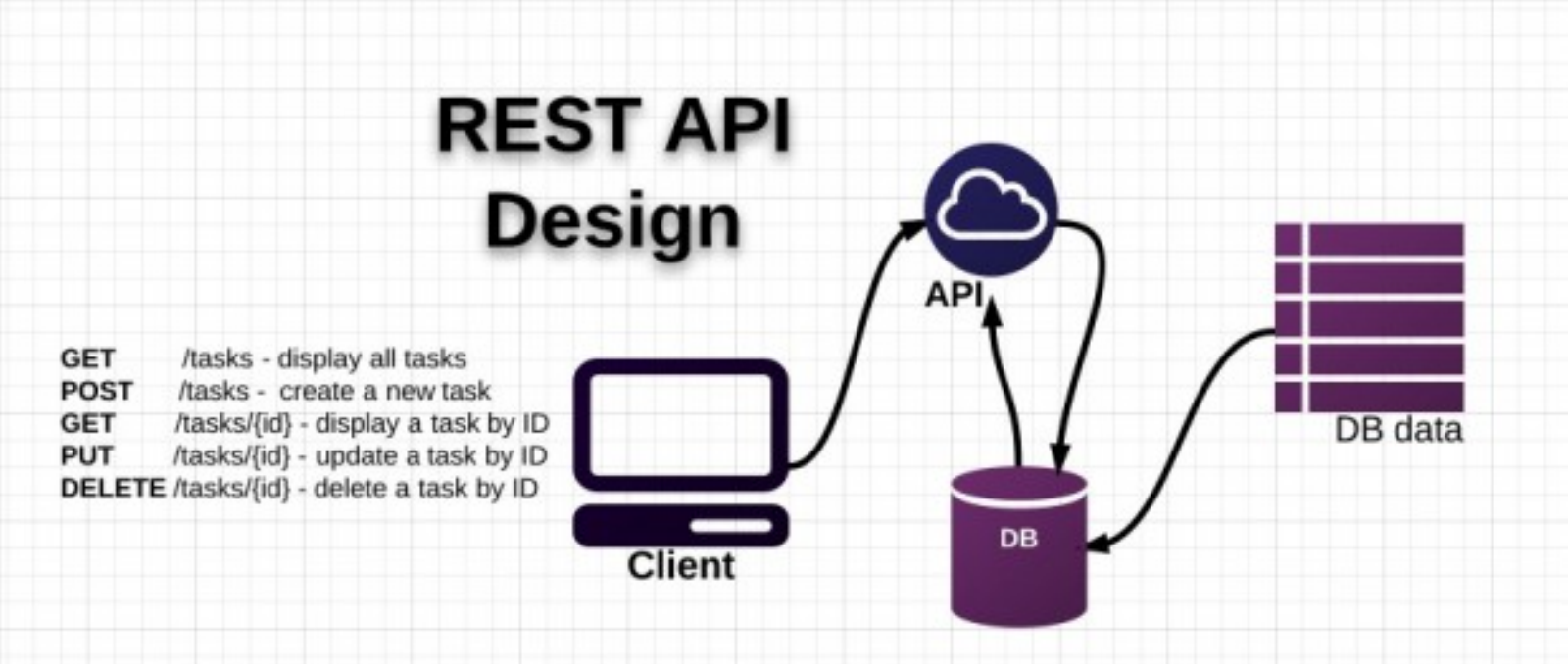
HTTP GET

HTTP POST

HTTP PUT

HTTP DELETE

VERBS


REST API
Design
GET /tasks - display all tasks
POST /tasks - create a new task
GET /tasks/{id} - display a task by ID
PUT /tasks/{id} - update a task by ID
DELETE /tasks/{id} - delete a task by ID
API
Client
DB
DB data

VERBS

HEAD =
GET but only retrieves metadata (headers)

GET

GET

How clients ask for the information they seek.

Issuing a GET request transfers the data from the server to the client in some representation

GET http://localhost/books

- Retrieve all books

GET http://localhost/books/ISBN-0011021

- Retrieve book identified with ISBN-0011021

GET http://localhost/books/ISBN-0011021/authors

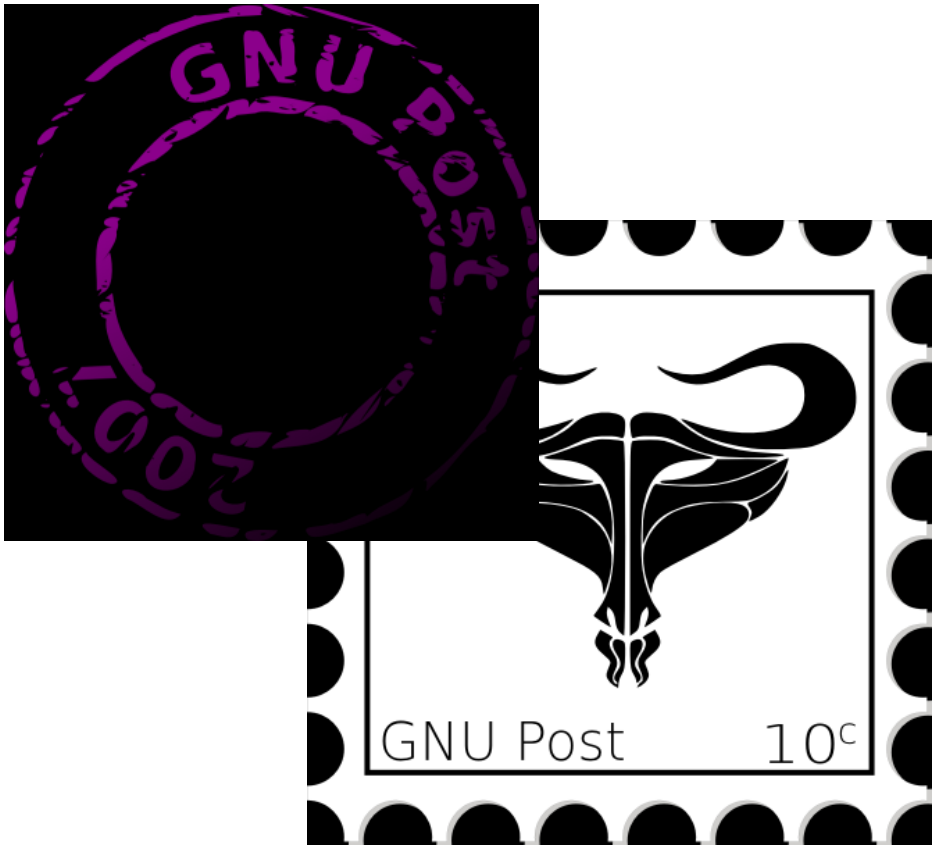
- Retrieve authors for book identified with ISBN-0011021

