Need for Code Metadata in JavaScript

aStatus: draft, URL: <http://goo.gl/42o100>  
Authors: [misko@google.com](mailto:misko@google.com), [iminar@google.com](mailto:iminar@google.com)

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Goal

The purpose of this document is to present the benefits of having first-class support for code metadata in JavaScript.

This document focusing only on making the argument that code metadata is wildly useful in practice on many platforms and is already being used in JavaScript via various workarounds that suffer from drawbacks.

The following are intentionally out of the scope of this document:

* syntax - how the metadata should be syntactically expressed in JavaScript
* implementation semantics - related doc: [annotations vs decorators](https://docs.google.com/document/d/1QchMCOhxsNVQz2zNvmzy8ibDMPT46MLf79X1QiDc_fU/edit#)
* metadata storage semantics, api and implementation

[Goal](#h.t6qix47izol8)

[Background](#h.vrjx201ykx06)

[Summary of code metadata usage in programming](#h.thrqdhfryunh)

[Better docs](#h.d9zlicnkmu2i)

[Instructions for tools](#h.1q0iz06xfj7i)

[Declarative meta-programing](#h.7pbvytnhyvva)

[DI](#h.aqc4ae8xybcb)

[Schema & entities](#h.u7ksvgqjvi2r)

[Parallel processing / processing order](#h.ifz87i176o07)

[Testing](#h.67n76vxmaa7o)

[Examples of code metadata usage (Prior Art)](#h.okyu12gf8pvv)

[Component frameworks](#h.33y6k4vnp179)

[AngularDart (Dart)](#h.b3uen7cgnl6b)

[Polymer (Dart)](#h.pegt8dvt5pf2)

[Dependency Injection](#h.lwk6sw77mzok)

[di.dart (Dart)](#h.xa2ar09xdh84)

[Guice (Java/Android)](#h.gadnzbm7czmn)

[Dagger (Java/Android)](#h.7l3xar1icw50)

[Redstone (Dart)](#h.n9xgy31lisjo)

[DORM (Dart)](#h.8fnb06a9ee6l)

[JavaEE (Java)](#h.qrw9h0hu3ucv)

[Code metadata workaround in JS today](#h.u6yt5uej0aa1)

[AngularJS v1](#h.sqi8jxtv7xrz)

[DI token metadata - $inject Property](#h.ujfhjw7zaequ)

[DI token metadata - Array Notation](#h.qz4tsglczedf)

[Component metadata - Directive Definition Object (DDO)](#h.3kjn5bth99dr)

[Error messages](#h.tmjwcorsjq5)

[express (Node.js)](#h.p6a67wap152f)

[Ember (Yehuda's TC39 preso)](#h.nx9f1l3b3a33)

[JSDoc (wikipedia page)](#h.jcaj7kzwmic)

[Closure Annotated JavaScript (styleguide, docs)](#h.c0ywkt9pltf)

# Background

Frameworks and tools that leverage declarative programming allow substantial reductions in code size and replace awkward configuration languages or other glue code. We've seen that in many, many places, for example Java, Guice, Dart, Closure JavaScript.

For frameworks and tools to be able to support declarative programing, we need the ability to attach metadata to code and ability to inspect the code and metadata without needing to executing code that is being inspected.

Today in JavaScript there are several workaround solutions for achieving these goals, and as we'll discuss later in this document all of them are mutually incompatible, but what's worse, all of them come with different drawbacks because of the current language constraints.

By having a proper language support we could remove all hacks, standardize and improve usability and toolability of JavaScript code.

# Summary of code metadata usage in programming

## Better docs

Purpose: being able to better describe apis so that the docs can be automatically generated.

Examples: @deprecated, @experimental, @expensive, @nullable

## Instructions for tools

Purpose: being able to describe properties of types, properties and functions that can be used by tools for error-checking, code-minification, and more.

Examples: @immutable, @idempotent, @nullable, @nosideeffects

## Declarative meta-programing

### 

### DI

Purpose: Annotations are the best way to specify additional metadata for injectable entities. [di.js](https://github.com/angular/di.js/), [guice](https://github.com/google/guice), [dagger](http://square.github.io/dagger/), [cdi/javaee](https://docs.oracle.com/javaee/6/tutorial/doc/giwhl.html) depend heavily on annotations.

