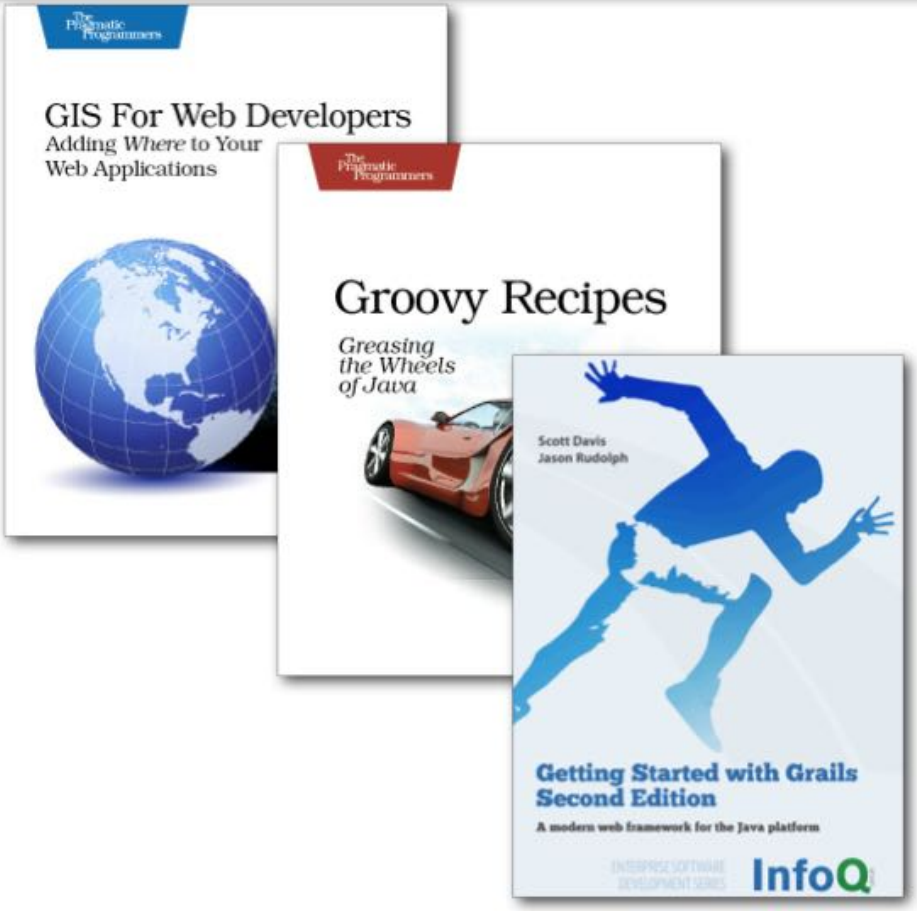
EXPLORING THE ARCHITECTURE OF THE MEAN STACK

MongoDB, ExpressJS, AngularJS, NodeJS

Scott Davis
Web: http://thirstyhead.com
Twitter: @scottdavis99
Slides: http://my.thirstyhead.com


ThirstyHead.com
training done right.

Scott Davis
@scottdavis99


The Pragmatic Programmers
GIS For Web Developers
Adding Where to Your Web Applications
The Pragmatic Programmers
Groovy Recipes
Greasing the Wheels of Java
Scott Davis
Jason Rudolph
Getting Started with Grails
Second Edition
A modern web framework for the Java platform
InfoQ


HTML

Mastering MEAN: Introducing the MEAN stack

Develop modern, full-stack, twenty-first-century web projects from end-to-end

Build a modern web application with MongoDB, Express, AngularJS, and Node.js in this six-part series by web development expert Scott Davis. This first installment includes a demo, sample code, and full instructions for creating a basic MEAN application. You'll also learn about Yeoman generators that you can use to bootstrap a new MEAN application quickly and easily.

→ View more content in this series | PDF (721 KB) | 0 Comments

Share:

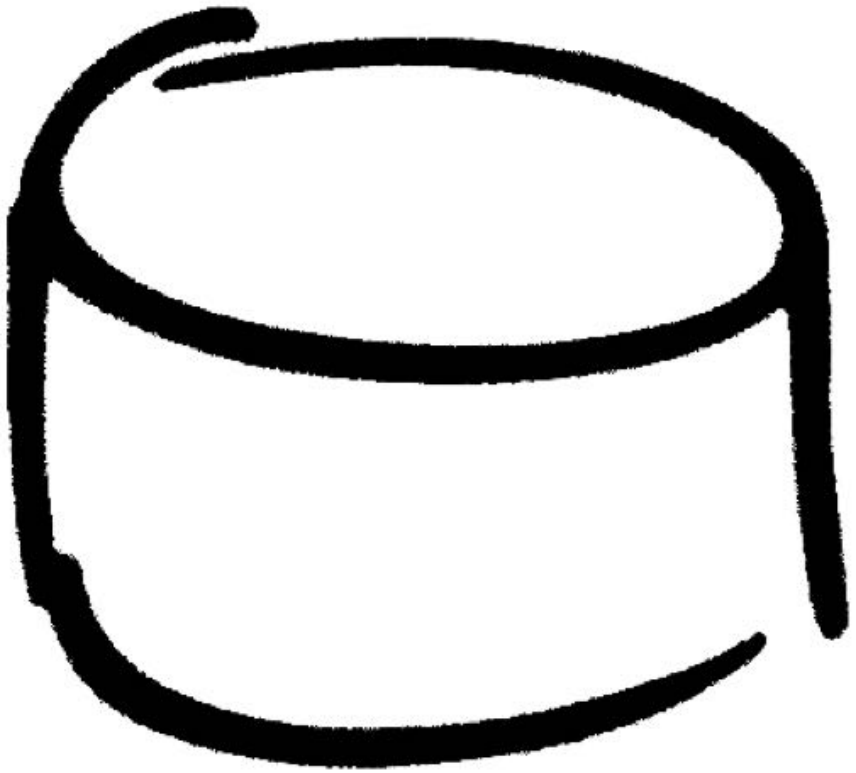
In his 2002 book, David Weinberger described the burgeoning web's content as a collection of *Small Pieces Loosely Joined*. That metaphor stuck with me, because it's easy to get tricked into thinking of the web as a monolithic technology stack. Actually, every website you visit is the product of a unique mixture of libraries, languages, and web frameworks.

MongoDB === Persistence

Linux ➡ NodeJS

Apache ➡ ExpressJS

MySQL ➡ MongoDB



Relational database

From Wikipedia, the free encyclopedia

A **relational database** is a digital database whose organization is based on the relational model of data, as proposed by E.F. Codd in 1970. This model organizes data into one or more tables (or "relations") of rows and columns, with a unique key for each row. Generally, each entity type described in a database has its own table, the rows representing instances of that entity and the columns representing the attribute values describing each instance. Because each row in a table has its own unique key, rows in other tables that are related to it can be linked to it by storing the original row's unique key as an attribute of the secondary row (where it is known as a "foreign key"). Codd showed that data relationships of arbitrary complexity can be represented using this simple set of concepts.

Prior to the advent of this model, databases were usually hierarchical, and each tended to be organized with a unique mix of indexes, chains, and pointers. The simplicity of the relational model led to its soon becoming the predominant type of database.

The various software systems used to maintain relational databases are known as Relational Database Management Systems (RDBMS).

Virtually all relational database systems use SQL (Structured Query Language) as the language for querying and maintaining the database.

```
select * from books;
```

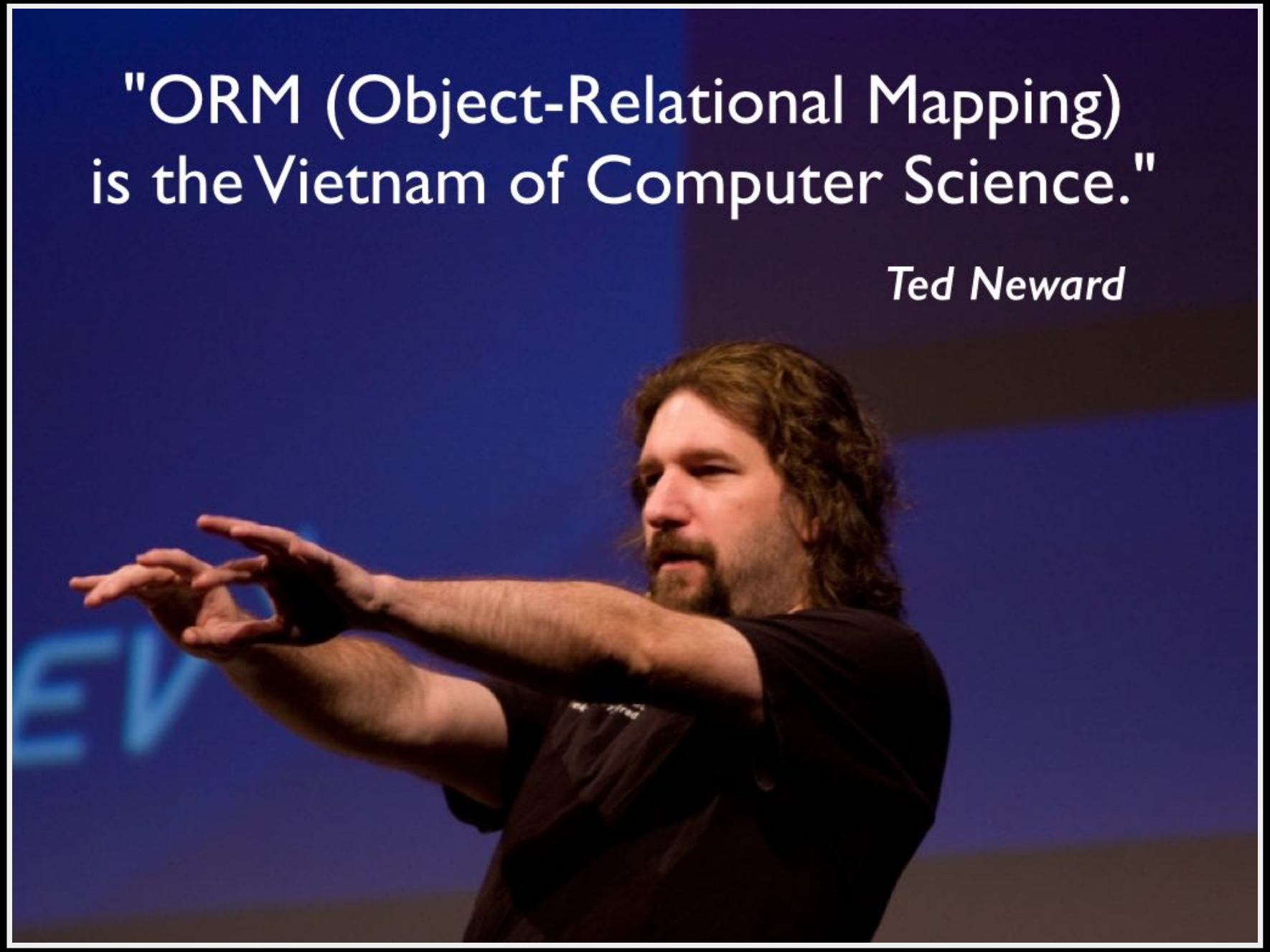
Object-relational mapping

From Wikipedia, the free encyclopedia

Object-relational mapping (**ORM**, **O/RM**, and **O/R mapping**) in computer science is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.

In object-oriented programming, data management tasks act on object-oriented (OO) objects that are almost always non-scalar values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

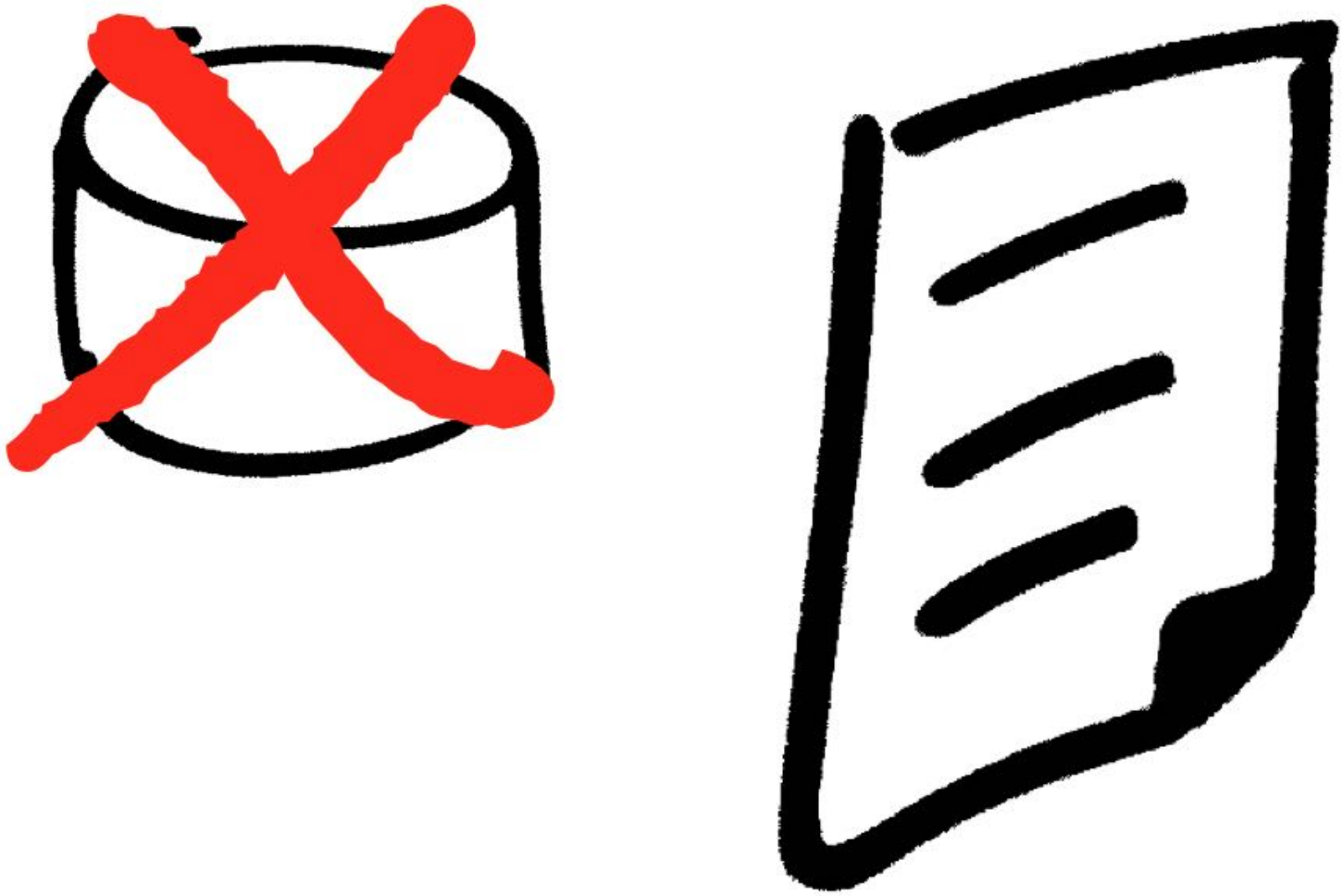
However, many popular database products such as structured query language database management systems (SQL DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping is used to implement the first approach.[1]


"ORM (Object-Relational Mapping)
is the Vietnam of Computer Science."
Ted Neward

eclipselink

OpenJPA


BRIDGE OUT
1/3 MILE AHEAD
LOCAL TRAFFIC ONLY



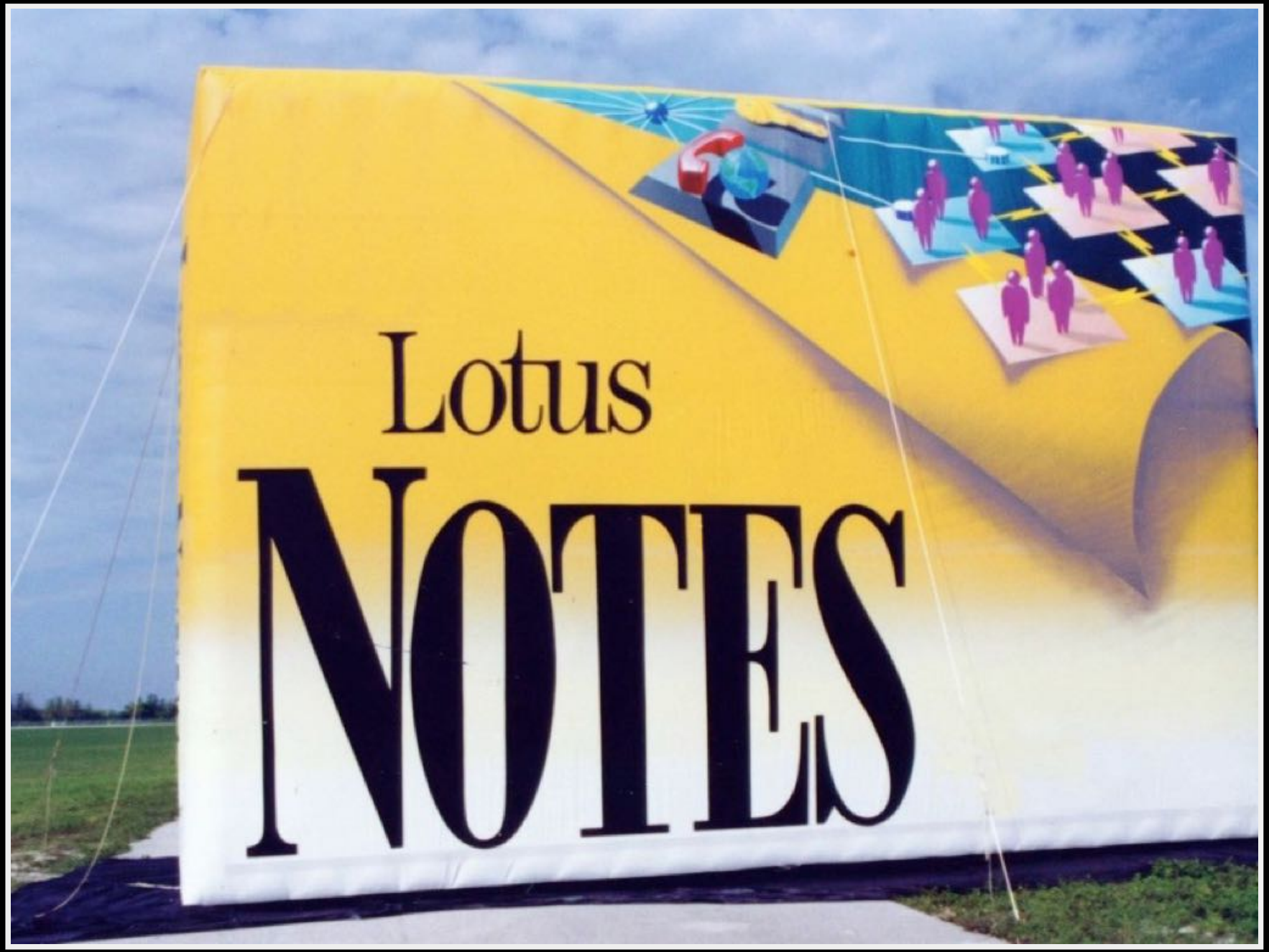
NoSQL

From Wikipedia, the free encyclopedia

A **NoSQL** (often interpreted as **Not only SQL**[1][2]) database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Motivations for this approach include simplicity of design, horizontal scaling, and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, graph, or document) differ from those used in relational databases, making some operations faster in NoSQL and others faster in relational databases. The particular suitability of a given NoSQL database depends on the problem it must solve.