PUT

PUT = create with ID set by the client / replace

POST

POST = Create a subordinate resource with ID set by server / update



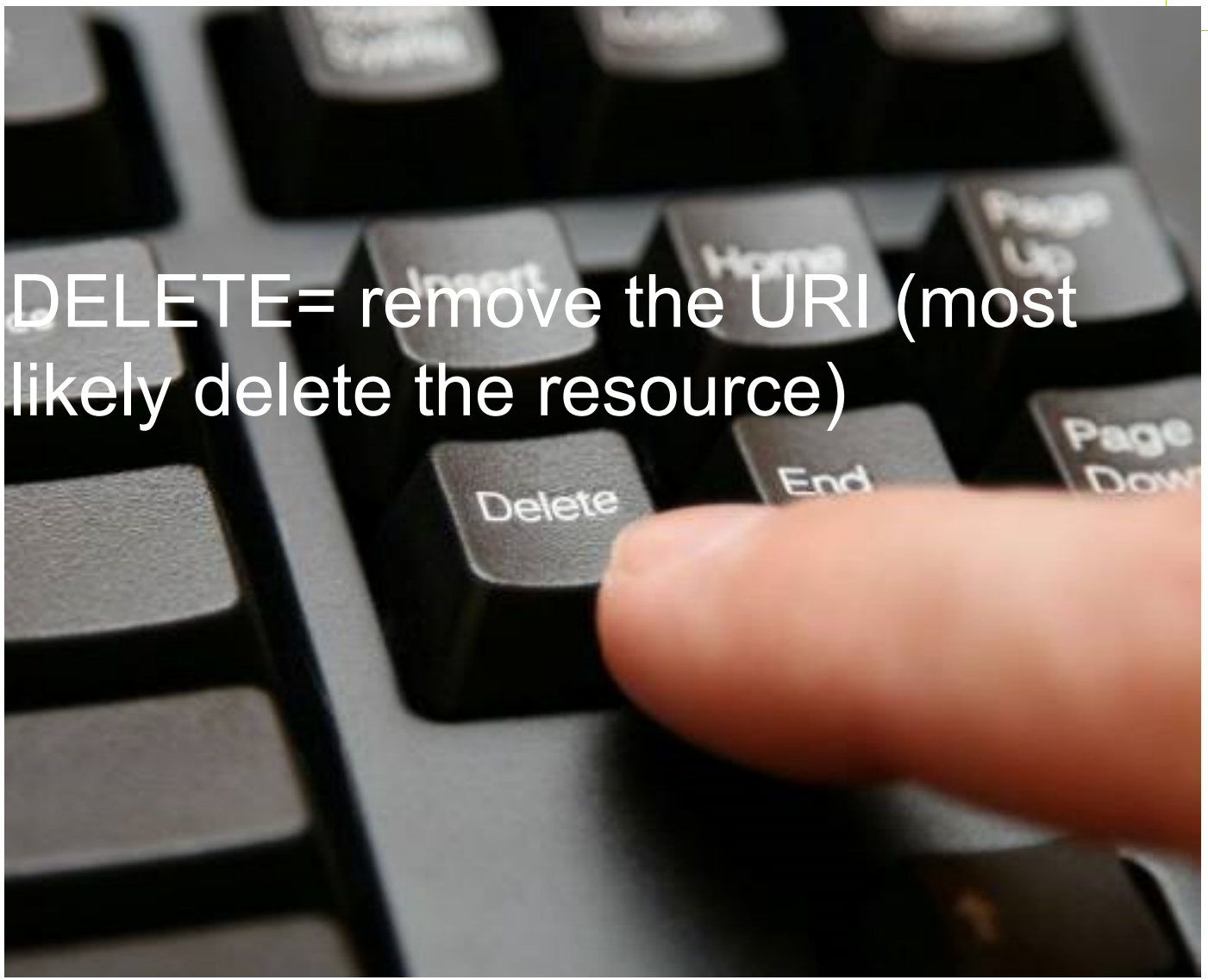
PUT, POST

HTTP POST creates a resource
HTTP PUT updates a resource

POST http://localhost/books/
Content: {title, authors[], …}
Creates a new book with given properties

PUT http://localhost/books/isbn-111
Content: {isbn, title, authors[], …}
Updates book identified by isbn-111 with submitted properties

DELETE


DELETE= remove the URI (most likely delete the resource)
Delete
End

DELETE

Removes the resource identified by the URI

DELETE http://localhost/books/ISBN-0011

Delete book identified by ISBN-0011

VERBS

OPTIONS –
The currently available verbs / requirements for communication

Representations

How data is represented or returned to the client for presentation.

Two main formats:

JavaScript Object Notation (JSON)

XML

It is common to have multiple representations of the same data

Representations

XML

```
<COURSE>
<ID>CS2650</ID>
<NAME>Distributed Multimedia Software</NAME>
</COURSE>
```

JSON

```
{course
{id: CS2650}
{name: Distributed Multimedia Sofware}
}
```

The REST Constraints

Client-Server architectural style

Stateless Server

- session state is kept on the client

Cacheable

- data is labelled as cacheable or non-cacheable

The REST Constraints

The Uniform interface is

- “The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components”

- for manipulating resources via URIs through a generic set of methods. A uniform interface reduces coupling compared to a services specific interface specification

The REST Constraints

The Uniform interface is composed of four constraints:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages – meta data & control data
- hypermedia as the engine of application state - “the hypertext constraint”, i.e. links in the data for navigation through the application. This is one of the most important constraints

