BOSTON UNIVERSITY

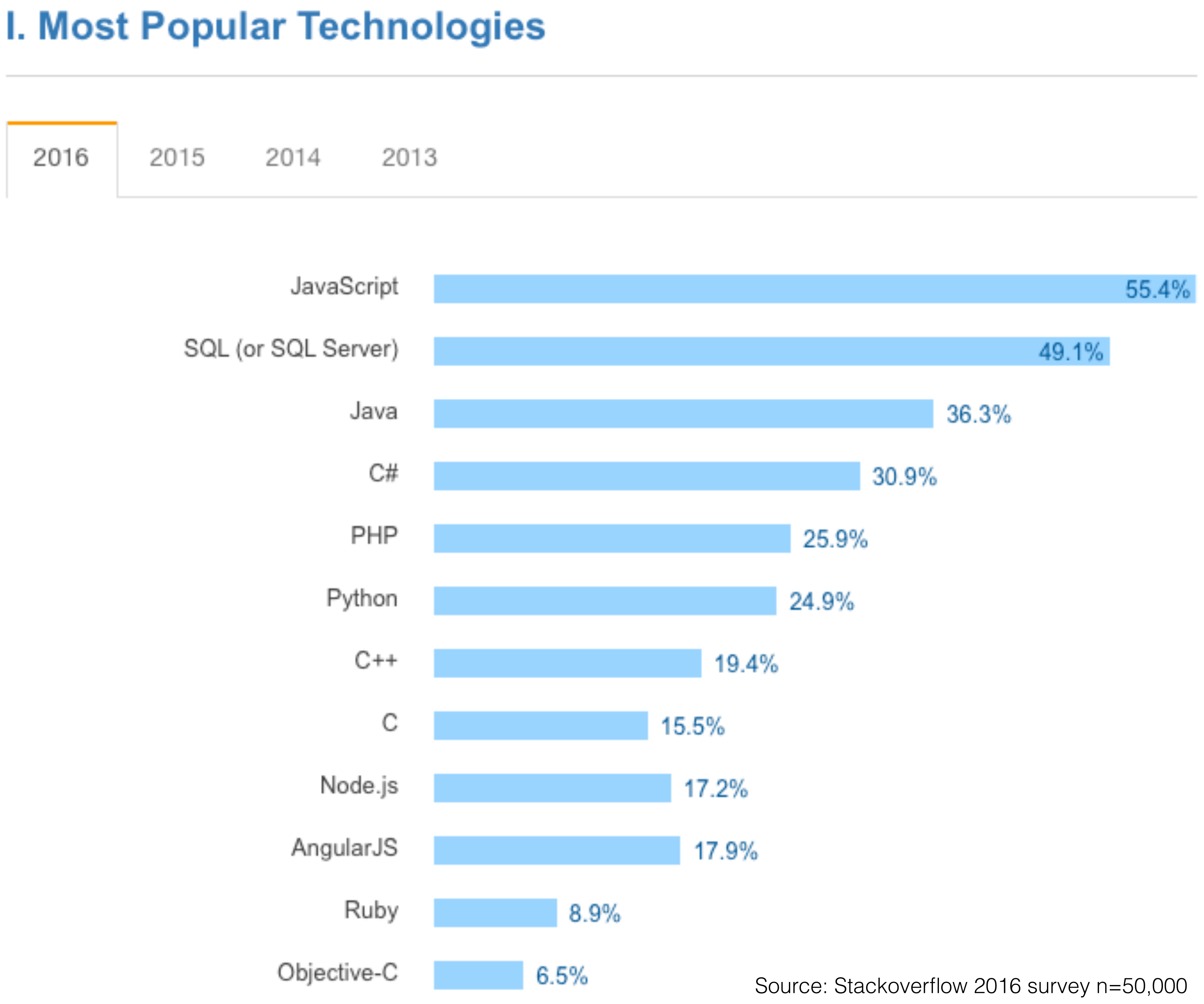
Department of Computer Science

CS591 MEAN: Angular.js

@perrydBUCS

What it is

- Open source (MIT license)
- A client-side framework
- Cross-browser
- 100% Javascript
- Extremely popular