NoSQL databases are increasingly used in big data and real-time web applications.[3] NoSQL systems are also called "Not only SQL" to emphasize that they may also support SQL-like query languages. Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, the lack of standardized interfaces, and huge investments in existing SQL.[4] Most NoSQL stores lack true ACID transactions, although a few recent systems, such as FairCom c-treeACE, Google Spanner (though technically a NewSQL database), FoundationDB and OrientDB have made them central to their designs.

[NoSQL]


Lotus
NOTES

Carlo Strozzi used the term *NoSQL* in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface.[5] Strozzi suggests that, as the current NoSQL movement "departs from the relational model altogether; it should therefore have been called more appropriately 'NoREL'",[6] referring to 'No Relational'.

Eric Evans reintroduced the term *NoSQL* in early 2009 when Johan Oskarsson of Last.fm organized an event to discuss open-source distributed databases.[7] The name attempted to label the emergence of an increasing number of non-relational, distributed data stores. Most of the early NoSQL systems did not attempt to provide atomicity, consistency, isolation and durability guarantees, contrary to the prevailing practice among relational database systems.[8]

Types of NoSQL databases [edit]

There have been various approaches to classify NoSQL databases, each with different categories and subcategories. Because of the variety of approaches and overlaps it is difficult to get and maintain an overview of non-relational databases. Nevertheless, a basic classification is based on data model. A few examples in each category are:

- **Column**: Accumulo, Cassandra, Druid, HBase, Vertica
- **Document**:Lotus Notes, Clusterpoint, Apache CouchDB, Couchbase, MarkLogic, MongoDB, OrientDB
- **Key-value**: CouchDB, Dynamo, FoundationDB, MemcacheDB, Redis, Riak, FairCom c-treeACE, Aerospike, OrientDB, MUMPS
- **Graph**: Allegro, Neo4J, InfiniteGraph, OrientDB, Virtuoso, Stardog
- **Multi-model**: OrientDB, FoundationDB, ArangoDB, Alchemy Database, CortexDB

Row/Column oriented

SQL

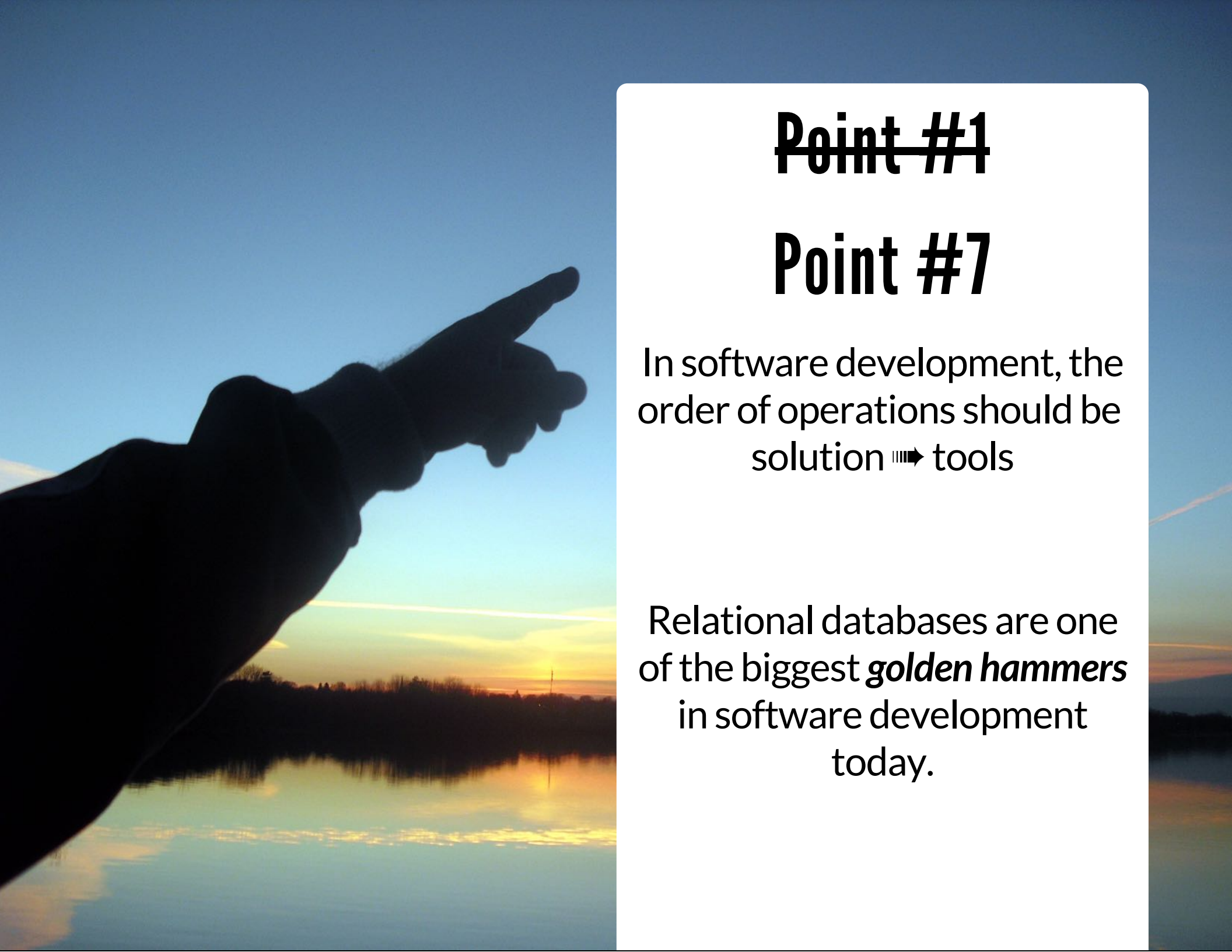
INSERT, SELECT, UPDATE, DELETE

Document oriented

JavaScript

POST, GET, PUT, DELETE

(REST)



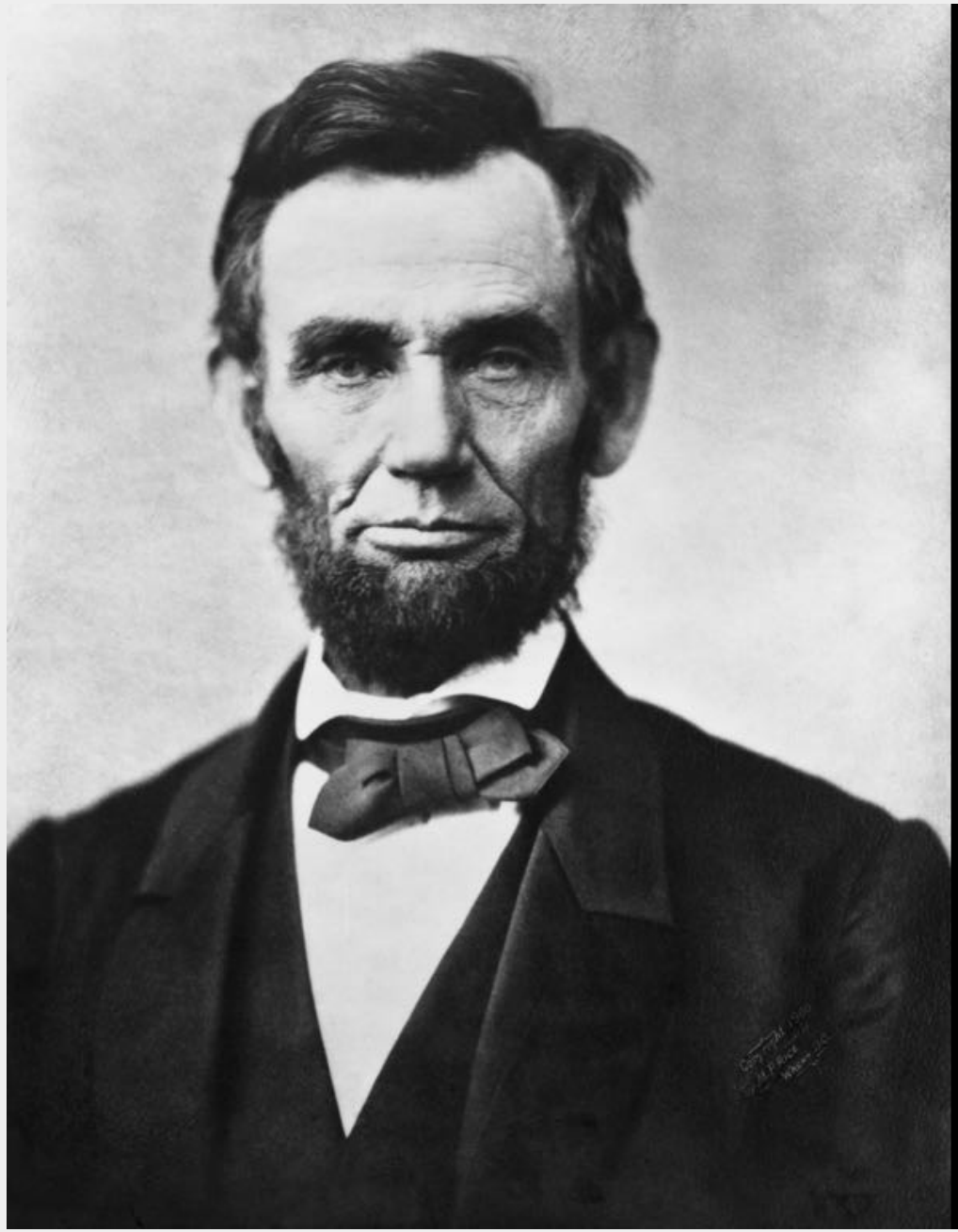
~~Point #1~~

Point #7

In software development, the order of operations should be solution ➠ tools

Relational databases are one of the biggest ***golden hammers*** in software development today.

Strong vs. Weak Typing



"If this is coffee, please bring me some tea;

If this is tea, please bring me some coffee..."

Abraham Lincoln

Coffee !== Tea

Java !== JavaScript

Strongly-typed (Java):

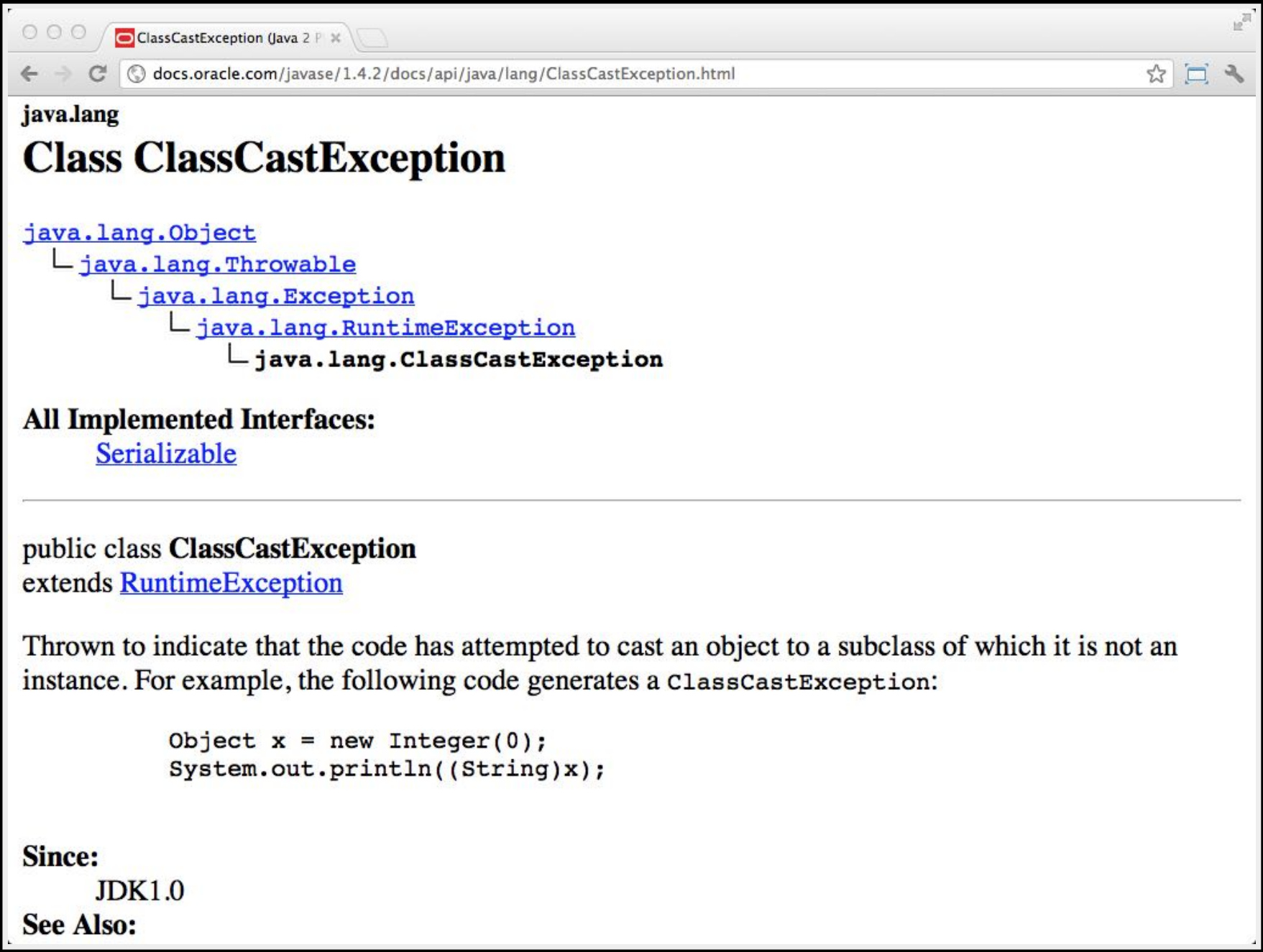
Strongly-typed (Java):

```
String name = "Scott";
```

```
Date now = new Date();
```

```
Person p = new Person(name, now);
```

```
Person p = "Scott";
// throws ClassCastException
```



java.lang

Class ClassCastException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ **java.lang.ClassCastException**

All Implemented Interfaces:

Serializable

---

public class **ClassCastException**
extends RuntimeException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a `ClassCastException`:

```
Object x = new Integer(0);
System.out.println((String)x);
```

Since:

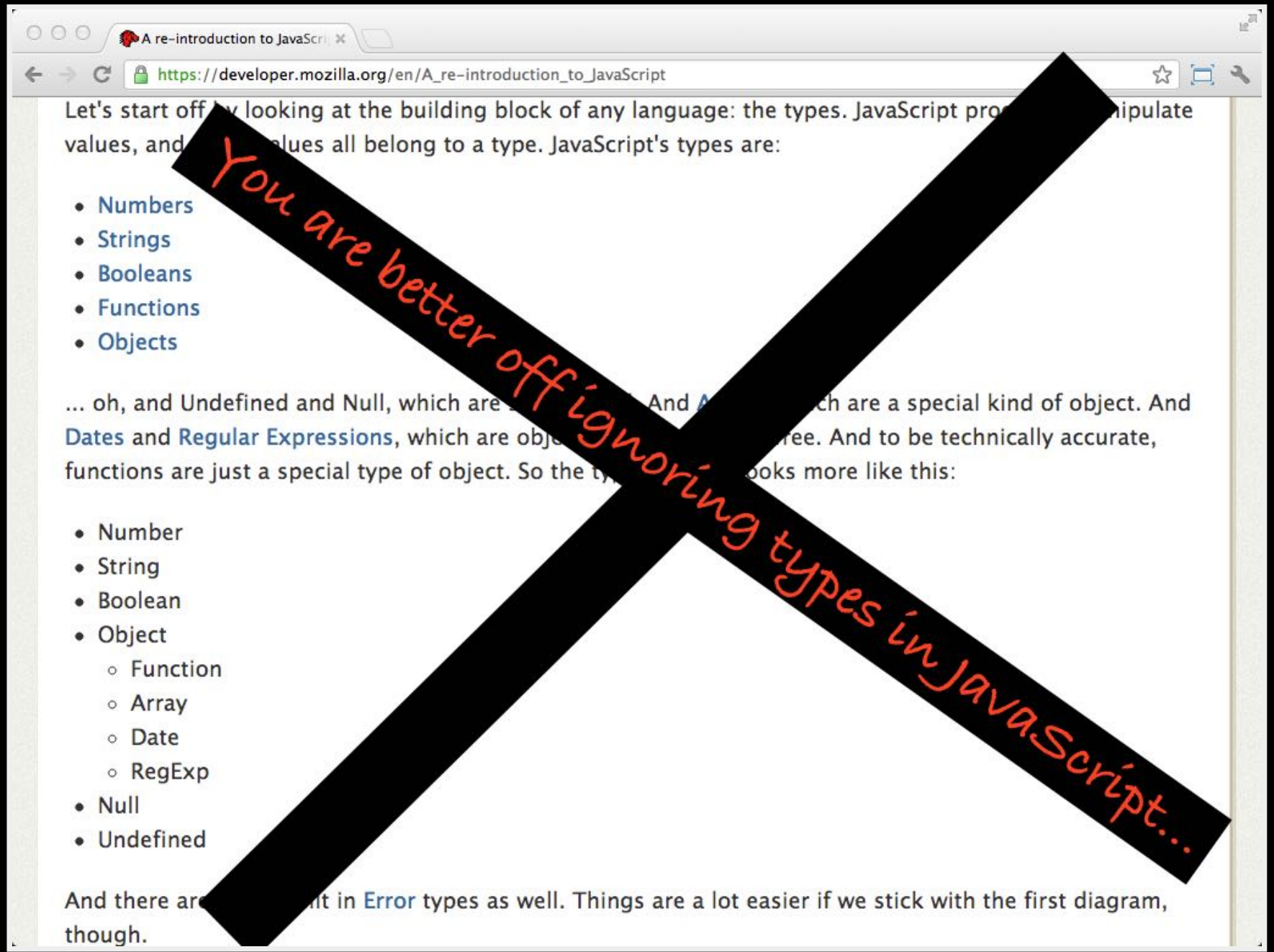
JDK1.0

See Also:

Weakly-typed (JavaScript):

```
var name = "Scott";
var now = new Date();
var p = new Person(name, now);
```

Notice I didn't say that JavaScript was "typeless"...