The REST Constraints

Layered system

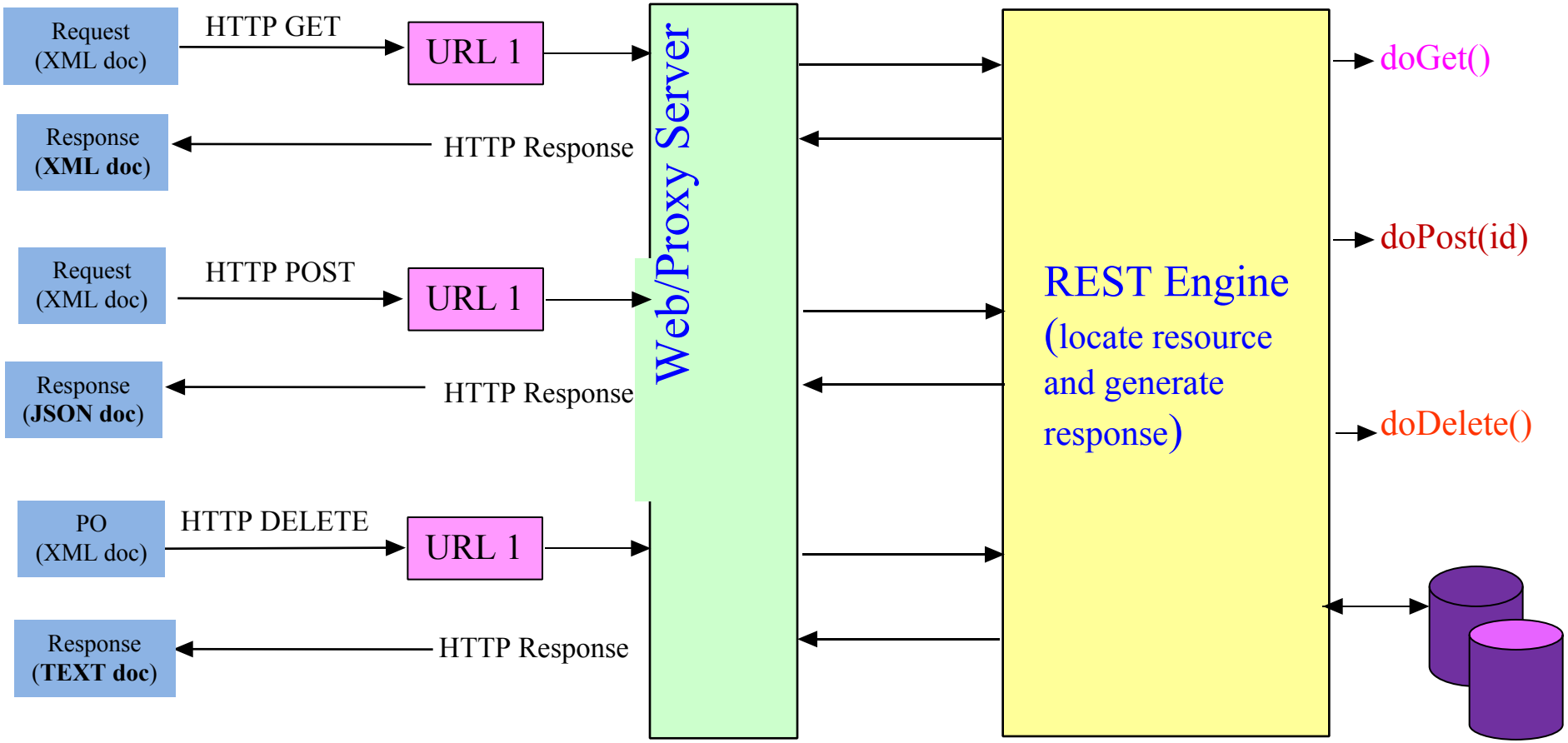
Code-On-Demand – “optional constraint”

- client functionality extended through download of code or scripts

The constraints will likely have design trade-offs

- caching may improve performance but decrease reliability if stale data is served
- a stateless server can increase information in each message & you now need to rely on the client to maintain state

Architecture Style



Web API using REST principles

1. **Define the resources**
2. Design the resource representations
3. HTTP Protocol considerations
   - response status codes
   - caching strategy
   - encodings
4. Security

SOFT resources

Candidate resources are:

- family-trees
- family-tree
- people
- person
- relations
- relation
- children
- child
- mother
- father
- brother
- sister
- spouse
- siblings

Web API using REST principles

1. Define the resources
2. **Design the resource representations**
3. HTTP Protocol considerations
   - response status codes
   - caching strategy
   - encodings
4. Security

Resource representations

Lets elicit some of them by working through example scenarios of request response messages across the uniform interface

Our initial URL will be

http://soft.example.org/family-trees

The server will create the hierarchy underneath this URL so client does not need to know the structure as it will be given it by the server in the representations – this avoids coupling between client & server over the hierarchy structure

Resource representations

First off create a family-tree with the request:

```
POST /family-trees  HTTP 1.1
Host: soft.example.org
Content-Type: application/xml
Content-Length: …

<?xml version="1.0" encoding="UTF-8"?>
<familyTree xmlns="http://soft.example.org/family-trees" >
        <name>The Addams Family Tree</name>
        <createdBy>Gomez Addams</createdBy>
</familyTree>
```

Response:

```
201 Created
Location: http://soft.example.org/family-trees/0008
Content-Type: application/xml
Content-Length: …

<?xml version="1.0" encoding="UTF-8"?>
<familyTree xmlns="http://soft.example.org/family-trees" xml:base="http://soft.example.org/family-trees/0008/">
        <name>The Addams Family Tree</name>
        <createdBy>Gomez Addams</createdBy>
        <link rel="edit" type="application/xml" href="" title="FamilyTree"/>
        <link rel="related" type="application/xml" href="people" title="People"/>
</familyTree>
```

Resource representations

In the previous response the link element with the edit attribute is for editing, the href is relative to xml:base (the link element is borrowed from the atom:link element)

From the previous response using the link element with the title attribute “People”, its relative href & the xml:base attribute we can now create an initial person with the request:

```
POST /family-trees/0008/people  HTTP 1.1
Host: soft.example.org
Content-Type: application/xml
Content-Length: …

<?xml version="1.0" encoding="UTF-8"?>
<person xmlns="http://soft.example.org/family-trees">
    <firstName>Morticia</firstName>
    <lastName>Addams</lastName>
    <birth>19800101</birth>
    <sex>F</sex>
</person>
```

Resource representations

Response:

with link elements for editing Morticia and Morticia's relations

```
201 Created
Location: http://soft.example.org/family-trees/0008/people/1234
Content-Type: application/xml
Content-Length: …

<?xml version="1.0" encoding="UTF-8"?>
<person xmlns="http://soft.example.org/family-trees"
    xml:base "http://soft.example.org/family-trees/0008/people/1234/">
    <firstName>Morticia</firstName>
    <lastName>Addams</lastName>
    <birth>19800101</birth>
    <sex>F</sex>
    <link rel="edit" type="application/xml" href="" title="Person"/>
    <link rel="related" type="application/xml" href="relations" title="Relations"/>
</person>
```