I. Most Popular Technologies

Source: Stackoverflow 2016 survey n=50,000

- The Angular code compiles in the browser
- This means the ‘application’ is entirely contained and live in the browser
- Since it is straight Javascript it coexists with other libraries such as Google’s Closure

Versions

- The current stable version is 2
- A developmental branch, v4, is released and usable
- v1.6 is a ‘legacy’ branch (but what we’ll use)
- Unfortunately the various branches take a pretty different approach to some of the fundamental concepts of Angular
- The buzz in the community is that if you are comfortable with 1.5 the transition to 2 / 4 will take a few days at most
- 1.6 will be supported for quite a while, and many development projects spinning up right now use it, mainly due to the Typescript requirement in 2 and 4

Angular parts

- The framework is quite rich and we'll only cover a few key portions of it here
- Just like Node/Express, there are an enormous number of styles and approaches to writing Angular apps
- We're going to keep it extremely simple!
- scotch.io is a good resource

Setting it up

- Angular requires jQuery

```
<html lang="en">
<head>
<script   src="https://code.jquery.com/jquery-2.2.3.js"></script>
```

- We then grab the Angular libraries

```
<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.5.2/
angular.js"></script>
```

Angular apps

- Angular functionality is contained in an **application**
- The **ng-app** attribute is used to name the app

```
<body ng-app="cs411">
```

- The app has scope between the opening and closing tags of the HTML element it is attached to (to </body> in this example
- You also can have an ‘anonymous’ app

```
<body ng-app>
```

Live data binding

- At its core Angular is MVC *in the client*
- The rendered page is the **view**, data is the **model**, and Javascript functions are the **controller**
- (Some call the approach MVVC, model, view, view-controller)

Angular expressions

- Angular uses a double-mustache notation to identify expressions
- The expressions are evaluated live in the page

- Expressions represent the *view* part of MVVC
- As the data behind the expression is updated, the expression updates

```
{{12 + 30}}
```

Models

- We use the ng-model attribute to place a value into the model
- Here, when the text in the textbox changes, the expression/view changes with it

```
<input type="text" ng-model="name"/>
<p/>
Your name is: {{name}}
```

- Note that the variable is available in the scope of the controller (as we'll see in a moment it is bound to a $scope object)

Controllers

- An app may have one or more controllers
- The controller has scope within the opening and closing tags it is specified in
- Controllers let us add functions to the app

```
<body ng-app="cs411">
<h1>Welcome!</h1>
<div ng-controller="cs411ctrl">
    ...
</div>
<div ng-controller="anotherController">
    ...
</div>
```

$scope

- Controllers are connected to the page using the **$scope** variable
- $scope is essentially an object shared by the app and the page
- (In Javascript it is easy to add instance and method variables to an object on the fly)
- $scope has the same scope that the controller has
- If you need an app-wide variable to bind to, you can use $rootscope
- (You might need to do this if you have several controllers on a page)

Creating a controller

- A controller is part of an app, and so first we define the app, then its controllers

app name

```
<script>
    angular.module('cs411', [])
            .controller('cs411ctrl', function($scope, $http) {
            … })
```

- Sadly, Angular calls these things ‘apps’ in one place and ‘modules’ in another

app name

```
<body ng-app="cs411">
<h1>Welcome!</h1>
<div ng-controller="cs411ctrl">
```

Adding functions to $scope

- Inside the controller, we can add Javascript functions to $scope so that they can be called from the HTML page

In the controller…

```
angular.module('cs411', [])
        .controller('cs411ctrl', function($scope) {

            $scope.makeItUpper = function() {
                $scope.name = $scope.name.toUpperCase();
            };
```

In the HTML file…

```
<button ng-click="makeItUpper()">UPPER</button>
```

- As we saw earlier, we can add variables to $scope with the ng-model directive

```
<input type="text" ng-model="name"/>
<p/>
Your name is: {{name}}
```

Services

- Angular comes with several built-in services, and you also can define your own
- Common services are
  - $http provides a way to connect to a web server
  - $location gives info about the web page (similar to window.location)
  - $interval and $timeout timers for events
  - $resource for interacting with a RESTful back end

- To use a service in a controller, pass it into the controller function. Example: use the $http service

```
angular.module('cs411', [])
        .controller('cs411ctrl', function($scope, $http) {
```

- This is know as dependency injection
- We also could inject the dependency into the module
- My advice: Inject where needed, not everywhere (the spray-and-pray approach)

Creating services

- Angular services are singletons
- They are typically stateless
    - They have no public variables
    - They perform some sort of function for the caller and return either status or a value
- An example: A service that counts the number of characters in a string

Why would we use services?