A re-introduction to JavaScri
https://developer.mozilla.org/en/A_re-introduction_to_JavaScript
Let's start off by looking at the building block of any language: the types. JavaScript pro ipulate
values, and lues all belong to a type. JavaScript's types are:
Numbers
Strings
Booleans
Functions
Objects
... oh, and Undefined and Null, which are And ch are a special kind of object. And
Dates and Regular Expressions, which are obj ee. And to be technically accurate,
functions are just a special type of object. So the t ks more like this:
Number
String
Boolean
Object
Function
Array
Date
RegExp
Null
Undefined
And there ar t in Error types as well. Things are a lot easier if we stick with the first diagram,
though.
You are better off ignoring types in JavaScript...

== vs. ===

(aka JavaScript “Truthiness”)

```
1  console.log(1 == 1)                                  //true
2  console.log("JavaScript" == "JavaScript");           //true
3
4  console.log(1 == true);                              //true
5  console.log(0 == false);                             //true
6  console.log("" == false);                            //true
7  console.log("0" == false);                           //true
8
```

In JavaScript:

== tests for "truthy" equality

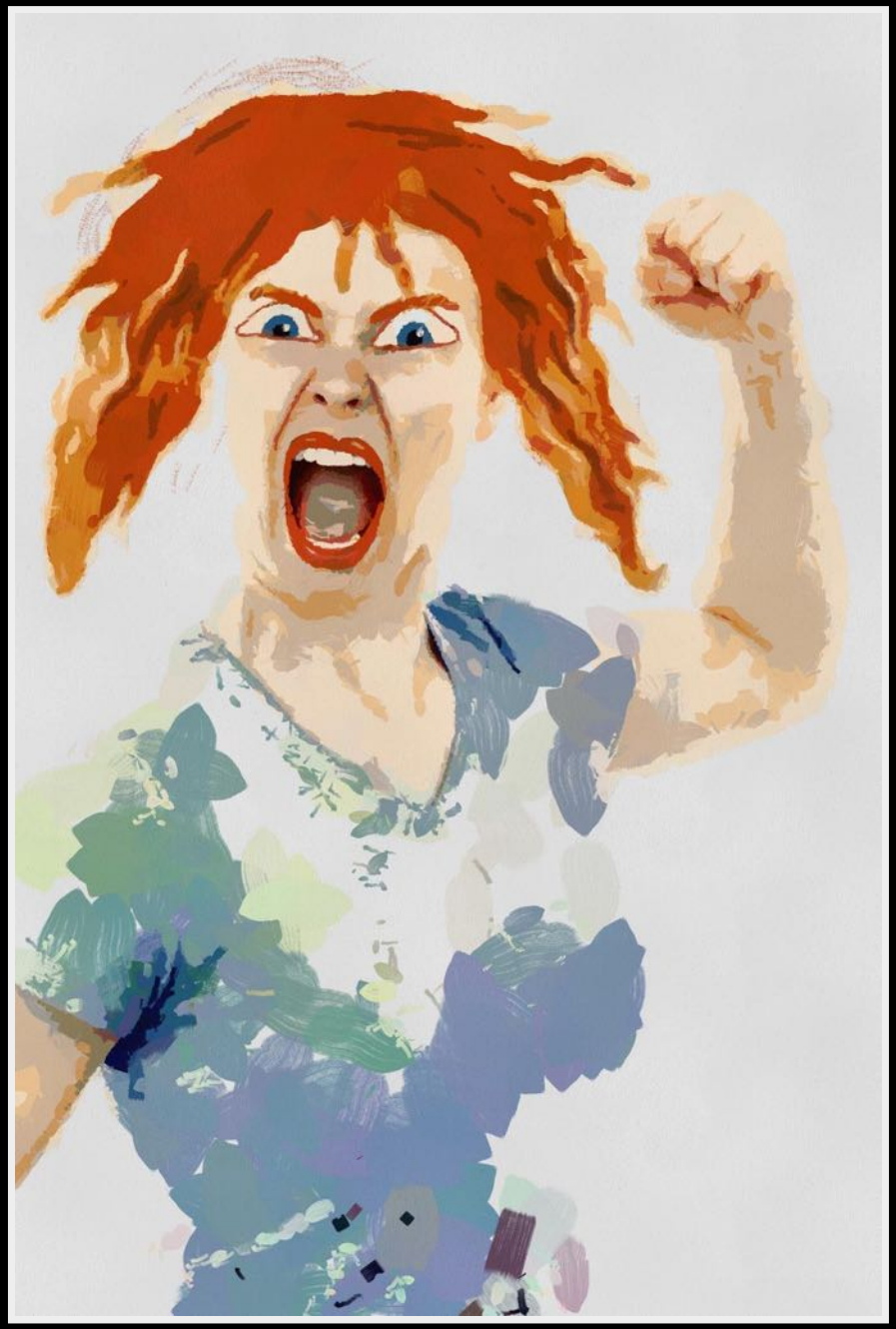
In other words, == does type coercion

```
1  console.log(1 == 1)                                  //true
2  console.log("JavaScript" == "JavaScript");           //true
3
4  console.log(1 == true);                              //true
5  console.log(0 == false);                             //true
6  console.log("" == false);                            //true
7  console.log("0" == false);                           //true
8
9  console.log("true" == true);                         //FALSE!
10 console.log(false == "false");                       //FALSE!
11
12 console.log(0 == "0");                               //true
13 console.log(0 == "");                                //true
14 console.log("" == "0");                              //FALSE!
15
```

In JavaScript:

== tests for "truthy" equality

In other words, == does type coercion



~~Misguided~~ JavaScript Hate

Yeah, uh, not all
JavaScript hate is misguided...

11.9.3 The Abstract Equality Comparison Algorithm

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is the same as Type(*y*), then
   a. If Type(*x*) is Undefined, return **true**.
   b. If Type(*x*) is Null, return **true**.
   c. If Type(*x*) is Number, then
      i. If *x* is **NaN**, return **false**.
      ii. If *y* is **NaN**, return **false**.
      iii. If *x* is the same Number value as *y*, return **true**.
      iv. If *x* is **+0** and *y* is **−0**, return **true**.
      v. If *x* is **−0** and *y* is **+0**, return **true**.
      vi. Return **false**.
   d. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
   e. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**. Otherwise, return **false**.
   f. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String,
   return the result of the comparison *x* == ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number,
   return the result of the comparison ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ToNumber(*y*).
8. If Type(*x*) is either String or Number and Type(*y*) is Object,
   return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String or Number,
   return the result of the comparison ToPrimitive(*x*) == *y*.
10. Return **false**.

In JavaScript:

=== tests for "strict" equality

In other words, NO type coercion

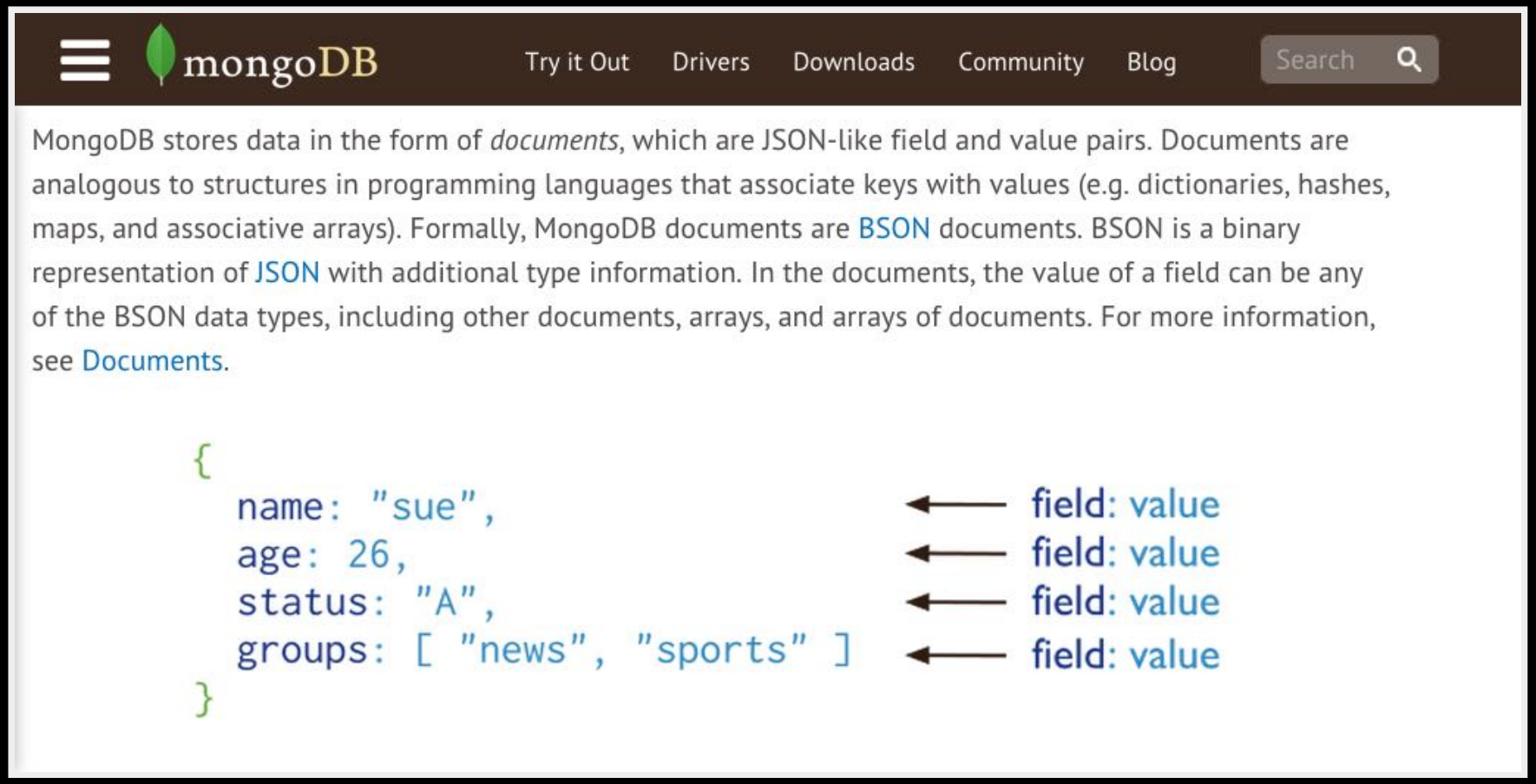
```
1  console.log(1 === 1)                                  //true
2  console.log("JavaScript" === "JavaScript");           //true
3
4  console.log(1 === true);                              //FALSE!
5  console.log(0 === false);                             //FALSE!
6  console.log("" === false);                            //FALSE!
7  console.log("0" === false);                           //FALSE!
8
9  console.log("true" === true);                         //FALSE!
10 console.log(false === "false");                       //FALSE!
11
12 console.log(0 === "0");                               //FALSE!
13 console.log(0 === "");                                //FALSE!
14 console.log("" === "0");                              //FALSE!
15
16 var lang = "JavaScript"
17 var anotherLang = new String("JavaScript")
18 console.log(lang === anotherLang);                     //FALSE!
```

Fascinating, but...

...what does strong vs. weak typing have to do with MongoDB?

1. MongoDB reduces language impedance mismatch

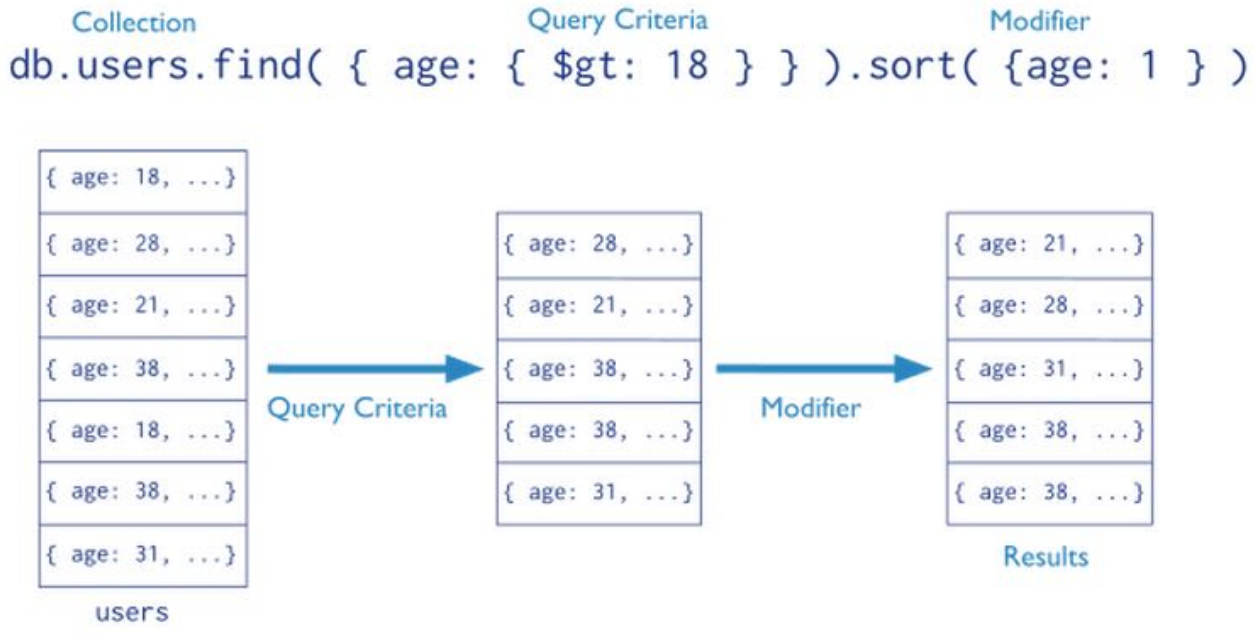
A JavaScript-based persistence solution is a perfect match for a JavaScript-based application.



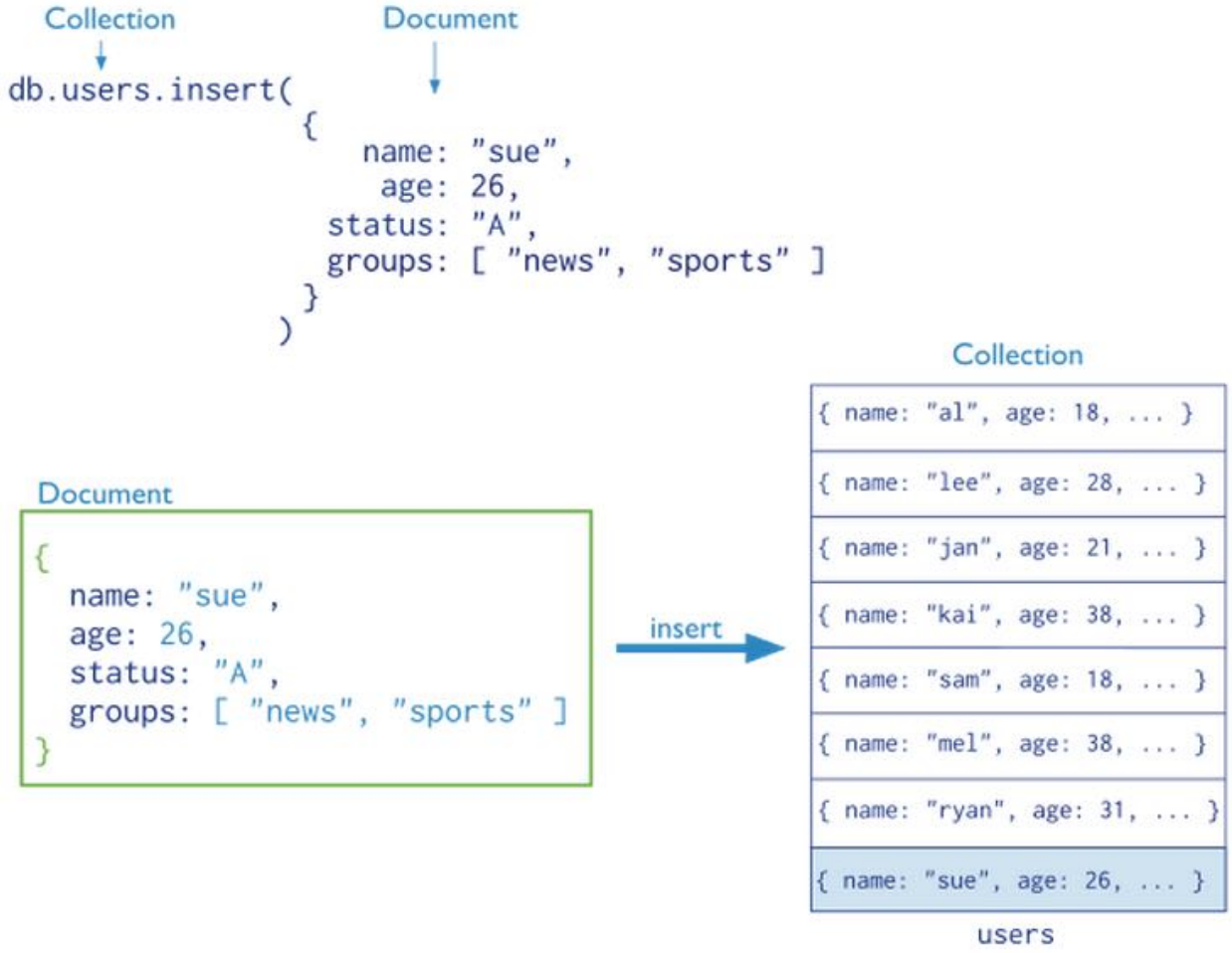
Query

In MongoDB a query targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips, and sort orders.