Resource representations

After creating two extra people who are Morticia's mother and brother, add them as relations of Moritica:

```
POST  /family-trees/0008/people/1234/relations  HTTP 1.1
Host: soft.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: …

role=mother&person=http://soft.example.org/family-trees/0008/people/2344

201 Created
Location: http://soft.example.org/family-trees/0008/people/1234/relations/mother
Content-Length: 0


POST  /family-trees/0008/people/1234/relations  HTTP 1.1
Host: soft.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: …

role=brother&person=http://soft.example.org/family-trees/0008/people/5588

201 Created
Location: http://soft.example.org/family-trees/0008/people/1234/relations/brothers/5588
Content-Length: 0
```

Resource representations

If we GET Morticia's relations using the edit link from the initial Post response using:

```
GET  /family-trees/0008/people/1234/relations  HTTP 1.1
Host: soft.example.org
```

Response:

```
200 OK
Location: http://soft.example.org/family-trees/0008/people/1234/relations
Content-Type: application/xml
Content-Length: …

<relations xmlns="http://soft.example.org/family-trees"
    xml:base "http://soft.example.org/family-trees/0008/people/1234/relations/">
    <link rel="edit" type="application/xml" href="" title="Relations"/>
    <relation role="mother" person="http://soft.example.org/family-trees/0008/people/2234">
        <link rel="edit" type="application/xml" href="mother" title="Mother"/>
    </relation>
    <relation role="brother" person="http://soft.example.org/family-trees/0008/people/5588">
        <link rel="edit" type="application/xml" href="brothers/5588" title="Brother"/>
    </relation>
</relations>
```

Resource representations

Lets modify Morticia's mother with a PUT request:

```
PUT  /family-trees/0008/people/1234/relations/mother  HTTP 1.1
Host: soft.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: …

role=mother&person=http://soft.example.org/family-trees/0008/people/5556
```

Response:

```
200 OK
Location: http://soft.example.org/family-trees/0008/people/1234/relations/mother
Content-Type: application/xml
Content-Length: …

<?xml version="1.0" encoding="UTF-8"?>
<relation  xmlns="http://soft.example.org/family-trees"
    xml:base "http://soft.example.org/family-trees/0008/people/1234/relations/mother"
    role="mother" person="http://soft.example.org/family-trees/0008/people/5556">
    <link rel="edit" type="application/xml" href="" title="Mother"/>
</relation>
```

Resource representations

Lets delete Morticia's brother with a DELETE request:

```
DELETE  /family-trees/0008/people/1234/relations/brothers/5588  HTTP 1.1
Host: soft.example.org
```

Response:

```
200 OK
```

Web API using REST principles

1. Define the resources
2. Design the resource representations
3. **HTTP Protocol considerations**
   - response status codes
   - caching strategy
   - encodings
4. Security

SOFT response status code

Specify the HTTP method response codes from SOFT

POST

- on success will return *201 Created* and the *Location:* header will have the URI of the new resource
- failure code: *400 Bad Request*

GET

- on success will return *200 OK*
- failure codes: *400 Bad Request, 404 Not Found*

SOFT response status code

PUT

- on success will return *200 OK*
- failure codes: *400 Bad Request, 404 Not Found, 409 Conflict*

DELETE

- on success will return *200 OK*
- failure codes: *400 Bad Request, 404 Not Found, 410 Gone*

If security is applied and the user is not authenticated then the response is *401 Unauthorised*

Consider SOFT's Caching Strategy

Cache Topology

- Private Caches – in the Web browser
- Proxy Caches – proxy server for scalability

Cache Processing

- Server specifies expiration time using Cache-Control (HTTP/1.1) or Expires (HTTP/1.0) headers
- Revalidation using a "conditional GET", only gets a new copy of the document if it has changed , client can send either header:
  - If-Modified-Since, send date that client has as Last-Modified date, only resolution to the second
  - If-None-Match, send the ETag value that was sent from the server back to the server to see if the cache stale

Consider HTTPs encoding for software

Content-Encoding, encoding of the data by

- compression (gzip ...) , encryption (PGP…)

Transfer-Encoding, encoding to change the way the data is transferred:

- chunked, send the data in chunks , don’t specify a content length

Web API using REST principles

1. Define the resources
2. Design the resource representations
3. HTTP Protocol considerations
   - response status codes
   - caching strategy
   - encodings
4. **Security**

SOFT Security

Could use

- basic authentication, need to use HTTPS to encrypt the username password information in the header
- digest authentication, MD5 checksum of the username, password so these are not sent in clear text
- HMAC: Keyed-Hashing for Message Authentication, a hash of selected elements from the request with a secret key

Concluding REST

Think of the resources:

1. Any information can be a resource
2. Their data format (representations with URIs) and
3. Manipulation through a uniform interface

If REST constraints aren't needed then don't worry about them. An API that has *all* the REST constraints is a RESTful API.

WHAT'S BACKBONE?

Backbone.js gives structure to web applications by providing:

- models with key-value binding
- custom events
- collections with a rich API of enumerable functions
- views with declarative event handling
- RESTful JSON interface wrapper to connect you own API

WHAT'S BACKBONE?

When working on a web application that involves a lot of JavaScript, one of the first things you learn is to stop tying your data to the DOM. It's all too easy to create JavaScript applications that end up as tangled piles of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server.

For rich client-side applications, a more structured approach is often helpful.

WHAT'S BACKBONE?

With Backbone, you represent your data as Models, which can be created, validated, destroyed, and saved to the server.

Whenever a UI action causes an attribute of a model to change, the model triggers a *"change"* event; all the Views that display the model's state can be notified of the change, so that they are able to respond accordingly, re-rendering themselves with the new information.

In a finished Backbone app, you don't have to write the glue code that looks into the DOM to find an element with a specific *id*, and update the HTML manually — when the model changes, the views simply update themselves.