- Angular services help us create a cleanly object-oriented application
  - We can move responsibility for an operation to a service object rather that repeating it in several places
  - We can write code that uses the abstraction of the service; it doesn't care how the service performs the work, just that it adheres to a published interface
- Stateless functions — those that have no dependencies on internal variables — are good candidates
- We can easily unit test services and reuse them in other applications

Defining a service

- There are (of course) several ways to create a service in angular
- They are mostly similar; my advice is to pick a method and use it until for some reason you need another method
- I like the ‘factory’ method because it is similar to what I’d do in C++ or Java when I need an object
  - (Note that this is not the Factory or Abstract Factory pattern in which an object is created based on a set of parameters)
- The service is created at the Angular module level (what is called the ‘app’ in the view)

```
angular.module('cs411', [])
       .factory('stringService', function() {

        return {
            countCharacters: function(stringToCount) {
                return stringToCount.length;
            }
        };
    })
    .controller('cs411ctrl', function($scope, $http, stringService){

        //CREATE (POST)
        $scope.createUser = function() {
            if($scope.dbID) {$scope.updateUser($scope.dbID);}
. . .
etc
```

Note: Better to put these in separate files…

- It wouldn’t be unusual for the service itself to use a service
- If that’s the case, just inject it as you would do with a controller

```
angular.module('cs411')
    .factory('stringService', function($http) {

    return {
        countCharacters: function(stringToCount) {
            let url="http://anAPItoCountChars.api.com?string=" + stringToCount;
            return $http.get(url);
        }
    };
})
```

A quick note about $http

- A call to a $http service is asynchronous
- The Javascript engine isn't — it doesn't block
- We have to handle the asynchronous return of data (what we get back from the call is a Javascript promise)
- We can use a .then or .success construct to wait for the result

```
$http.get('http://somewhere.com').
then(function(res) { do something here with returned data in res})
```

Directives

- ng-app, ng-controller and so on are **directives**
- Angular has quite a few built in, and you also can create new ones
- You can even create new HTML tags…

In the controller…

```
.directive('nameDisplay', function() {
    return {
        scope: true,
        restrict: 'EA',
        template: "<b>This can be anything {{name}}</b>"}
```

In the HTML file…

```
<name-display name="{{name}}"></name-display>
```

ng-repeat

- A very useful directive is ng-repeat, which lets us iterate over a series of objects

```
<!— users is a collection of user objects —>
<ul>
    <li ng-repeat="user in users">
        {{user.name}}
    </li>
</ul>
```

- Let’s put all of this together to wire some front end code to the back end user API