In the following diagram, the query process specifies a query criteria and a sort modifier:

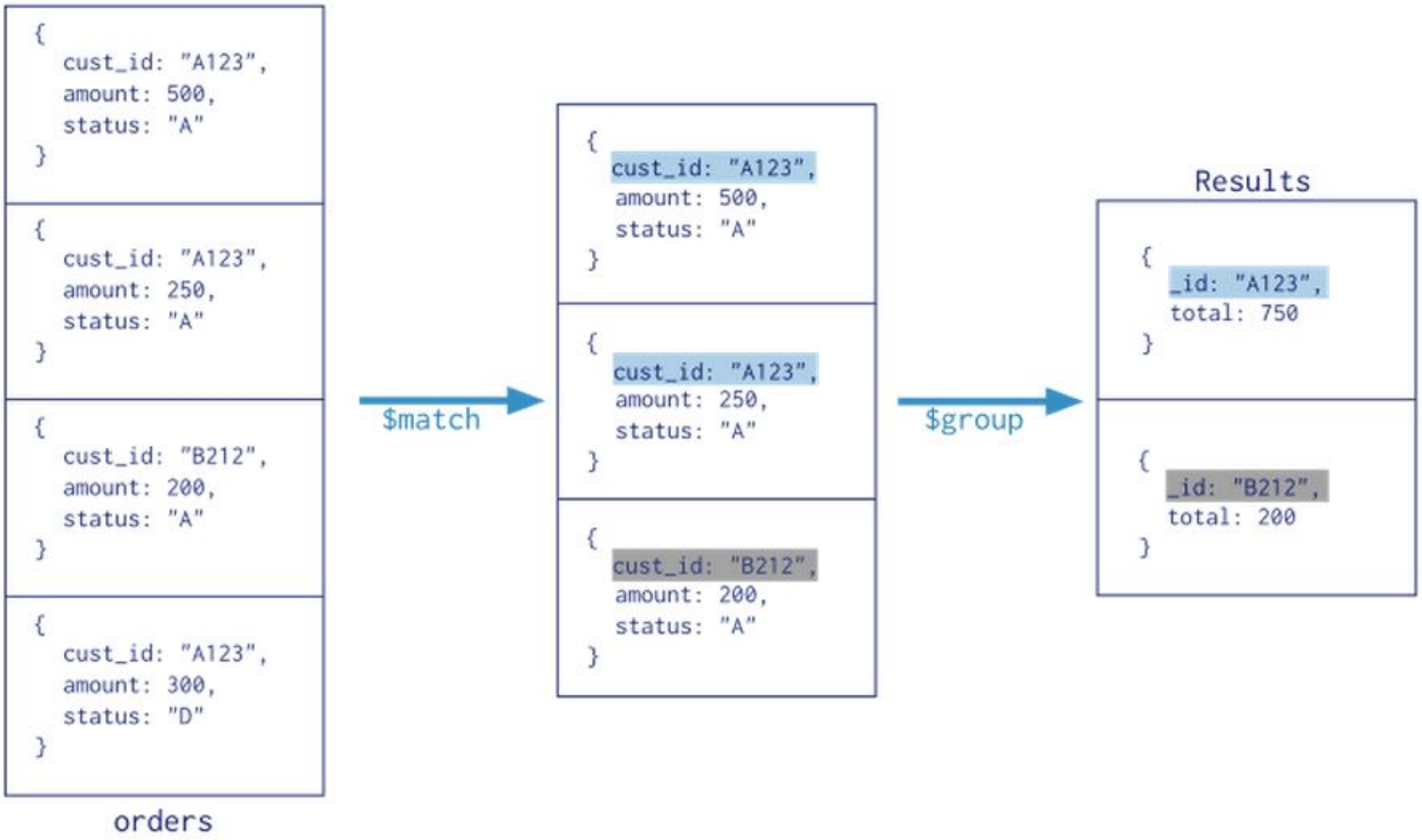


In the following diagram, the insert operation adds a new document to the **users** collection.



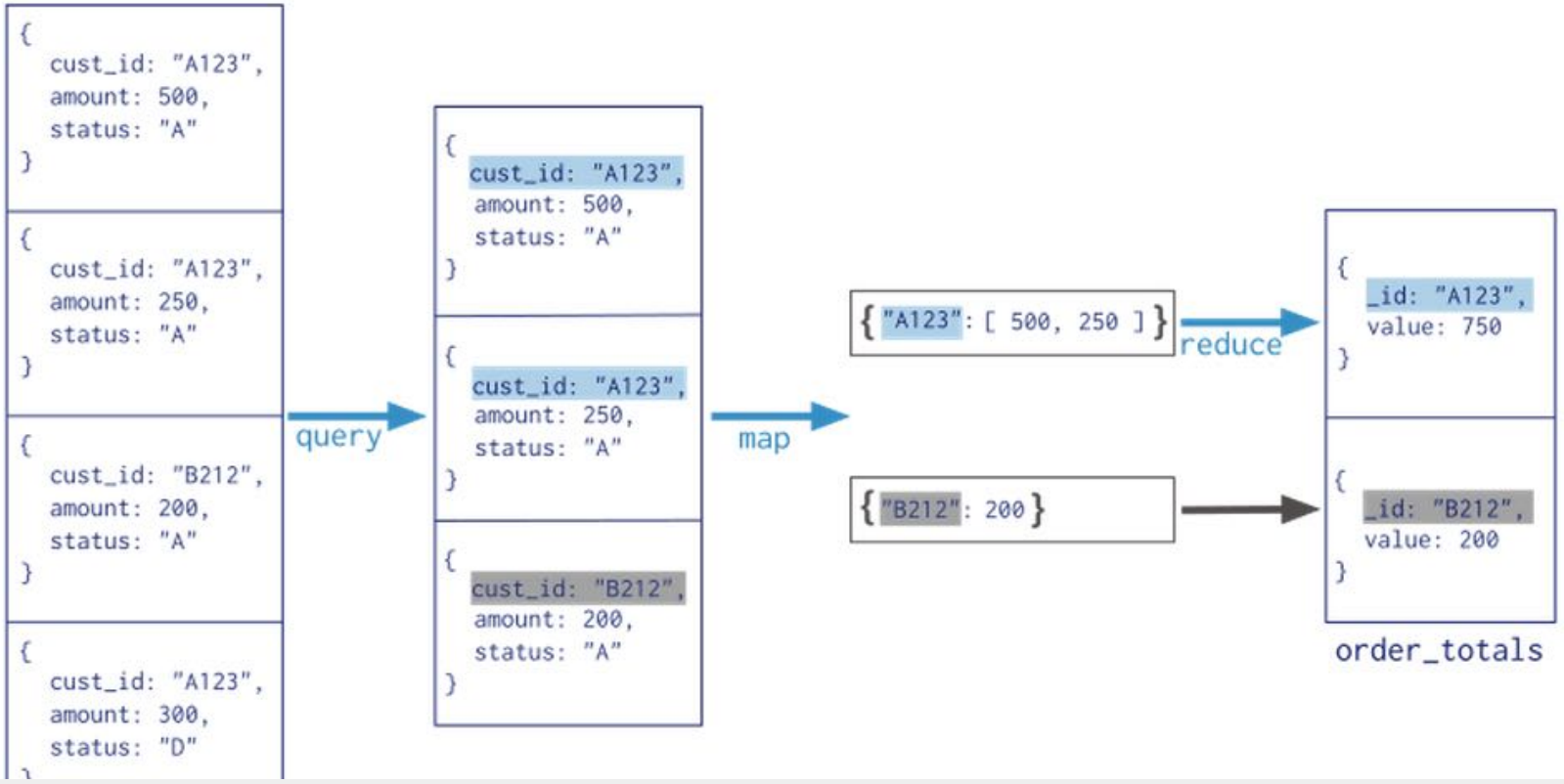
Collection

```
db.orders.aggregate( [
    $match stage ──►   { $match: { status: "A" } },
    $group stage ──►   { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                     ] )
```



mongoDB Try it Out Drivers Downloads Community Blog Search

```
db.orders.mapReduce(
        map      ──►  function() { emit( this.cust_id, this.amount ); },
        reduce   ──►  function(key, values) { return Array.sum( values ) },
                      {
        query    ──►    query: { status: "A" },
        output   ──►    out: "order_totals"
                      }
                    )
```



2. MongoDB reduces modeling impedance mismatch

MongoDB simply offers durable, persistent JSON.

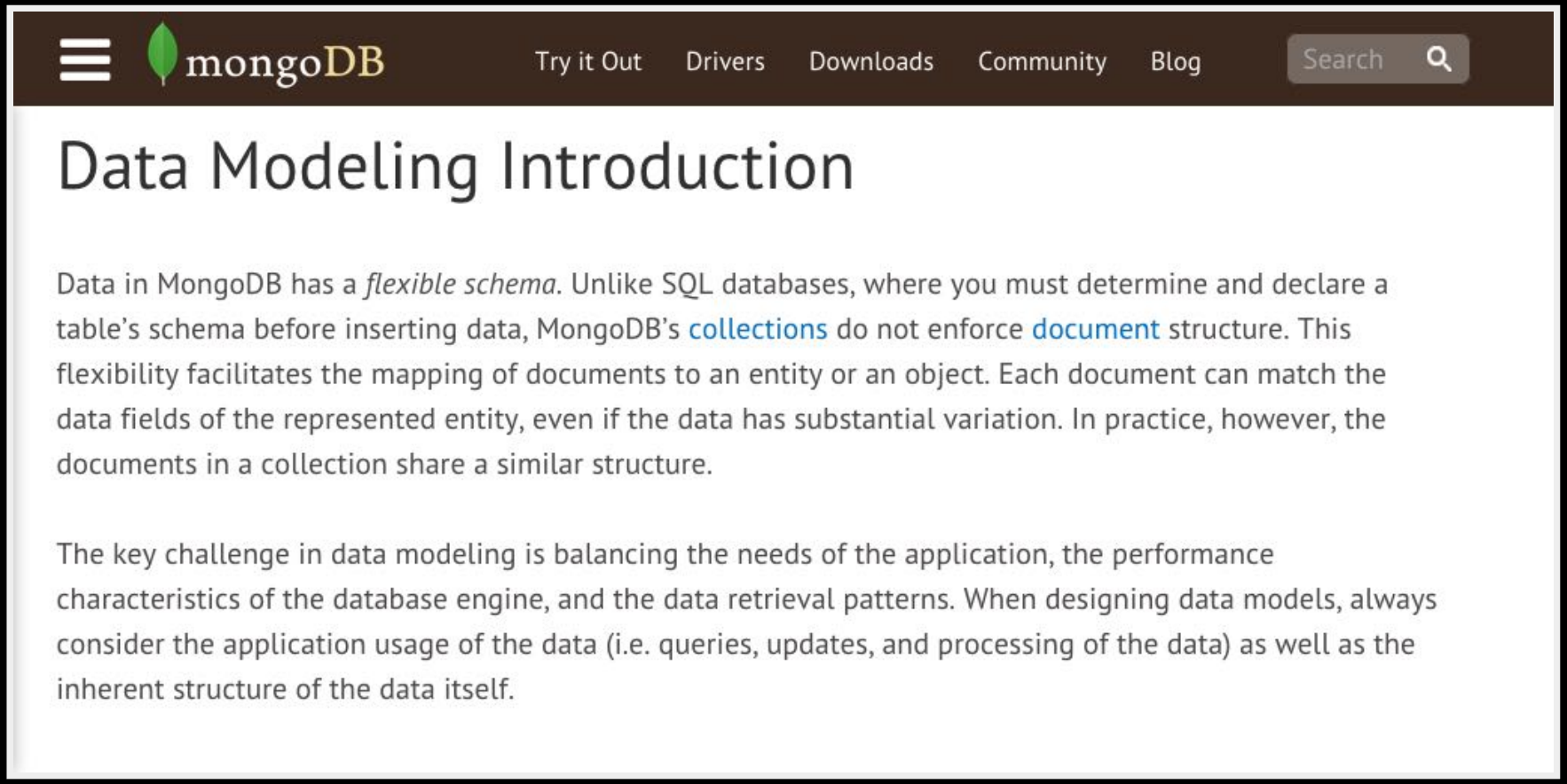
Object-relational impedance mismatch

From Wikipedia, the free encyclopedia

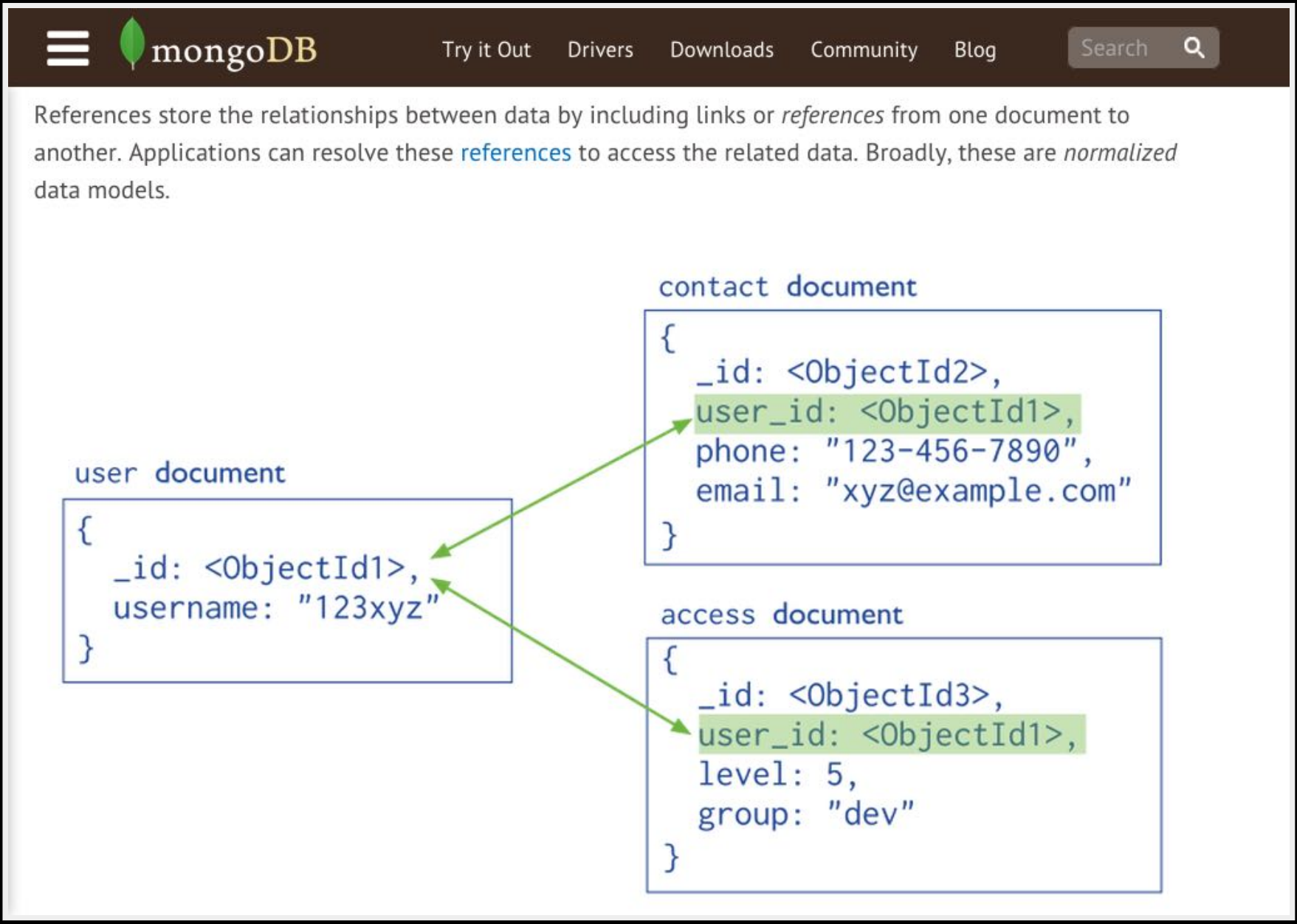
The **object-relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style; particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schema.

The term *object-relational impedance mismatch* is derived from the electrical engineering term *impedance matching*.