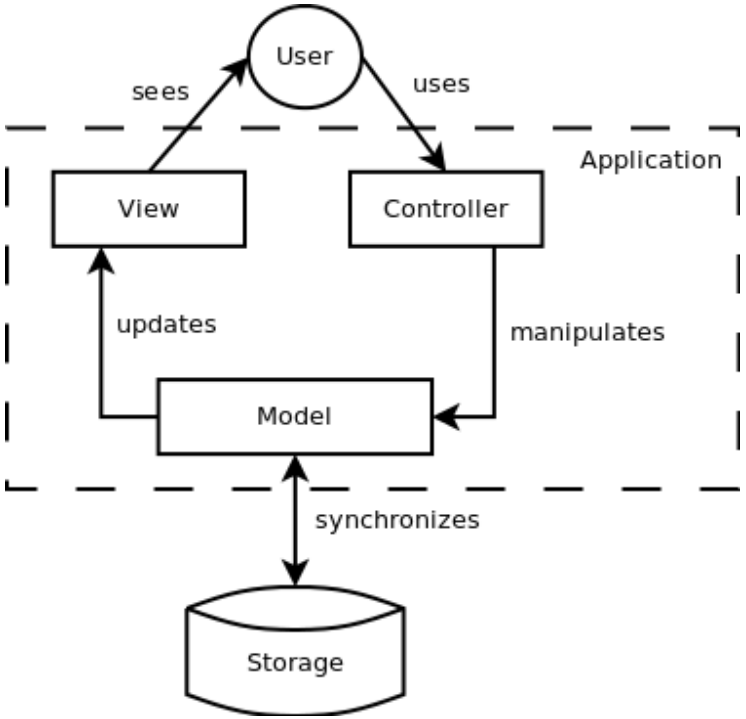
Now, you can see:

07 Backbone JavaScript Basics

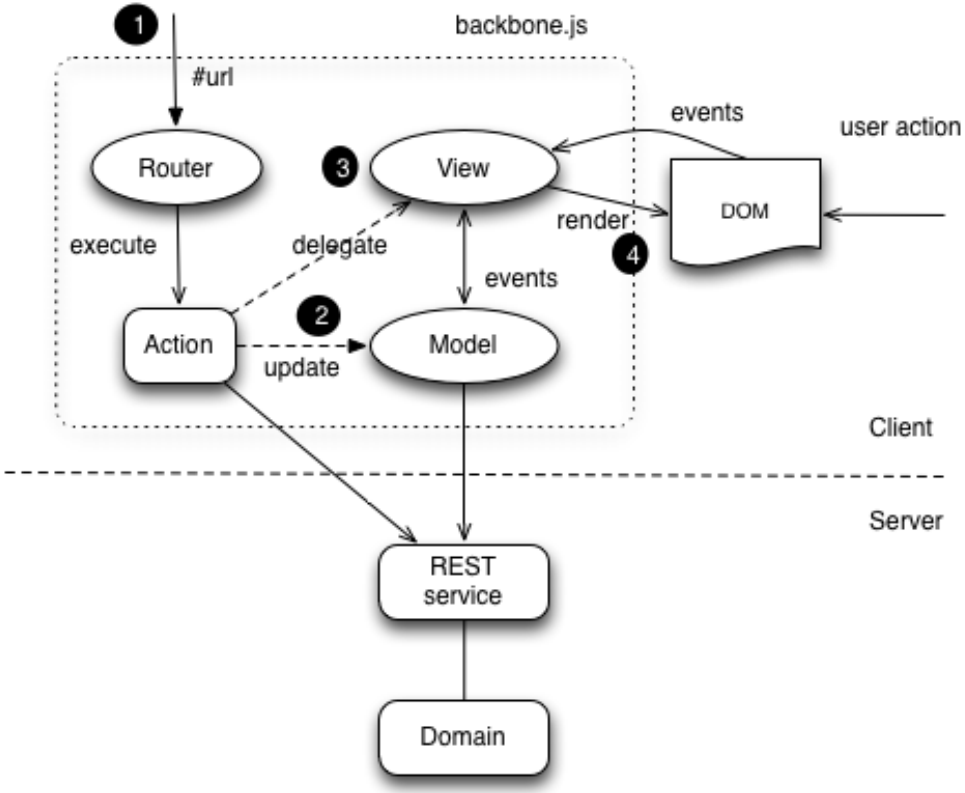
http://youtu.be/dR8IKvTmojQ

WHAT'S BACKBONE?

MVC:



Backbone:



Now, you can see:

08 Backbone Why MVC

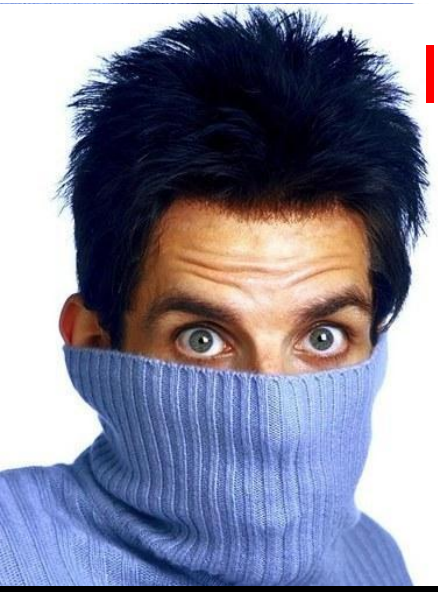
http://youtu.be/LScCT4gY-GU

MODELS

MVC: MODEL

Represent knowledge and data

Respond to requests about state



Isolated from views and controllers

Models contains data, along with your own domain-specific methods

Backbone.Models provides:

- a validation mechanism,
- a publisher/subscription mechanism,
- a basic set of functionality for managing changes
- methods to create, update, save, and destroy model data against a RESTful server

Backbone.Model

extend
constructor/initialize
url/urlRoot
defaults
get/set
validate
toJSON
save
State!
and more…

```
var Sidebar = Backbone.Model.extend({
    promptColor: function () {
        var cssColor = prompt("Please enter a CSS color:");
        this.set({ color: cssColor });
    }
});

window.sidebar = new Sidebar;

sidebar.on('change:color', function (model, color) {
    $('#sidebar').css({ background: color });
});

sidebar.set({ color: 'white' });

sidebar.promptColor();
```

Backbone.Model

extend

```
var Note = Backbone.Model.extend({

    initialize: function() {  },

    author: function() { },

    coordinates: function() {  },

    allowedToEdit: function(account) {
        return true;
    }

});

var PrivateNote = Note.extend({

    allowedToEdit: function(account) {
        return account.owns(this);
    }

});
```

Backbone.Model

extend
Overriding base methods

```
var Note = Backbone.Model.extend({
    set: function(attributes, options) {
        Backbone.Model.prototype.set.call(this, attributes, options);
    }
});
```

Backbone.Model

constructor/initialize

```
var book = new Note({
    title: "Shopping list",
    contents: "beer"
});
```

initialize

```
var Note = Backbone.Model.extend({
    initialize: function (args) {
        console.log(args);
    }
});
```