Examples: @inject, @provide, @transient, @singleton

### Schema & entities

Purpose: ORM frameworks like [breeze.js](http://www.getbreezenow.com/), [ember-data](https://github.com/emberjs/data)[[1]](#footnote-0), [javaee entity beans](https://docs.oracle.com/javaee/6/tutorial/doc/giwhl.html) need a way to describe entities and relationships between entities.

Examples: @hasOne, @hasMany, @eagerlyLoaded

### Parallel processing / processing order

Purpose: Annotations are used to calculate the order of processing steps, and also to determine which steps can happen in parallel.

Examples: Google-internal application framework

## Testing

Purpose: provide metadata about tests/specs

Examples: @timeout(5000), @flaky, @disabled, @expensive, @description('should do foo..')

# Examples of code metadata usage (Prior Art)

Code metadata commonly expressed via annotations is used in many languages, most notably in Java and Dart. Following is a list of how these annotations are used in different frameworks / libraries.

## Component frameworks

### 

### [AngularDart](https://angulardart.org) (Dart)

|  |
| --- |
| @Component(  selector: 'my-component'  map: const {  'title': '@title',  'selection': '<=>currentItem',  'on-selection-change': '&onChange'})  class MyComponent {  String title;  var currentItem;  ParsedFn onChange;  } |

### 

### [Polymer](https://www.dartlang.org/polymer/) (Dart)

|  |
| --- |
| @CustomTag('x-pizza') class XPizza extends PolymerElement {  // ...  @published  String get special\_name => readValue(#special\_name);  set special\_name(String newValue) => writeValue(#special\_name, newValue);  // ... } |

## Dependency Injection

Dependency injection is used to describe how an application instances should be assembled. It is a very useful and productive abstraction.

### 

### [di.dart](https://github.com/angular/di.dart) (Dart)

|  |
| --- |
| @Injectable()  class MyService {  MyService(Dependency1 dep1, Dependency2 dep2) {...}  } |

### 

## 

### [Guice](https://github.com/google/guice) (Java/Android)

|  |
| --- |
| @ImplementedBy(MyService)  class MyInterface {  }    @Singleton  @Named('MyService')  class MyService implements MyInterface {  @Inject()  MyService() {...}  } |

### 

### [Dagger](http://square.github.io/dagger/) (Java/Android)

|  |
| --- |
| @Module  class DripCoffeeModule {  @Provides Heater provideHeater() {  return new ElectricHeater();  }    @Provides Pump providePump(Thermosiphon pump) {  return pump;  }  } |

## [Redstone](http://redstonedart.org/) (Dart)

Redstone is a server-side microframework that allows developers to publish functions and classes through a web interface by annotating them.

|  |
| --- |
| @app.Route("/")  helloWorld() => "Hello, World!";  @app.Interceptor(r'/admin/.\*')  adminFilter() {  if (app.request.session["username"] != null) {  app.chain.next();  } else {  app.chain.interrupt(statusCode: HttpStatus.UNAUTHORIZED);  //or app.redirect("/login.html");  }  }  @app.ErrorHandler(404)  handleNotFoundError() => app.redirect("/error/not\_found.html"); |

## [DORM](https://pub.dartlang.org/packages/dorm) (Dart)

Dorm is a client side library, it can he hooked up to a server side ORM implementation such as Hibernate. Dorm makes serverside entities available to client applications.

|  |
| --- |
| // A declaration of an ID field called 'bar' and of type Bar  @Property(BAR\_SYMBOL, 'bar', Bar) // Bar being another [Entity]  @Id() // Indicates that this property is an identity field, you can have multiple Id fields for combined key support  @NotNullable() // We need a value when persisting this [Entity]  @DefaultValue(const Bar('A null Bar')) // the default value when creating a new Bar();  final DormProxy<Bar> \_bar = new DormProxy<Bar>(BAR, BAR\_SYMBOL); // the actual value is proxied  static const String BAR = 'bar';  static const Symbol BAR\_SYMBOL = const Symbol('orm\_domain.TestEntity.bar'); // full path + prop name  Bar get bar => \_bar.value;  set bar(Bar value) => \_bar.value = notifyPropertyChange(BAR\_SYMBOL, \_bar.value, value); |

## [JavaEE](http://www.oracle.com/technetwork/java/javaee/overview/index.html) (Java)

Various pieces of the enterprise application stack leverage annotations. Usage includes DI, ORM, routing and more.