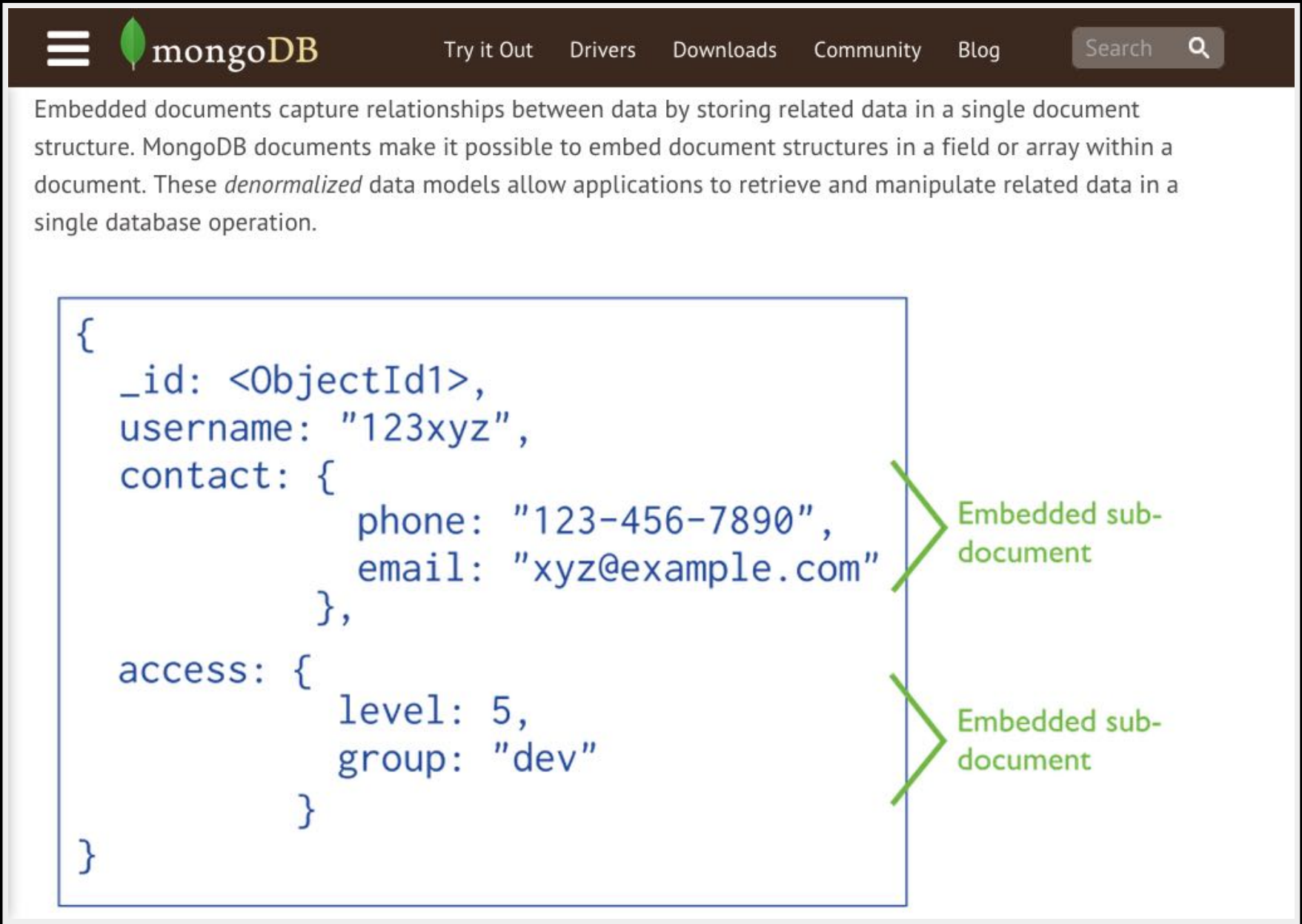
Schema-less persistence lends itself nicely
to an agile, emergent design.



MongoDB has primary keys but no joins in queries.



Embedded docs more naturally match domain-driven design.

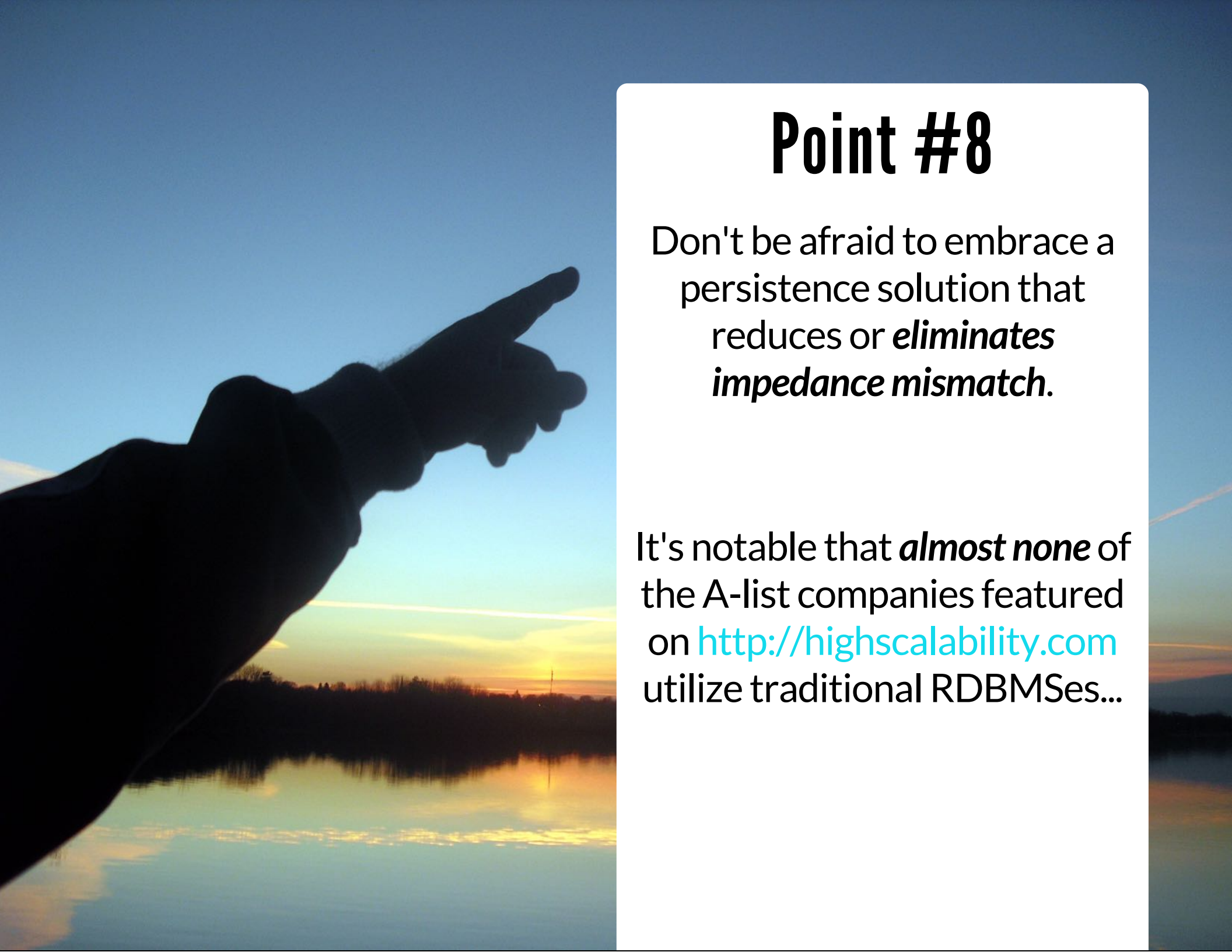


MongoDB modeling introduces new idioms:

- Since we don't have joins, it's OK to denormalize data.
- Since we have embedded sub-documents,
  1:1 relationships go away.
- Since we have embedded arrays,
  1:Few relationships are commonplace.

MongoDB still supports common RDBMS idioms:

- Indexing is still crucial to performance.
- Clustering / failover is available via replica sets.
- For large datasets, sharding distributes data across the cluster.



Point #8

Don't be afraid to embrace a persistence solution that reduces or ***eliminates impedance mismatch***.

It's notable that ***almost none*** of the A-list companies featured on http://highscalability.com utilize traditional RDBMSes...

http://mongoosejs.com/

Installation

First install node.js and mongodb. Then:

```
$ npm install mongoose
```

Overview

Connecting to MongoDB

First, we need to define a connection. If your app uses only one database, you should use `mongoose.connect`. If you need to create additional connections, use `mongoose.createConnection`.

Both `connect` and `createConnection` take a `mongodb://` URI, or the parameters `host, database, port, options`.

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my_database');
```

Once connected, the `open` event is fired on the `Connection` instance. If you're using `mongoose.connect`, the `Connection` is `mongoose.connection`. Otherwise, `mongoose.createConnection` return value is a `Connection`.

Defining a Model

Models are defined through the `Schema` interface.

```
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

var BlogPost = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , date      : Date
});
```

Aside from defining the structure of your documents and the types of data you're storing, a Schema handles the definition of:

- Validators (async and sync)
- Defaults
- Getters
- Setters
- Indexes
- Middleware
- Methods definition
- Statics definition
- Plugins

The following example shows some of these features:

```
var Comment = new Schema({
    name  :  { type: String, default: 'hahaha' }
  , age   :  { type: Number, min: 18, index: true }
  , bio   :  { type: String, match: /[a-z]/ }
  , date  :  { type: Date, default: Date.now }
  , buff  :  Buffer
});

// a setter
Comment.path('name').set(function (v) {
  return capitalize(v);
});

// middleware
Comment.pre('save', function (next) {
  notify(this.get('email'));
  next();
});
```

Take a look at the example in `examples/schema.js` for an end-to-end example of a typical setup.

Mongoose supports embedded subdocs and arrays of subdocs.

mongoose

- home
- FAQ
- plugins
- change log
- support
- fork
- guide
  - schemas
    - types
  - models
  - documents
    - sub docs
  - queries
  - validation
  - middleware
  - population
  - connections
  - plugins
  - contributing
  - migrating from 2.x
  - 3.6 release notes
  - 3.8 release notes
- API docs
- quick start

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

If you want to add additional keys later, use the Schema#add method.

Each key in our `blogSchema` defines a property in our documents which will be cast to its associated SchemaType. For example, we've defined a `title` which will be cast to the String SchemaType and `date` which will be cast to a `Date` SchemaType. Keys may also be assigned nested objects containing further key/type definitions *(e.g. the `meta` property above).*

Mongoose schemas support 1:M relationships

mongoose

- home
- FAQ
- plugins
- change log
- support
- fork
- guide
  - schemas
    - types
  - models
  - documents
    - sub docs
  - queries
  - validation
  - middleware
  - population
  - connections
  - plugins
  - contributing
  - migrating from 2.x

Arrays

Provide creation of arrays of SchemaTypes or Sub-Documents.

```
var ToySchema = new Schema({ name: String });
var ToyBox = new Schema({
  toys: [ToySchema],
  buffers: [Buffer],
  string:  [String],
  numbers: [Number]
  // ... etc
});
```

Note: specifying an empty array is equivalent to `Mixed`. The following all create arrays of `Mixed`:

```
var Empty1 = new Schema({ any: [] });
var Empty2 = new Schema({ any: Array });
var Empty3 = new Schema({ any: [Schema.Types.Mixed] });
var Empty4 = new Schema({ any: [{}] });
```

Even though MongoDB doesn't support joins, Mongoose supports 'em via populate.

mongoose

- home
- FAQ
- plugins
- change log
- support
- fork
- guide
  - schemas
    - types
  - models
  - documents
    - sub docs
  - queries
  - validation
  - middleware
  - population
  - connections
  - plugins
  - contributing
  - migrating from 2.x

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where population comes in.

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s). We may populate a single document, multiple documents, plain object, multiple plain objects, or all objects returned from a query. Let's look at some examples.

```
var mongoose = require('mongoose')
  , Schema = mongoose.Schema

var personSchema = Schema({
  _id     : Number,
  name    : String,
  age     : Number,
  stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

var storySchema = Schema({
  _creator : { type: Number, ref: 'Person' },
  title    : String,
  fans     : [{ type: Number, ref: 'Person' }]
});
```

mongoose

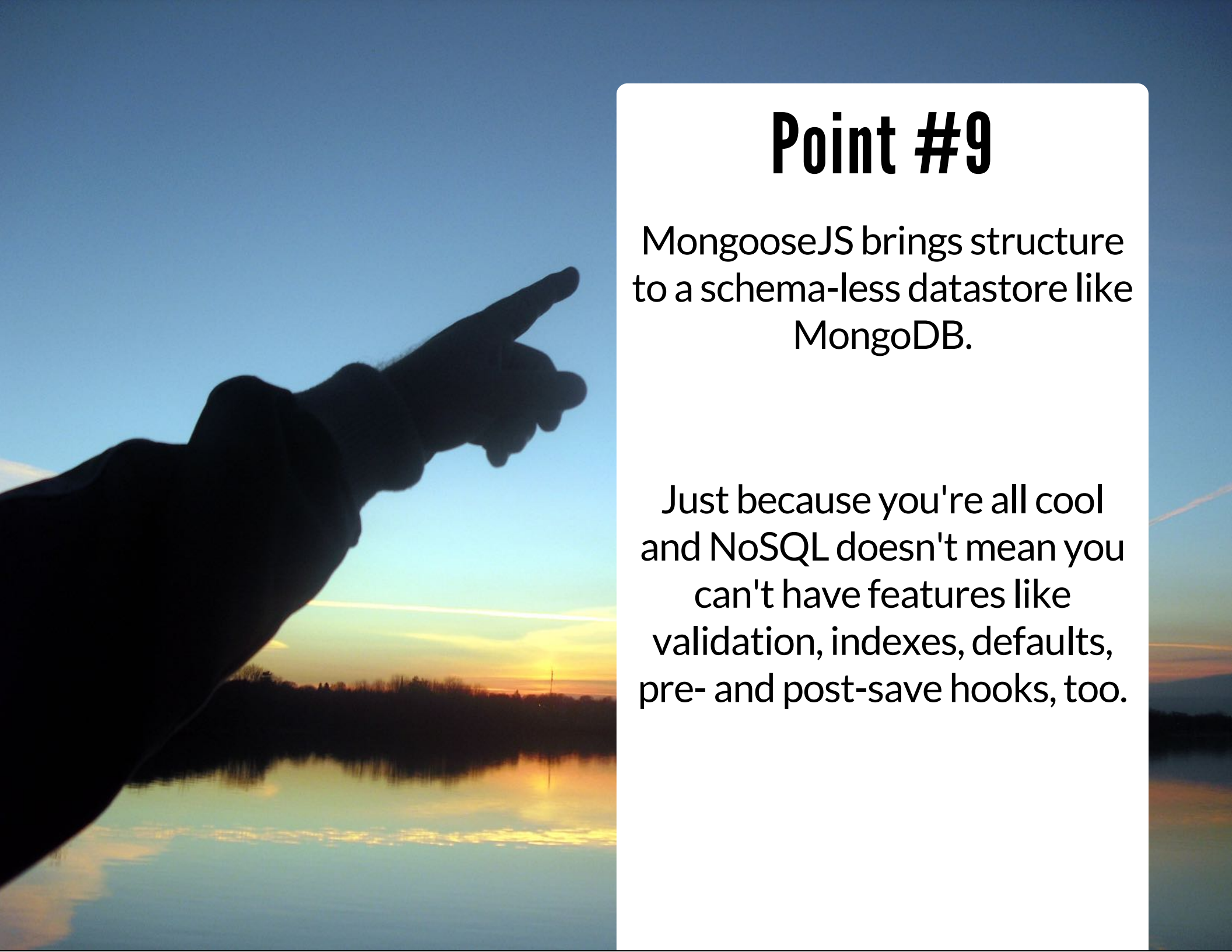
Population

So far we haven't done anything much different. We've merely created a `Person` and a `Story`. Now let's take a look at populating our story's `_creator` using the query builder:

```
Story
.findOne({ title: 'Once upon a timex.' })
.populate('_creator')
.exec(function (err, story) {
  if (err) return handleError(err);
  console.log('The creator is %s', story._creator.name);
  // prints "The creator is Aaron"
})
```

Populated paths are no longer set to their original `_id` , their value is replaced with the mongoose document returned from the database by performing a separate query before returning the results.

Arrays of refs work the same way. Just call the populate method on the query and an array of documents will be returned *in place* of the original `_ids`.



Point #9

MongooseJS brings structure to a schema-less datastore like MongoDB.

Just because you're all cool and NoSQL doesn't mean you can't have features like validation, indexes, defaults, pre- and post-save hooks, too.

AngularJS === User Interface

https://angularjs.org/

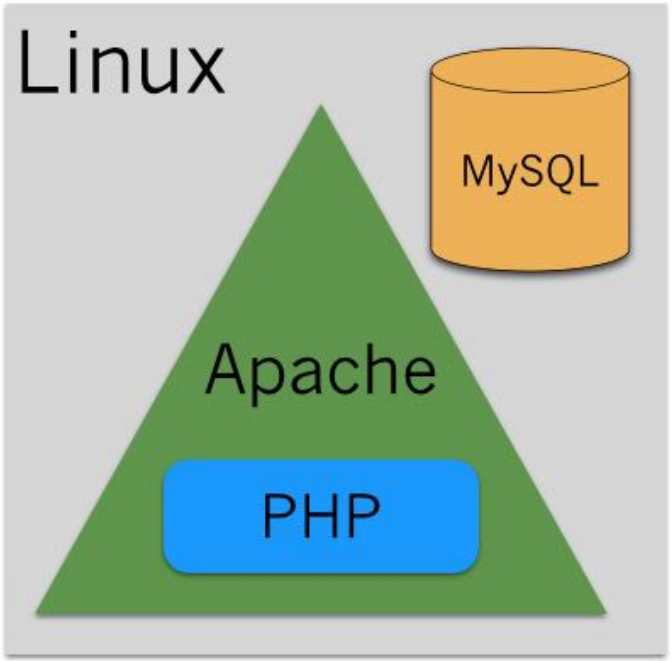
Linux ➡ NodeJS

Apache ➡ ExpressJS

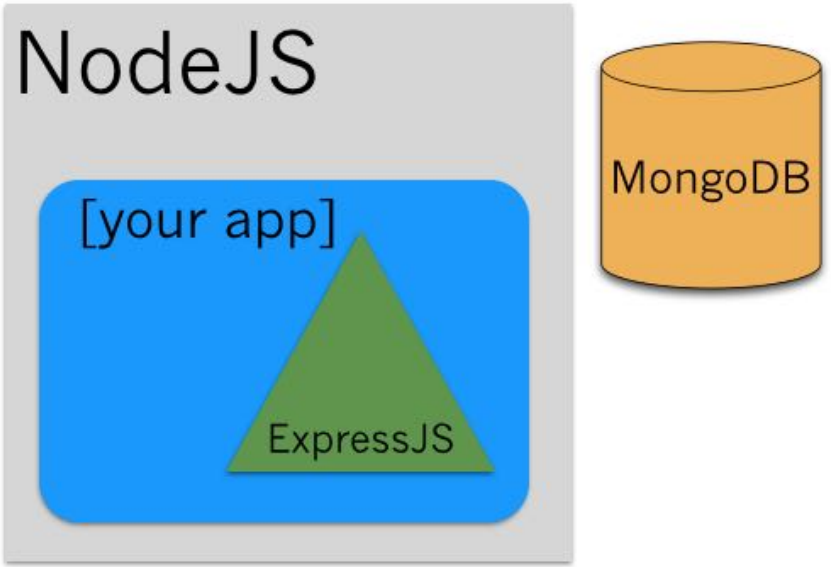
MySQL ➡ MongoDB

Perl ➡ AngularJS

LAMP


Linux
MySQL
Apache
PHP

MEAN


NodeJS
[your app]
ExpressJS
MongoDB


Browser
[your app]
AngularJS

Single-page application

From Wikipedia, the free encyclopedia

A **single-page application** (**SPA**), is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[1] or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although modern web technologies (such as those included in the HTML5 `pushState()` API) can provide the perception and navigability of separate logical pages in the application. Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

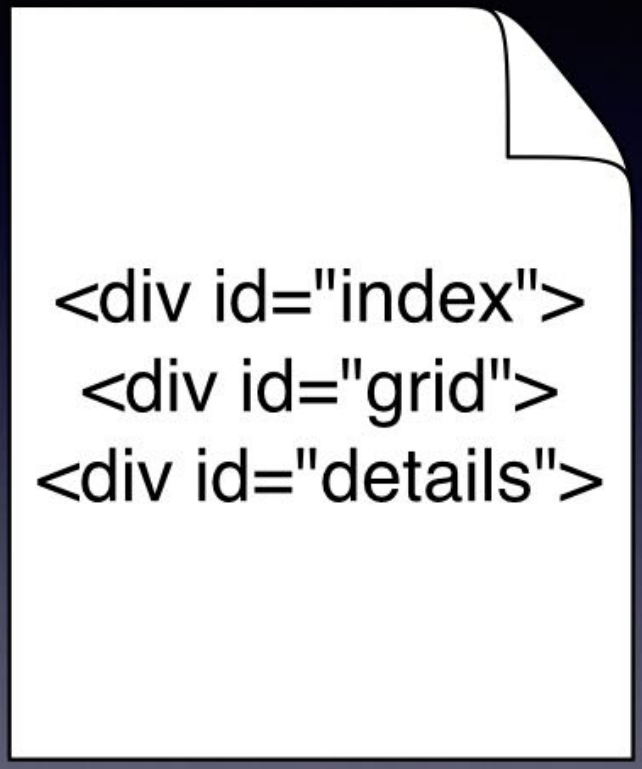
A SPA has four characteristics:

- Single Page
- Client-side
- Component-oriented
- Asynchronous / Event-driven

1. Traditional Multipage App vs. Single-page App

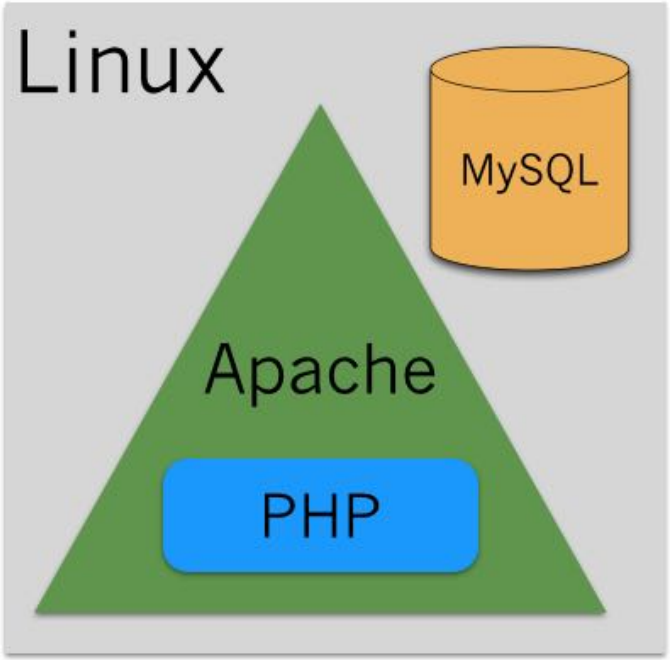
1. Traditional Multipage App vs. Single-page App

1. Traditional Multipage App vs. Single-page App



Think "State Transition"
instead of "Page Transition"

LAMP



Browser

2. Server-side MVC vs. Client-side MVC

2. Server-side MVC vs. Client-side MVC

Request: HTTP GET
/search?q=JavaScript

2. Server-side MVC vs. Client-side MVC

Router

Request: HTTP GET
/search?q=JavaScript

2. Server-side MVC vs. Client-side MVC

Request: HTTP GET
/search?q=JavaScript

2. Server-side MVC vs. Client-side MVC

Database

Model(s)

View

Controller

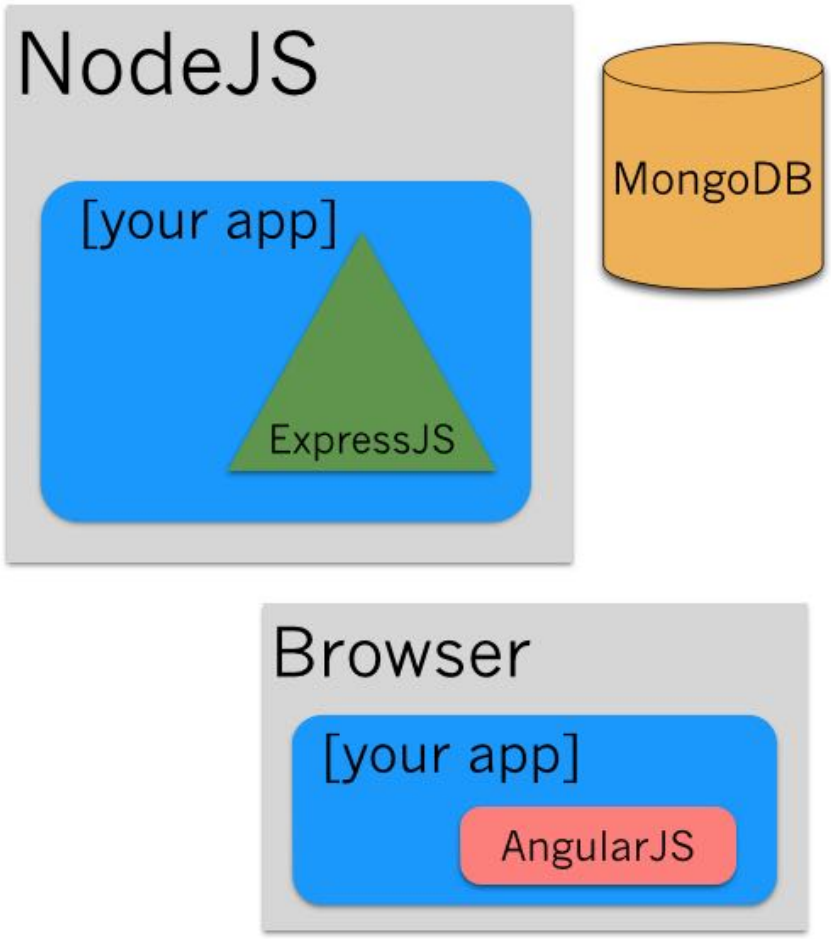
Router

Request: HTTP GET
/search?q=JavaScript

Response:
text/html

results
.html

MEAN


NodeJS
[your app]
ExpressJS
MongoDB
Browser
[your app]
AngularJS

2. Server-side MVC vs. Client-side MVC

2. Server-side MVC vs. Client-side MVC

#search?q=JavaScript

2. Server-side MVC vs. Client-side MVC

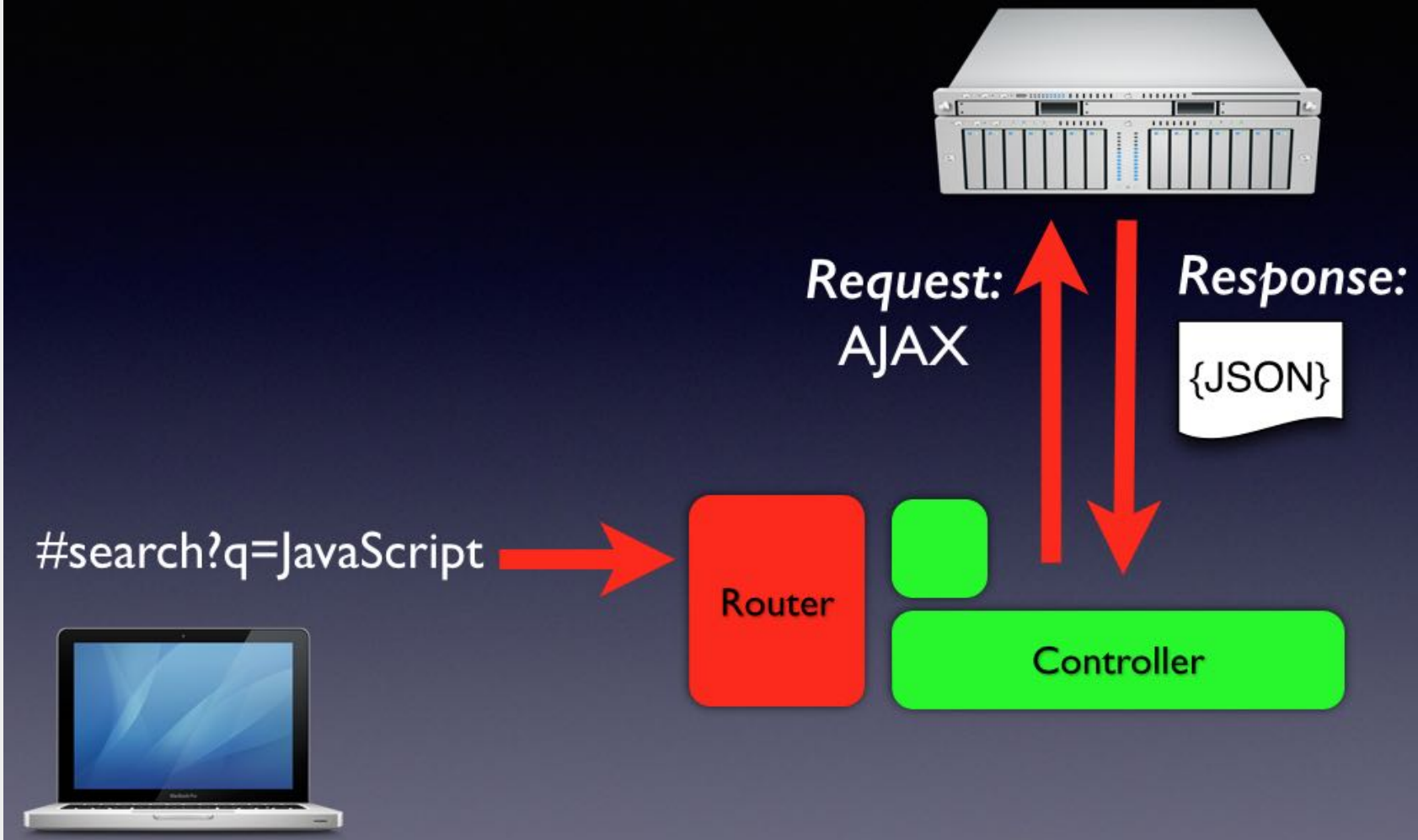
#search?q=JavaScript

Router

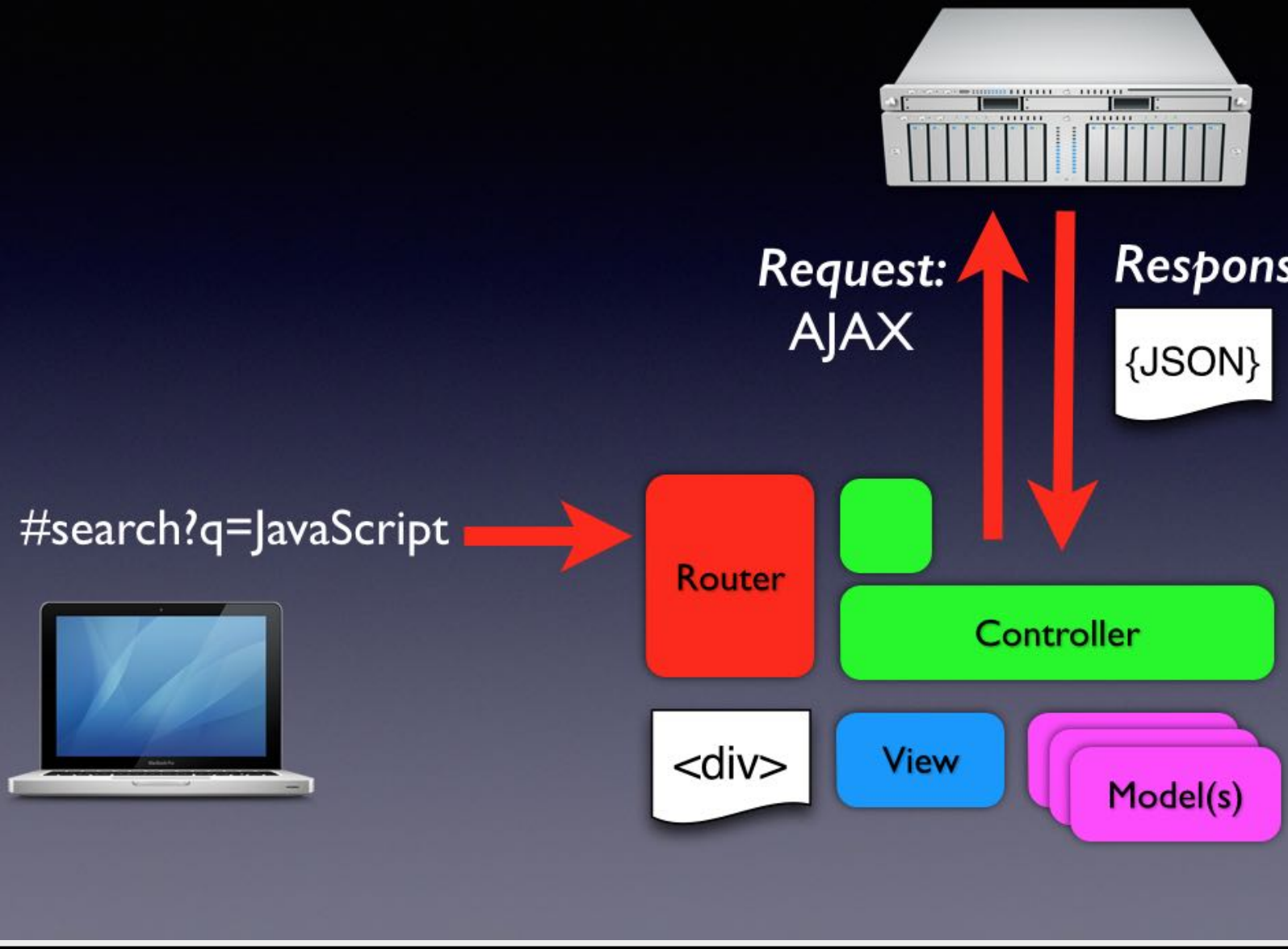
2. Server-side MVC vs. Client-side MVC

#search?q=JavaScript

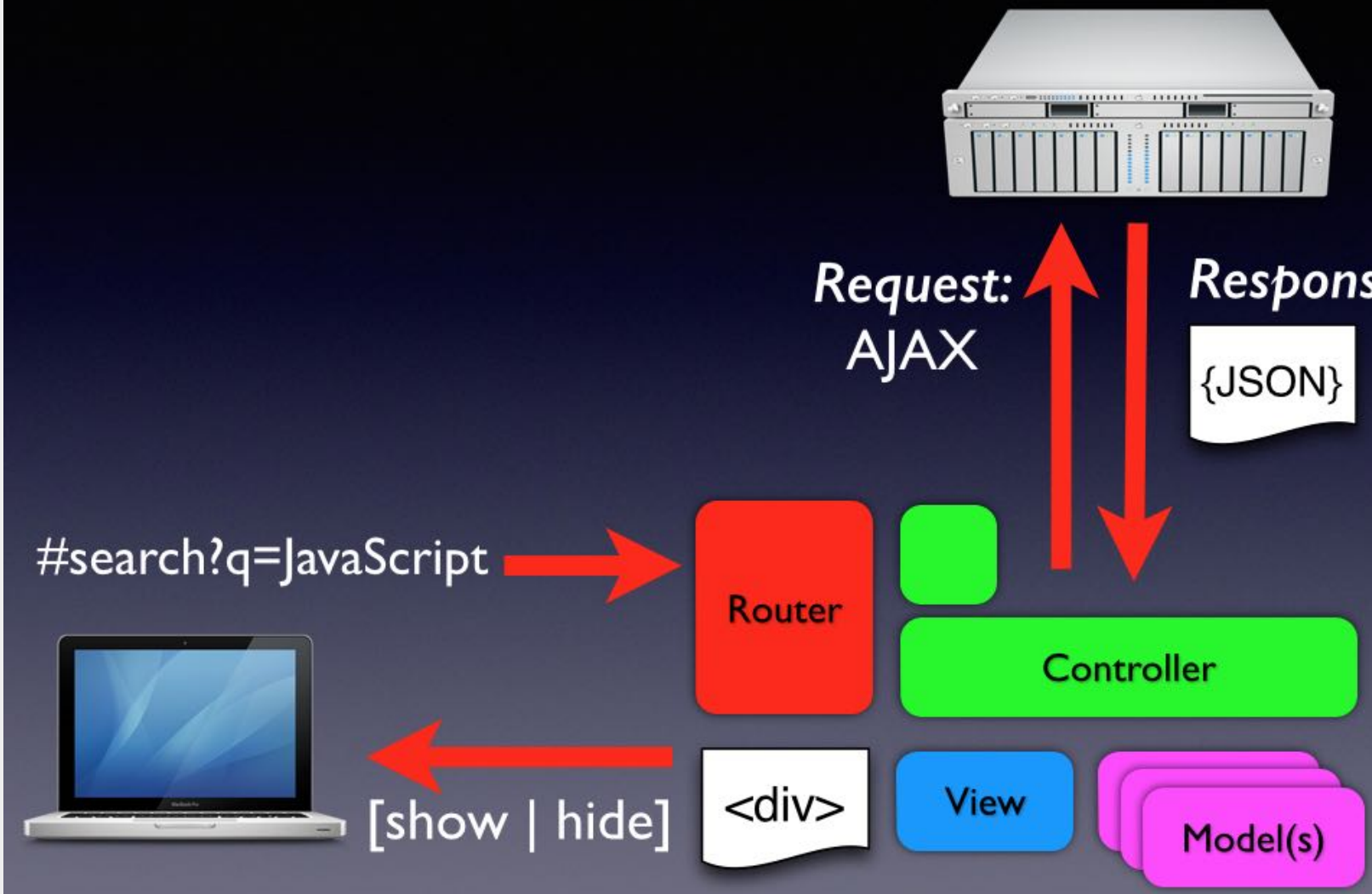
2. Server-side MVC vs. Client-side MVC



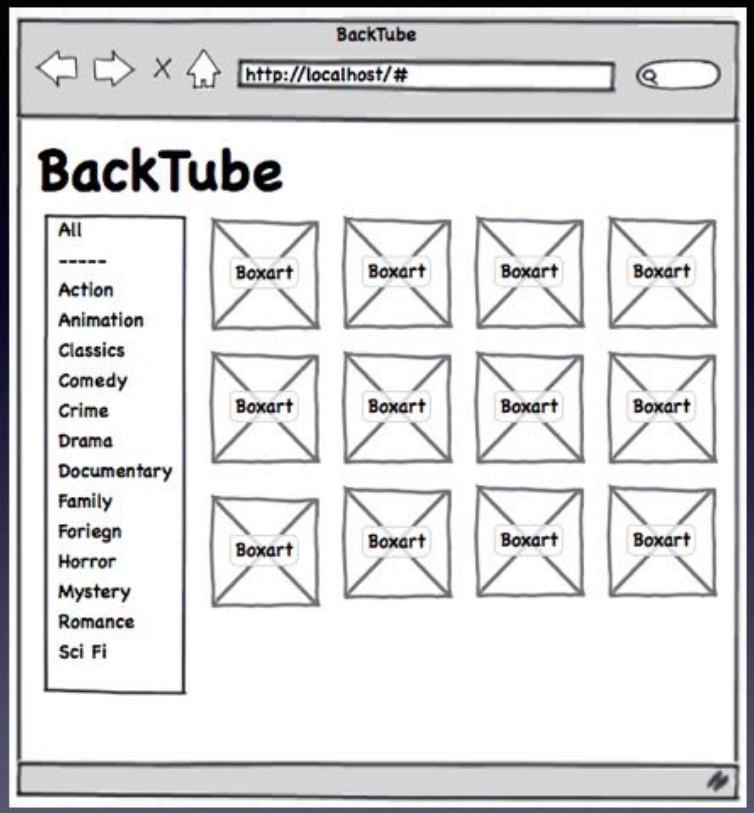
2. Server-side MVC vs. Client-side MVC



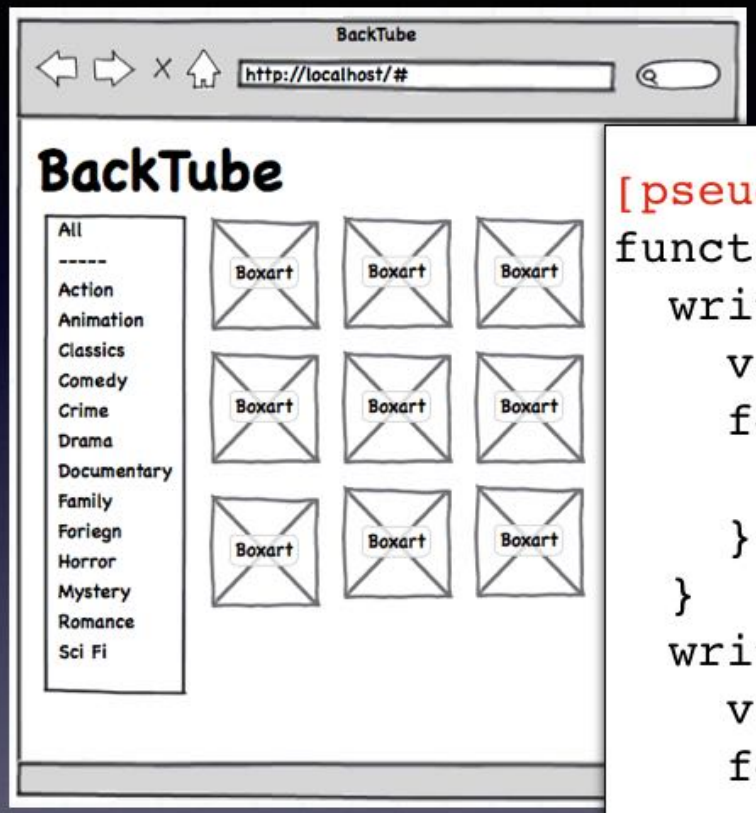
2. Server-side MVC vs. Client-side MVC



3. Page-centric vs. Component-oriented

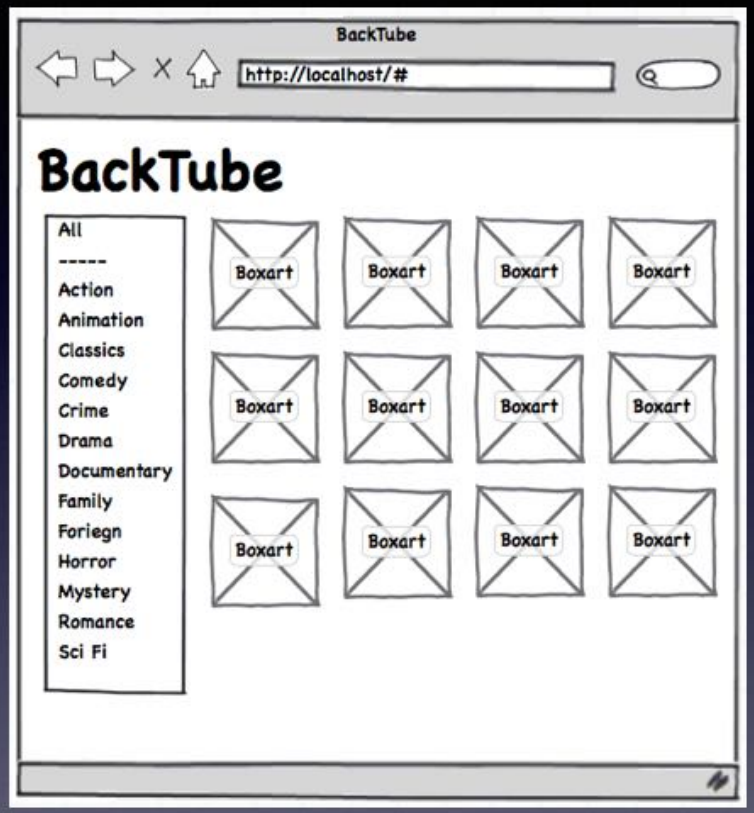


3. Page-centric vs. Component-oriented

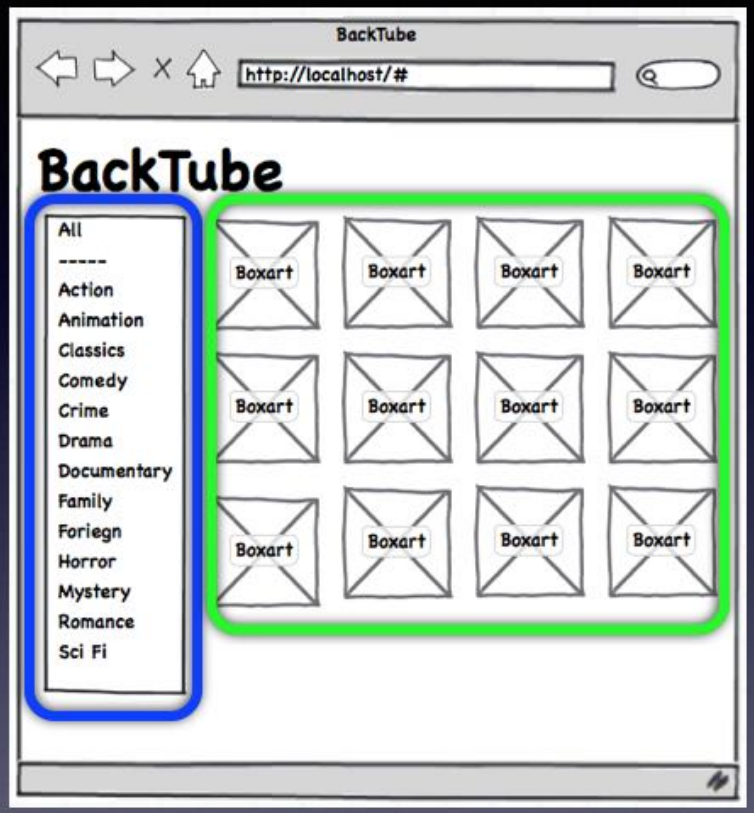