Backbone.Model

url/urlRoot

```
var url = "/[collection.url]/[id]";
var url2 = "/notes/12345";
var url3 = "/[urlRoot]/[id]";

var Note = Backbone.Model.extend({
    urlRoot: '/notes'
});
```

defaults

```
var Note = Backbone.Model.extend({
    defaults: {
        title: 'untitled',
        contents: ''
    }
});
```

Backbone.Model

get/set

```
// set
note.set({
    title: "March 20",
    content: "In his eyes she eclipse
});
note.set("title", "A Scandal in Bohem

//get
note.get("title");
```

Backbone.Model

validate

```
var Chapter = Backbone.Model.extend({
    validate: function (attrs) {
        if (attrs.end < attrs.start) {
            return "can't end before it starts";
        }
    }
});

var one = new Chapter({
    title: "Chapter One: The Beginning"
});

one.on("error", function (model, error) {
    alert(model.get("title") + " " + error);
});

one.set({
    start: 15,
    end: 10
});
```

Backbone.Model

toJSON

```
console.log(note.toJSON());
// {title: "A Scandal in Bohemia", contents: "beer" }
```

save

```
note.save();
note.save({title: 'server title'});
note.save({title: 'server title again'}, {
    error: function () { },
    success: function () { }
});
```

Backbone.Model

State!

```
note.on("change:title", function (model, title) {
    console.log(model.changedAttributes()); // {title: 'Wish list'}
    console.log(model.previousAttributes()); // {title: 'Shopping list', contents: 'beer'}
    var previousTitle = model.previous("title");
    console.log("Changed title from " + previousTitle + " to " + title);
    console.log(model.hasChanged()); // true
});

note.set({ title: "Wish list" });
```

Backbone.Model

More!

Model

- extend
- constructor / initialize
- get
- set
- escape
- has
- unset
- clear
- id
- idAttribute
- cid
- attributes
- changed
- defaults
- toJSON
- fetch
- save
- destroy
- validate
- isValid
- url
- urlRoot
- parse
- clone
- isNew
- change
- hasChanged
- changedAttributes
- previous
- previousAttributes

Now, you can see:

09 Backbone Models

http://youtu.be/0KPNlrJ92aI

Backbone.Model Events

on

```
object.on(event, callback, context);

book.on("change:title change:author", function () {

});
```

off

```
// Removes just the `onChange` callback.
object.off("change", onChange);

// Removes all "change" callbacks.
object.off("change");

// Removes the `onChange` callback for all events.
object.off(null, onChange);

// Removes all callbacks for `context` for all events.
object.off(null, null, context);

// Removes all callbacks on `object`.
object.off();
```

Backbone.Events define methods that are mixed in each Model and allows for triggering and listening of events

trigger

```
object.trigger(event, args);

var args = { id: 1, name: 'awesome name' };
object.trigger('added', args)
```

Backbone.Model Events

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },
  initialize: function(){
    console.log('This model has been initialized.');
    this.on('change', function(){
        console.log('- Values for this model have changed.');
    });
  }
});
var myTodo = new Todo();
myTodo.set('title', 'On each change of attribute values listener is triggered.');
console.log('Title has changed: ' + myTodo.get('title'));
```

Backbone Proxy Model

```
var book = {
    title: 'A Tale of Two Cities',
    author: 'Charles Dickens',
    genre: 'Historical',
};

var bookView = new BookView(book);

book.genre = 'Social Criticism';

bookView.genreChanged();
```

How does BookView know that the book's genre changed?

We could manually tell it...

but not without creating a tangled mess.

```
var book = new Backbone.Model({
    title: 'A Tale of Two Cities',
    author: 'Charles Dickens',
    genre: 'Historical',
});

var bookView = new BookView(book);

book.set({genre: 'Social Criticism'});
```

Let's wrap our object in a proxy.

Now we must use the proxy functions. (Until ECMAScript Harmony!)

Meanwhile, inside BookView...

```
book.on('change:genre', onChange);
```

Validation automatically occurs when the model is persisted using the .save() method or when .set() is called with the validate option set to true

```
var Todo = Backbone.Model.extend({
  defaults: {
    completed: false
  },

  validate: function(attribs){
    if(attribs.title === undefined){
        return 'Remember to set a title for your todo.';
    }
  }
});
var myTodo = new Todo();
myTodo.set('completed', true, {validate: true});
// logs: Remember to set a title for your todo
```

COLLECTIONS

Collections are ordered set of Models. The 'model' attribute indicates the kind of model contained

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});
var TodosCollection = Backbone.Collection.extend({
  model: Todo
});
var myTodo = new Todo({title:'Read the whole book', id: 2});
// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);
console.log("Collection size: " + todos.length); // Collection size: 1
```

COLLECTIONS: Retrieve & Change

A model instance can be retrieved using the .get() method. That method gets a model from a collection, specified by an id, a cid, or by passing in a model.

```
var todo = todoCollection.get(110);
```

If used the .at() method, then we can retrieve model by index

```
var todo = todoCollection.at(0);
```

Model instances can be added through .add(), and removed through .remove()

```
var a = new Todo({ title: 'Go to Jamaica.'}),
    b = new Todo({ title: 'Go to China.'}),
    c = new Todo({ title: 'Go to Disneyland.'});var
todos = new TodosCollection([a,b]);

todos.add(c);
todos.remove([a,b]);
```

COLLECTIONS: Retrieve & Change

Internally, Backbone.Collection contains an array of models enumerated by their id property, if the model instances happen to have one.

When collection.get(id) is called this array is checked for existence of the model instance with the corresponding id.

Sometimes you may also want to get a model based on its client id.

The client id is a property that Backbone automatically assigns to models that have not yet been saved.