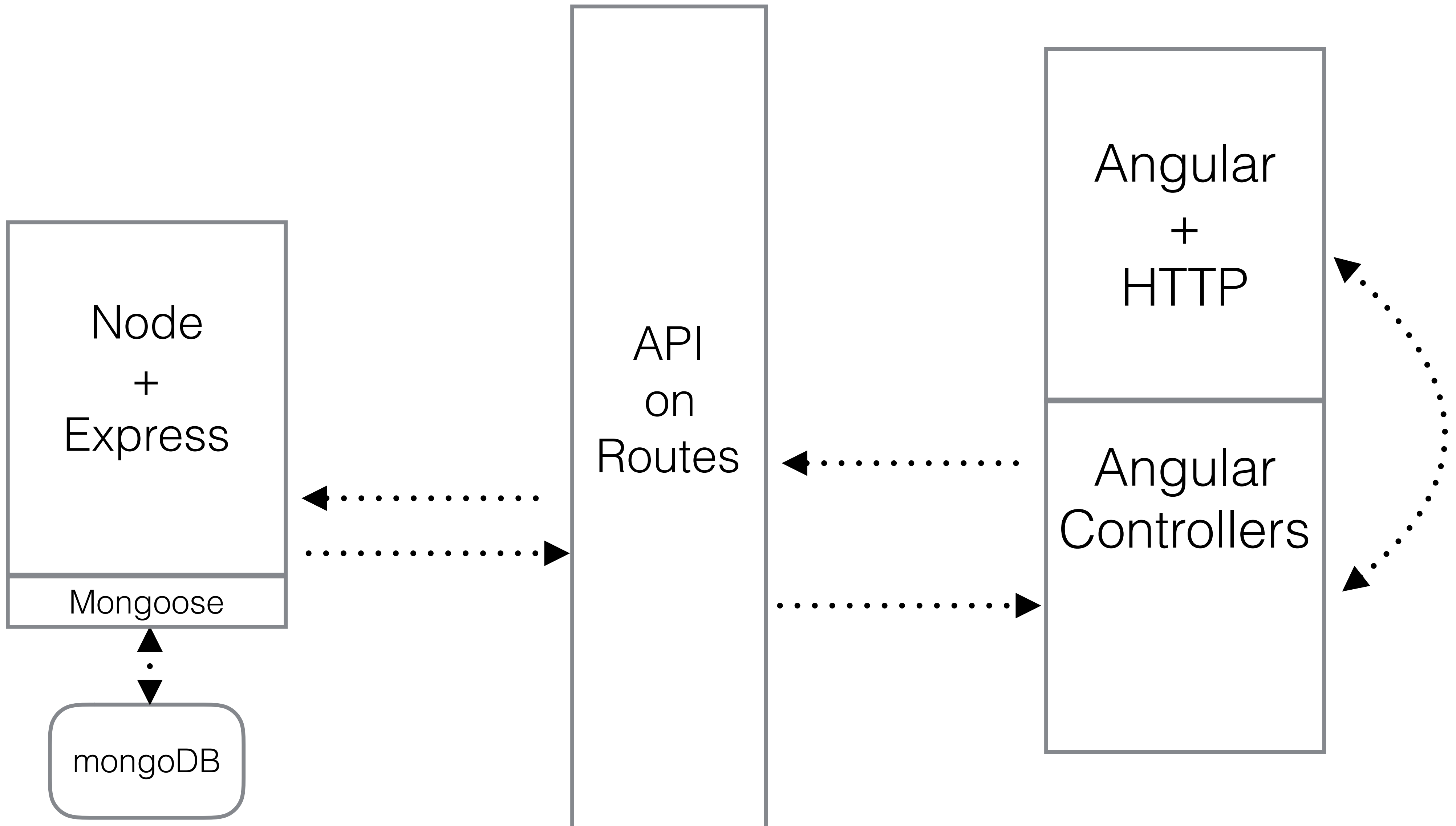
A decoupled app

- Just a word or two about front end and back end
- In this style of app, Node/Express/Mongo are used to create an API for the web application
- Angular is used to create the front end
- The two are completely decoupled…changing something in Mongo will *not* change it on the client (Angular) side
- If you truly need tighter coupling, you can use something like socket.io to send and receive events between client and server

- Also, Node will typically be your web server for the application (though it doesn't have to be), so you need to tell Express when to serve Angular files vs EJS or Jade as it normally would
- Simply define your api on its own route and use the express.static middleware to direct anything else to your HTML files

```
//Back end APIis served on the /api route
app.use('/api', api);

//Pass anything other than /api to Angular
app.use(express.static(path.join(__dirname, 'public')));
```


Node
+
Express
Mongoose
mongoDB
API
on
Routes
Angular
+
HTTP
Angular
Controllers

Things we're *not* talking about

- Angular has a ton of features that we're not looking at right now, but that you might find useful for your app; we'll look at several of these in the next few lectures
  - Directives like ng-show to hide/display divs
  - Angular client-side routing
  - Partials and templates (based on routes)
  - Filters to format model data
  - Internal (in-page) routing
- w3schools.com has some good info on these written in a very concise way
- We've also not mentioned styles…Bootstrap.css isn't a bad way to start