|  |
| --- |
| @Entity  public class Vehicle {  @Persistent  protected String vehicleName = null;  @Getter  public String getVehicleName() {  return this.vehicleName;  }  public void setVehicleName(@Optional vehicleName) {  this.vehicleName = vehicleName;  }  public List addVehicleNameToList(List names) {  @Optional  List localNames = names;  if(localNames == null) {  localNames = new ArrayList();  }  localNames.add(getVehicleName());  return localNames;  }  } |

## Other examples (stuff that needs more research)

Martin Fowler's [explanation of how to do annotations in Ruby](http://martinfowler.com/bliki/RubyAnnotations.html)

Python has both [decorators](https://www.python.org/dev/peps/pep-0318/) and [function annotations](https://www.python.org/dev/peps/pep-3107/)

# Code metadata workaround in JS today

## [AngularJS v1](https://angularjs.com/)

### DI token metadata - $inject Property

|  |
| --- |
| MyController**.$inject = ['dependency1', 'dependency2'];**  function MyController(dependency1, dependency2) {}  myApp.**controller**('MyController', MyController); |
| // Could become  **@Inject**(**'dependency1'**, **'dependency2'**)  **@Controller**  function MyController(dependency1, dependency2) {} |

Issues with the current approach:

* relies on function declaration hoisting in order to prefix the function with annotation (which surprises developers and causes code-style/linting tools to emit warnings)
* monkey-patches functions with proprietary properties

### DI token metadata - Array Notation

|  |
| --- |
| var MyController = **['dependency1', 'dependency2',**  function MyController(dependency1, dependency2) {}**]**;  myApp.**controller**('MyController', MyController);  // or  myApp.**controller**('MyController', **['dependency1', 'dependency2',**  function MyController(dependency1, dependency2) {}**]**) |
| // Could become  **@Inject**(**'dependency1'**, **'dependency2'**)  **@Controller**  function MyController(dependency1, dependency2) {} |

Issues with the current approach:

* conceals the true identity of functions by wrapping them into arrays (this requires all code that deals with this function to first unwrap it)
* since metadata is not directly associated with the function, it's possible for a function to lose its metadata or have a single function annotated with duplicate and possibly conflicting metadata
* if access to the raw function is required in the current lexical scope, the function can't be naturally defined inline of the array, or it must be unwrapped and re-exported.

### Component metadata - Directive Definition Object (DDO)

|  |
| --- |
| // DDO example:  myApp.directive('myDirective', function() {  **templateUrl: 'my-directive.html',**  **templateNamespace: 'svg',**  **otherOption: someValue,**  controller: SomeController  }); |
| // Combined with ES6 class syntax could become:  **@component({**  **templateUrl: 'my-directive.html',**  **templateNamespace: 'svg',**  **otherOption: someValue**  **})**  class MyDirective {  } |

Issues with the current approach:

* conceals the true identity of functions by wrapping them into objects (this requires all code that deals with this function to first unwrap it)
* since metadata is not directly associated with the function, it's possible for a function to lose its metadata or have a single function annotated with duplicate and possibly conflicting metadata
* if access to the raw function is required in the current lexical scope, the function can't be naturally defined inline, or it must be unwrapped and re-exported.

|  |
| --- |
| // DDO + DI example:  myApp.directive('myDirective', function() {  **templateUrl: 'my-directive.html',**  **templateNamespace: 'svg'**  controller: **['dependency1', 'dependency2',** function SomeController(dependency1, dependency2) {}**]**  }); |
| // Combined with ES6 class syntax could become:  **@component({**  **templateUrl: 'my-directive.html',**  **templateNamespace: 'svg'**  **})**  **@inject(Dependency1, Dependency2)**  class MyDirective {  constructor(dependency1, dependency2) {}  } |

### Error messages

We provide verbose error messages to developers in dev build but strip the error messages and replace them with links to docs in production builds. This saves us a significant chunk of payload in production.

|  |
| --- |
| **throw** **minErr**('$controller')('noscp',  "Cannot export controller '{0}' as '{1}'! No $scope object provided via `locals`.",  name, identifier); |
| @**minifiable**  class ControllerError extends Error{  constructor(message) {  this.message = message;  }  };  **throw new** **ControllerError**(`Cannot export controller '${name}' as '${identifier}'! No $scope object provided via `locals`.`); |

Issues with the current approach:

* it's very hard to safely minify all error messages because regular static analysis is not sufficient to determine all call sites where we throw a minifiable error (we rely on code conventions and code reviews to get this right, but it's very fragile)
* the code is very unnatural and requires a lot of domain specific knowledge to write properly.