You can get a model's client id from its .cid property.

```
var books = new Backbone.Collection([
    book1,
    book2,
    book3
]);

var booksView = new BooksView(books);

books.add(book4);
```

We do something similar for arrays.

```
books.on('add', onAdd);
```

Collections

model
url
create
add
get
where
fetch
reset
and more…

```
var TodoList = Backbone.Collection.extend({
    model: Todo,
    url: '/todos',
    completed: function() {
        return this.filter(function( todo ) {
            return todo.get('completed');
        });
    },
    remaining: function() {
        return this.without.apply( this, this.completed() );
    },
    nextOrder: function() {
        if ( !this.length ) {
            return 1;
        }
        return this.last().get('order') + 1;
    },
    comparator: function( todo ) {
        return todo.get('order');
    }
});
```

Collections

model

```
var Notes = Backbone.Collection.extend({
    model: Note
});
```

url

```
var NotesSimple = Backbone.Collection.extend({
    url: '/notes'
});

// Or, something more sophisticated:

var Notes = Backbone.Collection.extend({
    url: function () {
        return this.document.url() + '/notes';
    }
});
```

Collections

create

```
var milkNote = notes.create({ title: 'Get milk!' });
```

add

```
var notes = new Notes();

notes.add(note);
notes.add(
    { title: 'Do the washing!' },
    { title: 'write a presentation' }
);
```

Collections

get

```
console.log(notes.get(1).toJSON());
```

where

```
notes.where({ title: "A Scandal in Bohemia" });
```

fetch

```
notes.fetch();
notes.fetch({page: 2});
```

reset

```
<script>
    notes.reset(@Model.Notes.ToJson())
</script>
```

Now, you can see:

10 Backbone Collections

http://youtu.be/NQND0Z5rlUc

COLLECTIONS: Underscore Methods

Backbone proxies to Underscore.js to provide 28 iteration functions on Backbone.Collection.

```
books.each(function(book) {
  book.publish();
});
var titles = books.map(function(book) {
  return book.get('title');
});
var publishedBooks = books.filter(function(book) {
  return book.get('published') === true;
});
var alphabetical = books.sortBy(function(book) {
  return book.author.get('name').toLowerCase();
});
```

Collections

More!

Collection

- extend
- model
- constructor / initialize
- models
- toJSON
- **Underscore Methods (28)**
- add
- remove
- get
- getByCid
- at
- push
- pop
- unshift
- shift
- length
- comparator
- sort
- pluck
- where
- url
- parse
- fetch
- reset
- create

Views

Backbone's Views determine how data is displayed in the screen. They are usually subscribed to model events, so when the model changes the view is updated

```
var TodoView = Backbone.View.extend({
  tagName:  'li',
  events: {
    'dblclick label': 'edit'
  },

  initialize: function() {
    this.listenTo(this.model, 'change', this.render);
  },

  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    this.input = this.$('.edit');
    return this;
  },

  edit: function() {
  }
});
```

All views have a DOM element at all times (the el property), whether they've already been inserted into the page or not.

If you want to create a new element for your view, set any combination of the following view's properties:

- tagName,id and className
- a new element will be created for you by the framework and a reference to it will be available at the el property
- if nothing is specified tagName defaults to div.

Views: DOM Element

```
var ItemView = Backbone.View.extend({
  tagName: 'li'
});
var BodyView = Backbone.View.extend({
  el: 'body'
});
var item = new ItemView();
var body = new BodyView();
alert(item.el + ' ' + body.el);
```

```
var book = {
    title: 'A Tale of Two Cities',
    author: 'Charles Dickens',
    genre: 'Historical',
};

…pass book to BookView…

book.genre = 'Social Criticism';
```

Meanwhile inside BookView…

```
$scope.book = book;
$scope.$watch('book', function() {
        console.log('changed!');
    }, true);
```

Views

el/tagName/$el
render
remove
events

```
var NoteRow = Backbone.View.extend({
    tagName: "li",
    className: "note-row",
    events: {
        "click .icon":          "open",
        "click .button.edit":   "openEditDialog",
        "click .button.delete": "destroy"
    },

    render: function () {
        return this;
    }

});
```

Views

el/tagName/$el

```
var ItemView = Backbone.View.extend({
    tagName: 'li'
});

var BodyView = Backbone.View.extend({
    el: '#item-list'
});

var item = new ItemView();
var body = new BodyView();

body.$el.append(item.el);
```

Views

render

```
var NoteRow = Backbone.View.extend({
    tagName: "li",
    className: "note-row",
    events: {
        "click .icon":          "open",
        "click .button.edit":   "openEditDialog",
        "click .button.delete": "destroy"
    },

    render: function () {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    }

});
```

remove

```
item.remove();
```

Views

events

```
events: {
    "click .icon":          "open",
    "click .button.edit":   "openEditDialog",
    "click .button.delete": "destroy"
},
```

Backbone Responding to and updating view

```
Name: <input type="text" class="name-in">
<h1>Hello <span class="name-out"></span></h1>
```

```
Backbone.View.extend({
    events: {
        'keyup .name-in': 'onNameChange'
    },
    onNameChange: function(event) {
        // TODO: Optimize
        var name = $(event.target).val();
        this.$('.name-out').text(name);
    }
});
```

ROUTING

Routers help manage application state and connect URLs to application events

This is achieved using hash-tags with URL fragments, or using the browser's pushState and History API

ROUTING

```
var Workspace = Backbone.Router.extend({

  routes: {
    'help':                 'help',    // #help
    'search/:query':        'search',  // #search/kiwis
    'search/:query/p:page': 'search'   // #search/kiwis/p7
  },

  help: function() {
  },

  search: function(query, page) {
  }
});
```

After initial application bootstrap call Backbone.history.start(), or Backbone.history.start({pushState: true}) to route the initial URL

Router

routes
route
navigate
history

```
var Workspace = Backbone.Router.extend({

    routes: {
        "help":                 "help",    // #help
        "search/:query":        "search",  // #search/kiwis
        "search/:query/p:page": "search"   // #search/kiwis/p7
    },

    help: function() {
    },

    search: function(query, page) {
    }

});
```

Router

routes

```
routes: {
    "help":                 "help",    // #help
    "search/:query":        "search",  // #search/kiwis
    "search/:query/p:page": "search"   // #search/kiwis/p7
},
```

route

```
initialize: function(options) {

    // Matches #page/10, passing "10"
    this.route("page/:number", "page", function(number) {
    });

    // Matches /117-a/b/c/open, passing "117-a/b/c" to this.open
    this.route(/^(.*?)\/open$/, "open");

},

open: function(args) {

}
```

Router

navigate

```
        openPage: function(pageNumber) {
            this.document.pages.at(pageNumber).open();
            this.navigate("page/" + pageNumber);
        }
    };

// Or ...

    app.navigate("help/troubleshooting", {trigger: true});

// Or ...

    app.navigate("help/troubleshooting", {trigger: true, replace: true});
```

Now, you can see:

11 Backbone.js and the Server

http://youtu.be/4eZjPjkQMeA

Collection.fetch is used to retrieve a set of Models from the server. Model.fetch will retrieve just one Model data