```
[pseudocode]
function buildPage(){
  write.menuCategories{
    var sidebar = $("#sidebar");
    for(i=0;i<length;i++){
      sidebar.append("<li>...");
    }
  }
  write.movies{
    var movieList = $("#list");
    for(i=0;i<length;i++){
      movieList.append("<li>...");
    }
  }
}
```

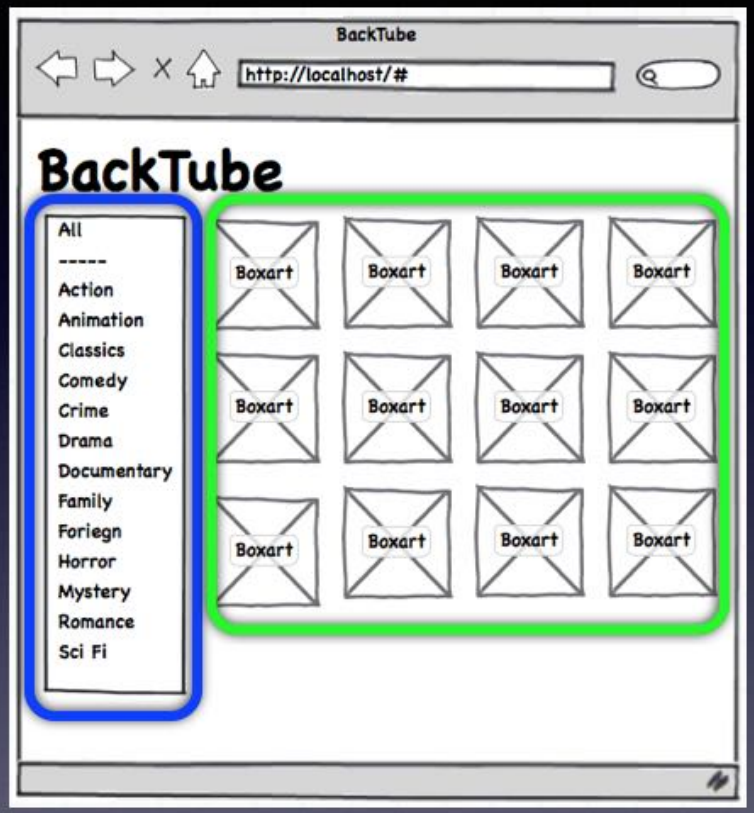
3. Page-centric vs. Component-oriented



3. Page-centric vs. Component-oriented



3. Page-centric vs. Component-oriented




```
[pseudocode]
var sidebar = new Sidebar();
var list = new MovieList();

sidebar.addEventListener(
  {component:list}
);
```

4. Synchronous vs. Asynchronous / Event-driven

4. Synchronous vs. Asynchronous / Event-driven

```
[pseudocode - synchronous]
function buildPage(){
  var items = getMenuItems();
  writeSidebar(items);
  var movies = getMovies();
  writeMovies(movies);
}
```

4. Synchronous vs. Asynchronous / Event-driven

4. Synchronous vs. Asynchronous / Event-driven

```
[sidebar - onClick]
$scope.viewMoviesOfType = function(categoryId){
  Movies.query({'catId': categoryId})
        .$promise
        .then(function(ajaxResponse){
            var data = {
              'movieList':ajaxResponse
            };
            $scope.$emit('changeMainView',
                         {'data':data});
        });
}

[main - eventListener]
$scope.$on('changeMainView', function(e, args){
             $scope.data = args.data;
           });
```

ANGULARJS

by Google

HTML enhanced for web apps!

1. Traditional Multipage App vs. Single-page App
2. Server-side MVC vs. Client-side MVC
3. Page-centric vs. Component-oriented
4. Synchronous vs. Asynchronous / Event-driven

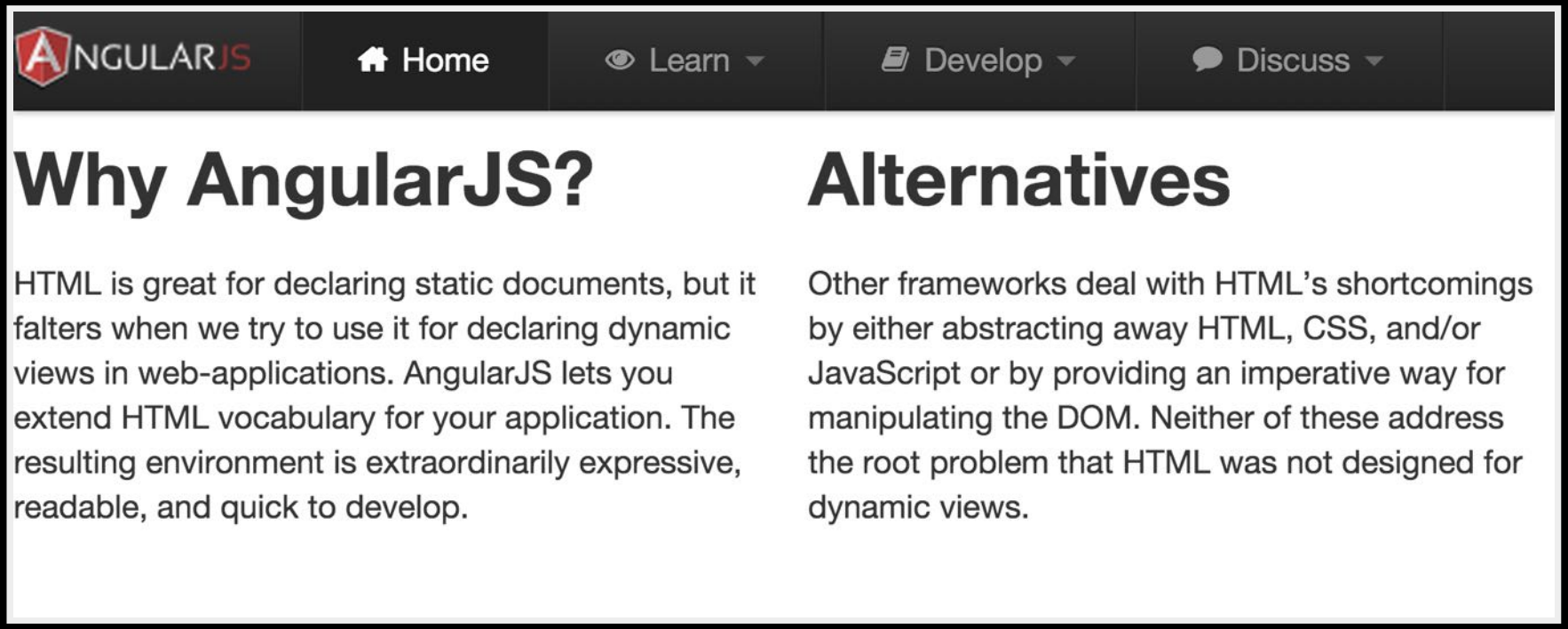
AngularJS has four characteristics:

- HTML-centric
- Declarative
- Component-oriented
- Dependency-injection

AngularJS has four characteristics:

JavaScript-centric ➠ HTML-centric

https://angularjs.org/


ANGULARJS
Home
Learn
Develop
Discuss
Why AngularJS?
HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.
Alternatives
Other frameworks deal with HTML's shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views.

Tabs

Presentations | Speakers

- Custom AngularJS Directives
- Polymer: Less JS, More Righteous
- Developing Offline Applications

tabs.html (HTML)

```
<body>
  <h1>Tabs</h1>
  <tabs>
    <pane title="Presentations">
      <ul>
        <li>Custom AngularJS Directives</li>
        <li>Polymer: Less JS, More Righteous</li>
        <li>Developing Offline Applications</li>
      </ul>
    </pane>

    <pane title="Speakers">
      <ul>
        <li>Scott Davis</li>
        <li>Brian Sletten</li>
        <li>Venkat Subramaniam</li>
      </ul>
    </pane>
  </tabs>
</body>
```

GOAL:

Create a couple of custom AngularJS Directives for
<tabs> and <pane>

AngularJS === Pre-processor

What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile`) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Angular comes with a set of these directives built-in, like `ngBind` , `ngModel` , and `ngClass` . Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

What does it mean to "compile" an HTML template? For AngularJS, "compilation" means attaching event listeners to the HTML to make it interactive. The reason we use the term "compile" is that the recursive process of attaching directives mirrors the process of compiling source code in compiled programming languages.

Directive === Custom HTML Element (or Custom HTML Attribute)

```
<div ng-controller="Controller">
  <my-customer></my-customer>
</div>
```

AngularJS has four characteristics:

JavaScript-centric ➠ HTML-centric

Imperative ➠ Declarative

Declarative vs. Imperative

AngularJS is Declarative:
This is what I'd like to happen.

jQuery is Imperative:
This is how you should do it.

Simple AngularJS

Name: Bubba

Hello Bubba!

simple_angularjs.html

```
<!-- SOURCE: https://angularjs.org/ -->
<html ng-app>
  <head>
    <title>Simple AngularJS</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
                 1.2.20/angular.min.js"></script>
  </head>
  <body>
    <h1>Simple AngularJS</h1>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName" placeholder="Enter a name here">
      <hr>
      <h1>Hello {{yourName}}!</h1>
    </div>
  </body>
</html>
```

Simple jQuery

Name: Jane Doe

Hello Jane Doe!

simple_jquery.html