## [express](http://expressjs.com/) (Node.js)

|  |
| --- |
| // Current API  var express = require('express')  var app = express()  app.get('/', function (req, res) {  res.send('Hello World!')  })  var server = app.listen(3000, function () {  var host = server.address().address  var port = server.address().port  console.log('Example app listening at http://%s:%s', host, port)  }) |
| // Possible API with annotation  @server({  port: 3000  })  class Application {    @get('/')  getHelloWorld(req, res) {  res.send('Hello World!')  }  @on('start')  onStart(event) {  console.log(  'Example app listening at http://%s:%s', event.host, event.port)  }  }  express().start(Application); |

Issues with the current approach:

* tries to be declarative via imperative code constructs and conventions

## Ember ([Yehuda's TC39 preso](https://github.com/rwaldron/tc39-notes/raw/master/es6/2014-04/Decorators.pdf))

|  |
| --- |
| App.Person = Ember.Object.extend({  firstName: null,  lastName: null,  fullName: function() {  return this.get('firstName') + ' ' + this.get('lastName');  }.**property('firstName', 'lastName')**,  fullNameChanged: function() {  // deal with the change  }.**observes('fullName')**.**on('init')**  }); |
| // alternative implementation  App.Person = Ember.Object.extend({  firstName: null,  lastName: null,  fullName: **Em.computed('firstName', 'lastName',** function() {  return this.get('firstName') + ' ' + this.get('lastName');  }**)**,  fullNameChanged: **Em.observes('fullName',**  **Em.on('init',** function() {  // deal with the change  })  )  }); |
| // possible implementation with annotations and ES6 classes  class Person extends Ember.Object {    **@dependsOn('firstName', 'lastName')**  get fullName() {  return this.get('firstName') + ' ' + this.get('lastName');  }**,**  **@on('init')**  **@observes('fullName')**  fullNameChanged() {  // deal with the change  }  }; |

TODO: use examples from Yehuda's presentation

Issues with the current approach:

* monkey-patching global prototypes
* metadata follows the code that is being annotated which decreases readability

## 

## JSDoc ([wikipedia page](http://en.wikipedia.org/wiki/JSDoc))

|  |
| --- |
| /\*\*  \* Creates an instance of Circle.  \*  \* @constructor  \* @this {Circle}  \* @param {number} r The desired radius of the circle.  \*/  function Circle(r) {  /\*\* @private \*/ this.radius = r;  /\*\* @private \*/ this.circumference = 2 \* Math.PI \* r;  }      /\*\*  \* Calculates the circumference of the Circle.  \*  \* @deprecated  \* @this {Circle}  \* @return {number} The circumference of the circle.  \*/  Circle.prototype.calculateCircumference = function () {  return 2 \* Math.PI \* this.radius;  }; |
| // possible API with annotations and ES6 class syntax  @description('Creates an instance of Circle.')  @this(Circle)  @param(Number, 'The desired radius of the circle.')  class Circle(r) {  constructor(r) {  @private  this.radius = r;  @private this.circumference = 2 \* Math.PI \* r;  }      /\*\*  \* Calculates the circumference of the Circle.  \*  \* @this {Circle}  \* @return {number} The circumference of the circle.  \*/  @deprecated  Circle.prototype.calculateCircumference = function () {  return 2 \* Math.PI \* this.radius;  }; |

Issues with the current approach:

* all annotations are in the comments
* creating custom annotations is non-trivial
* low interoperability between tools
* inability to inspect or introspect metadata at runtime

## Closure Annotated JavaScript ([styleguide](https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml), [docs](https://developers.google.com/closure/))

|  |
| --- |
| /\*\*  \* A shape.  \* **@interface**  \*/  function Shape() {};  Shape.prototype.draw = function() {};  /\*\*  \* @constructor  \* **@implements** {Shape}  \*/  function Square() {};  Square.prototype.draw = function() {  ...  };  /\*\*  \* **@private**  \*/  Square.prototype.privateMethod = function() {  ...  }; |
| // possible API with annotations and es6 classes  /\*\*  \* A shape.  \*/  **@interface**  function Shape() {};  Shape.prototype.draw = function() {};  @**implements** {Shape}  class Square {  draw() => …  **@private**  privateMethod() => ...  }; |

Issues with the current approach:

* all annotations are in the comments
* creating custom annotations is non-trivial
* low interoperability between tools
* inability to inspect or introspect metadata at runtime

1. It is our understanding that ember-data authors currently prefer achieving the meta-programing goals via decorators rather than via pure static metadata. [↑](#footnote-ref-0)