Updates are performed individually through Model.save

Collection.create can be used to create a Model instance, add it to a Collection instance, and send it to the server.

```
var todos = new TodosCollection();
todos.fetch();
var todo2 = todos.get(2);
todo2.set('title', 'go fishing');
todo2.save();
// sends HTTP PUT to /todos/2
todos.create({title: 'Try out code samples'});
// sends HTTP POST to /todos and adds to collection
```

When `save()` is called on a model that was fetched from the server, it constructs a URL by appending the model's id to the collection's URL and sends an HTTP PUT to the server.
If the model is a new instance that was created in the browser then an HTTP POST is sent to the collection's URL.

```
var todos = new TodosCollection();
todos.fetch();
var todo2 = todos.get(2);
todo2.set('title', 'go fishing');
todo2.save();
// sends HTTP PUT to /todos/2
todos.create({title: 'Try out code samples'});
// sends HTTP POST to /todos and adds to collection
```

Model.destroy removes the instance from the containing collection, and from the server

```
todo2.destroy();
 // sends HTTP DELETE to /todos/2 and removes from collection
```

Backbone.Events provides methods for triggering and listening of events

This object is mixed into the other Backbone objects

```
//Bind a callback to an event, provides a context
book.on('change:title change:author', this.render, this);

//Remove a previously-bound callback function from an object
book.off('change', this.render);

//Triggers an event
this.model.trigger();

//Tell an object to listen to a particular event on an other
//object. Allows the object to keep track of the events, can
//be removed all at once later on
view.listenTo(model, 'change', view.render);

//Tell an object to stop listening to events
view.stopListening();
view.stopListening(model);
```

Backbone.js at this moment doesn't provide a method for disposing views

This may cause memory leaks if the same view is binded to different sources, like the DOM, or a Model

Add a dispose() method which at least do the following:

- call the view remove() method in order to remove the view from the DOM
- call the stopListening() method in order to remove any binding to any model
- if binded to events from a PubSub object, remove those bindings

Here's the complete list of built-in Backbone events, with arguments. You're also free to trigger your own events on Models, Collections and Views as you see fit.

"add" (model, collection, options) — when a model is added to a collection.

"remove" (model, collection, options) — when a model is removed from a collection.

"reset" (collection, options) — when the collection's entire contents have been replaced.

"sort" (collection, options) — when the collection has been re-sorted.

"change" (model, options) — when a model's attributes have changed.

"change:[attribute]" (model, value, options) — when a specific attribute has been updated.

"destroy" (model, collection, options) — when a model is destroyed.

"request" (model, xhr, options) — when a model (or collection) has started a request to the server.

"sync" (model, resp, options) — when a model (or collection) has been successfully synced with the server.

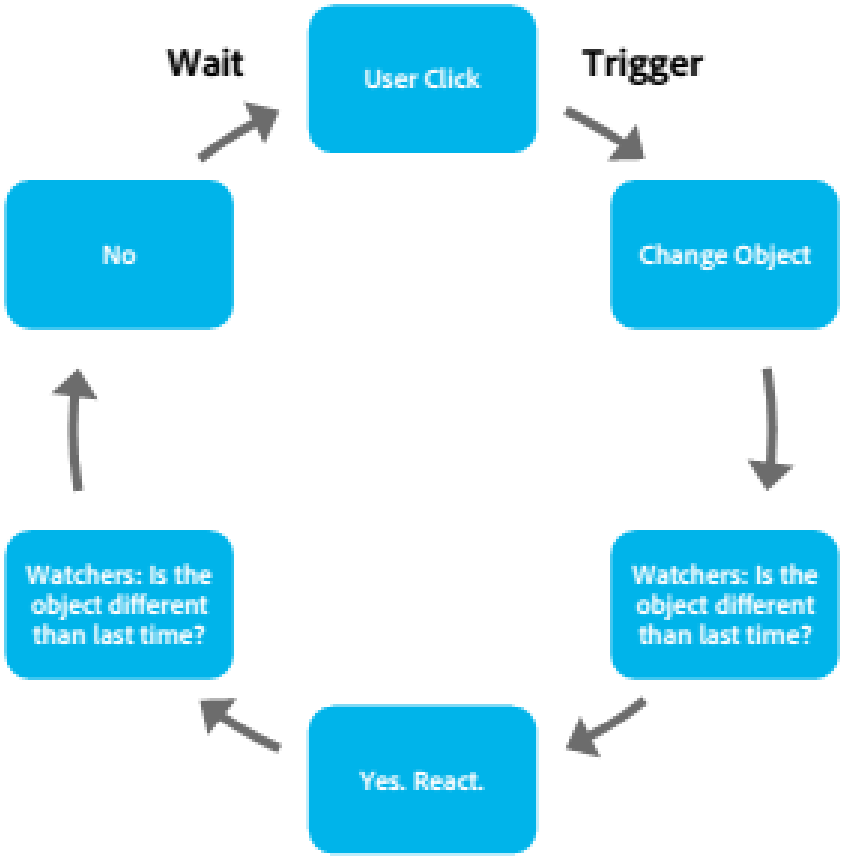
"error" (model, xhr, options) — when a model's save call fails on the server.

"invalid" (model, error, options) — when a model's validation fails on the client.

"route:[name]" (params) — Fired by the router when a specific route is matched.

"route" (router, route, params) — Fired by history (or router) when *any* route has been matched.

"all" — this special event fires for *any* triggered event, passing the event name as the first argument.



Digest Cycle

Triggered **automatically** on user interaction, http responses, etc.

Can be **manually** triggered.

Backbone Templates

```
<script id="users-template"
        type="text/x-handlebars-template">
    <ul>
        {{#users}}
        <li>Name: {{name}}, Email: {{email}}</li>
        {{/users}}
    </ul>
</script>
var data = {
    users: [
        { name: 'John', email: 'john@example.com' },
        { name: 'Jane', email: 'jane@example.com' }
    ]
};

var source = $('#users-template').html();
var template = Handlebars.compile(source);
var html = template(data);
$('body').html(html);
```

Backbone Templates

```
<body>
    <ul>
        <li>Name: John, Email: john@example.com</li>
        <li>Name: Jane, Email: jane@example.com</li>
    </ul>
</body>
```

Now, you can see:

12 Backbone js Client side

http://youtu.be/-P7rxaczmCM

Now, you can see:

13 Backbone js Server side

http://youtu.be/qDP37qF8KYs

Thank you!

Globant
we are ready