```
<html>
  <head>
    <title>Simple jQuery</title>
    <script src="//code.jquery.com/jquery-2.1.1.min.js"></script>
    <script>
    'use strict';
    $(document).ready(function(){
        var inputBox = $('#yourName');
        inputBox.on('keyup', function(){
            $('#yourNameGoesHere').html(inputBox.val());
        });
    });
    </script>
  </head>
  <body>
    <h1>Simple jQuery</h1>
    <div>
      <label>Name:</label>
      <input type="text" id="yourName" placeholder="Enter a name here">
      <hr>
      <h1>Hello <span id="yourNameGoesHere"></span>!</h1>
    </div>
  </body>
```

To paraphrase my friend
Dr. Venkat Subramaniam:

"You give imperative
instructions to children;
declarative instructions to
adults..."

My wife to me:

"Go upstairs and take a shower before dinner."

My wife to my son:

1. Go upstairs now
2. Walk into the bathroom
3. Take off all of your clothes
4. Turn on the water
5. Get in the shower
6. Wash your hair AND your face AND your body...

Iterate AngularJS

- AngularJS
- jQuery

iterate-angularjs.html (HTML)

```
<body>
  <h1>Iterate AngularJS</h1>
  <ul ng-controller="LibController">
    <li ng-repeat="lib in libraries">
      <a href="{{lib.url}}">{{lib.name}}</a>
    </li>
  </ul>
</body>
```

iterate-angularjs.html (JS)

```
<script>
  var LibController = function($scope){
    $scope.libraries = [
      {
        name: "AngularJS",
        url: "https://angularjs.org/"
      },
      {
        name: "jQuery",
        url: "http://jquery.com/"
      }
    ];
  };
</script>
```

Iterate jQuery

- AngularJS
- jQuery

iterate-jquery.html (HTML)

```
<body>
  <h1>Iterate jQuery</h1>
  <ul id="libs"></ul>
</body>
```

iterate-jquery.html (JS)

```
<script>
  'use strict';
  var libraries = [
      {
        name: "AngularJS",
        url: "https://angularjs.org/"
      },
      {
        name: "jQuery",
        url: "http://jquery.com/"
      }
    ];

  $(document).ready(function(){
    $.each(libraries, function(index, lib){
      var li = "<li><a href='" + lib.url + "'>" + lib.name + "</a>";
      $("#libs").append(li);
    })
  });
</script>
```

AngularJS has four characteristics:

JavaScript-centric ➠ HTML-centric

Imperative ➠ Declarative

Page-oriented ➠ Component-oriented

page-oriented ➠ DOM === global variable

Global Variables Are Bad

This is something I have a hard time putting in words. I've been bitten by globals in the past, so I 'know' they're 'bad', but for the life of me, I can't explain why. What I'd like is to have some straightforward "Here's why globals are bad" document I can point other people to, preferably with some concrete (if toy) examples.

As with all HeuristicRules, this is not a rule that applies 100% of the time. Code is generally clearer and easier to maintain when it does not use globals, but there are exceptions. It is similar in spirit to GotoConsideredHarmful, although use of global variables is less likely to get you branded as an inveterate hacker.

Why Global Variables Should Be Avoided When Unnecessary

- **Non-locality** -- Source code is easiest to understand when the scope of its individual elements are limited. Global variables can be read or modified by any part of the program, making it difficult to remember or reason about every possible use.
- **No Access Control or Constraint Checking** -- A global variable can be get or set by any part of the program, and any rules regarding its use can be easily broken or forgotten. (In other words, get/set accessors are generally preferable over direct data access, and this is even more so for global data.) By extension, the lack of access control greatly hinders achieving security in situations where you may wish to run untrusted code (such as working with 3rd party plugins).
- **Implicit coupling** -- A program with many global variables often has tight couplings between some of those variables, and couplings between variables and functions. Grouping coupled items into cohesive units usually leads to better programs.
- **Concurrency issues** -- if globals can be accessed by multiple threads of execution, synchronization is necessary (and too-often neglected). When dynamically linking modules with globals, the composed system might not be thread-safe even if the two independent modules tested in dozens of different contexts were safe.
- **Namespace pollution** -- Global names are available everywhere. You may unknowingly end up using a global when you think you are using a local (by misspelling or forgetting to declare the local) or vice versa. Also, if you ever have to link together modules that have the same global variable names, if you are lucky, you will get linking errors. If you are unlucky, the linker will simply treat all uses of the same name as the same object.
- **Memory allocation issues** -- Some environments have memory allocation schemes that make allocation of globals tricky. This is *especially* true in languages where "constructors" have side-effects other than allocation (because, in that case, you can express unsafe situations where two globals mutually depend on one another). Also, when dynamically linking modules, it can be unclear whether different libraries have their own instances of globals or whether the globals are shared.
- **Testing and Confinement** - source that utilizes globals is somewhat more difficult to test because one cannot readily

Encapsulation (object-oriented programming)

From Wikipedia, the free encyclopedia

Encapsulation is the packing of data and functions into a single component. The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist. It allows selective hiding of properties and methods in an object by building an impenetrable wall to protect the code from accidental corruption.

In programming languages, **encapsulation** is used to refer to one of two related but distinct notions, and sometimes to the combination[1][2] thereof:

- A language mechanism for restricting access to some of the object's components.[3][4]
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.[5][6]

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object-oriented programming, while other programming languages which provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many OOP languages hiding of components is not automatic or can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.

AngularJS Scopes

What are Scopes?

Improve this Doc

scope is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions and propagate events.

Scope characteristics

Scopes provide APIs ($watch) to observe model mutations.

Scopes provide APIs ($apply) to propagate any model changes through the system into the view from outside of the "Angular realm" (controllers, services, Angular event handlers).

Scopes can be nested to limit access to the properties of application components while providing access to shared model properties. Nested scopes are either "child scopes" or "isolate scopes". A "child scope" (prototypically) inherits properties from its parent scope. An "isolate scope" does not. See isolated scopes for more information.

Scopes provide context against which expressions are evaluated. For example `{{username}}` expression is meaningless, unless it is evaluated against a specific scope which defines the `username` property.

iterate-angularjs.html (JS)

```
<script>
  var LibController = function($scope){
    $scope.libraries = [
      {
        name: "AngularJS",
        url: "https://angularjs.org/"
      },
      {
        name: "jQuery",
        url: "http://jquery.com/"
      }
    ];
  };
</script>
```

iterate-angularjs.html (HTML)

```
<body>
  <h1>Iterate AngularJS</h1>
  <ul ng-controller="LibController">
    <li ng-repeat="lib in libraries">
      <a href="{{lib.url}}">{{lib.name}}</a>
    </li>
  </ul>
</body>
```

Angular scopes lead to directives

Once you understand scopes,
the next logical step is directives and isolate scope.

What we want to be able to do is separate the scope inside a directive from the scope outside, and then map the outer scope to a directive's inner scope. We can do this by creating what we call an **isolate scope**. To do this, we can use a directive's `scope` option:

script.js | index.html | my-customer-iso.html | Edit in Plunker

```
<div ng-controller="Controller">
  <my-customer info="naomi"></my-customer>
  <hr>
  <my-customer info="igor"></my-customer>
</div>
```

Name: Naomi Address: 1600 Amphitheatre

Name: Igor Address: 123 Somewhere

What we want to be able to do is separate the scope inside a directive from the scope outside, and then map the outer scope to a directive's inner scope. We can do this by creating what we call an **isolate scope**. To do this, we can use a directive's `scope` option:

script.js | index.html | my-customer-iso.html | Edit in Plunker

```
angular.module('docsIsolateScopeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.naomi = { name: 'Naomi', address: '1600 Amphitheatre' };
  $scope.igor = { name: 'Igor', address: '123 Somewhere' };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    scope: {
      customerInfo: '=info'
    },
    templateUrl: 'my-customer-iso.html'
  };
});
```

Name: Naomi Address: 1600 Amphitheatre

Name: Igor Address: 123 Somewhere

Angular v2.0.0

The Changing Web **All About Angular 2.0**

While massive changes have happened in the last couple of years, they pale in comparison to what's coming in the next 1-3 years. In a few months the ES6 spec will be finalized. It's not unreasonable to think that we'll see a browser in 2015 that implements the full spec. Today's browsers already support some of these features and are working on implementations of the rest right now. This means browser support for things like modules, classes, lambdas, generators, etc. These features fundamentally transform the JavaScript programming experience. But big changes aren't constrained merely to JavaScript. Web Components are on the horizon. The term Web Components usually refers to a collection of four related W3C specifications:

- Custom Elements - Enables the extension of HTML through custom tags.
- HTML Imports - Enables packaging of various resources (HTML, CSS, JS, etc.).
- Template Element - Enables the inclusion of inert HTML in a document.
- Shadow DOM - Enables encapsulation of DOM and CSS.

WebComponents.org

a place to discuss and evolve web component best-practices

POLYFILLS

The webcomponent.js polyfills enable Web Components in (evergreen) browsers that lack native support.

Install with Bower

```
bower install webcomponentsjs
```

Install with npm

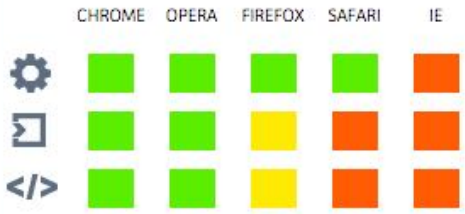
```
npm install webcomponents.js
```

Download webcomponents.js

0.5.5 (105KB minified, 31KB gzipped)

learn more about the polyfills

BROWSER SUPPORT



SPECS

CUSTOM ELEMENTS

This specification describes the method for enabling the author to define and use new types of DOM elements in a document.

HTML IMPORTS

HTML Imports are a way to include and reuse HTML documents in other HTML documents.

TEMPLATES

This specification describes a method for declaring inert DOM subtrees in HTML and manipulating them to instantiate document fragments with identical contents.

SHADOW DOM

This specification describes a method of

PRESENTATIONS

CREATING CONTAINER COMPONENTS IN WEB COMPONENTS AND ANGULAR

Architecting your Angular application with reusable components can be complicated. Many times, UI components that need multiple entry points for user markup and the standard ng-transclude do not do enough of what is required. Using Web Component standards, like the Shadow DOM, we can now enable our applications to easily handle this.

Read More >

see all presentations

COMMUNITY

AngularJS has four characteristics:

JavaScript-centric ➠ HTML-centric

Imperative ➠ Declarative

Page-oriented ➠ Component-oriented

Constructors (new) ➠ Dependency Injection

Dependency Injection

Improve this Doc

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Using Dependency Injection

DI is pervasive throughout Angular. You can use it when defining components or when providing `run` and `config` blocks for a module.

- Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function. These components can be injected with "service" and "value" components as dependencies.
- Controllers are defined by a constructor function, which can be injected with any of the "service" and "value" components as dependencies, but they can also be provided with special dependencies. See Controllers below for a list of these special dependencies.
- The `run` method accepts a function, which can be injected with "service", "value" and "constant" components as dependencies. Note that you cannot inject "providers" into `run` blocks.

Why Dependency Injection?

This section motivates and explains Angular's use of DI. For how to use DI, see above.

For in-depth discussion about DI, see Dependency Injection at Wikipedia, Inversion of Control by Martin Fowler, or read about DI in your favorite software design pattern book.

There are only three ways a component (object or function) can get a hold of its dependencies:

1. The component can create the dependency, typically using the `new` operator.
2. The component can look up the dependency, by referring to a global variable.
3. The component can have the dependency passed to it where it is needed.

The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies for test isolation.

The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.

```
function SomeClass(greeter) {
  this.greeter = greeter;
}

SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

Asking for dependencies solves the issue of hard coding, but it also means that the injector needs to be passed throughout the application. Passing the injector breaks the Law of Demeter. To remedy this, we use a declarative notation in our HTML templates, to hand the responsibility of creating components over to the injector, as in this example:

```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}
```

When Angular compiles the HTML, it processes the `ng-controller` directive, which in turn asks the injector to create an instance of the controller and its dependencies.

```
injector.instantiate(MyController);
```

This is all done behind the scenes. Notice that by having the `ng-controller` ask the injector to instantiate the class, it can satisfy all of the dependencies of `MyController` without the controller ever knowing about the injector.

This is the best outcome. The application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.

Angular DI ➠ stupid-easy testing

$httpBackend

View Source | Improve this Doc

- service in module ngMock

Fake HTTP backend implementation suitable for unit testing applications that use the $http service.

Note: For fake HTTP backend implementation suitable for end-to-end testing or backend-less development please see e2e $httpBackend mock.

During unit testing, we want our unit tests to run quickly and have no external dependencies so we don't want to send XHR or JSONP requests to a real server. All we really need is to verify whether a certain request has been sent or not, or alternatively just let the application make requests, respond with pre-trained responses and assert that the end result is what we expect it to be.

This mock implementation can be used to respond with static or dynamic responses via the `expect` and `when` apis and their shortcuts (`expectGET` , `whenPOST` , etc).

When an Angular application needs some data from a server, it calls the $http service, which sends the request to a real server using $httpBackend service. With dependency injection, it is easy to inject $httpBackend mock (which has the same API as $httpBackend) and use it to verify the requests and respond with some testing data without sending a request to a real server.

There are two ways to specify what test data should be returned as http responses by the mock backend when the code under test makes http requests:

- `$httpBackend.expect` - specifies a request expectation
- `$httpBackend.when` - specifies a backend definition

AngularJS has four characteristics:

- HTML-centric
- Declarative
- Component-oriented
- Dependency-injection

Conclusion

LAMP ➠ NEMA

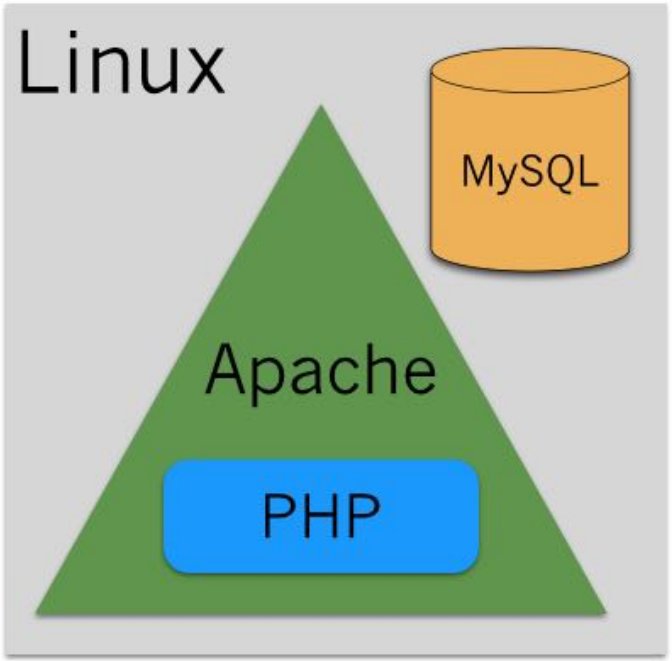
Linux ➠ NodeJS (Platform)

Apache ➠ ExpressJS (Web Server)

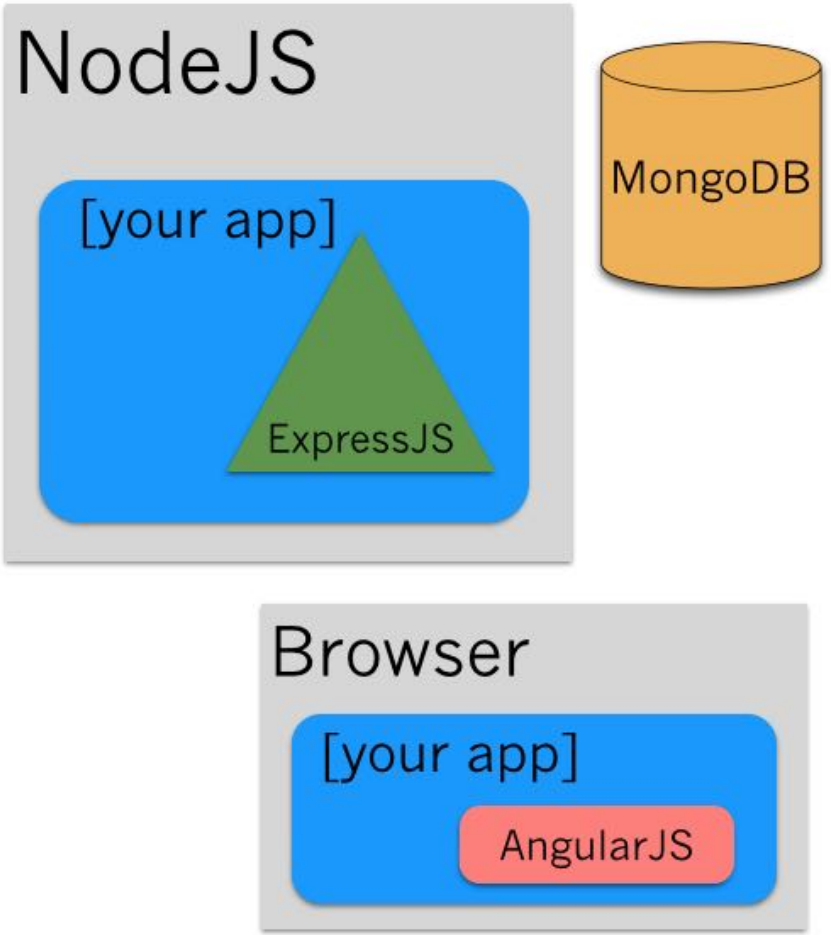
MySQL ➠ MongoDB (Persistence)

Perl ➠ AngularJS (User Interface)

LAMP


Linux
MySQL
Apache
PHP

MEAN


NodeJS
[your app]
ExpressJS
MongoDB
Browser
[your app]
AngularJS


0
DAYS
SINCE LAST NEW
JAVASCRIPT FRAMEWORK

"WHO HAS BEEN TASTING MY SOUP?"

The Goldilocks Framework

This soup is too hot...

too cold...

just right...

EXPLORING THE ARCHITECTURE OF THE MEAN STACK

MongoDB, ExpressJS, AngularJS, NodeJS

Scott Davis
Web: http://thirstyhead.com
Twitter: @scottdavis99
Slides: http://my.thirstyhead.com