Outline

- The purpose of models
- Defining new model types
- Instantiating models
- Working with model properties
- Model events
- Model identity
- Defaults
- Validation

The Purpose of Models

- Models form the core of your application
- They contain your application's state as well as logic and behavior
- Models are the single point of truth for data.
- Models provide a lifecycle
- They communicate changes to the rest of the application via events.

Defining New Model Types

- Create new Model 'types' by extending Backbone.Model

```
var Vehicle = Backbone.Model.extend({});
```

Use uppercase for type names

extend() is a function shared by Model, Collection, Router and View. It establishes an inheritance relationship between two objects.

Defining New Model Types (cont.)

- Model types can have ‘class properties’ as well

```
var Vehicle = Backbone.Model.extend({},
    {
        summary: function () {
            return 'Vehicles are
                    for travelling';}
    }
);

Vehicle.summary();
```

Instantiating Models

- To create a new model object call its constructor function with the 'new' operator
- The simplest case is to create an instance of Backbone.Model

```
var model = new Backbone.Model();
```

- Or use custom types

```
var Vehicle = Backbone.Model.extend({});
var ford = new Vehicle();
```

Instantiating Models (cont.)

- **Instantiate with property values**

```
var model = new Backbone.Model({
     name: ‘Peter’,
     age: 52
});
```

Initialize

- **If a model type has an 'initialize' function defined it will be called when the model is instantiated.**

```
var Vehicle = Backbone.Model.extend({
    initialize: function () {
        console.log('vehicle created');
    }
});

var car = new Vehicle();
// vehicle created
```

Model Inheritance

- Models can inherit from other models

```
var Vehicle = Backbone.Model.extend({});

var Car = Vehicle.extend({});
```

Working with Model Attributes

- Attributes can be set by passing an object to a model type's constructor, or by using the 'set' method

```
var ford = new Vehicle();
ford.set('type', 'car');
```

- Or set many properties at once

```
ford.set({
    'maximumSpeed': '99',
    'color': 'blue'
});
```

Working with Model Attributes (cont.)

- Read attributes with the ‘get’ method

```
ford.get(‘type’);
// car
```

- ‘Escape’ is like ‘get’ except that the output is html escaped

```
ford.set(‘description’,
  ‘<script>alert(“script injection”)</script>’);
ford.escape(‘description’);
```

Test for an Attribute

- Use ‘has’ to test if an attribute has been defined

```
var ford = new Vehicle();
ford.set(‘type’, ‘car’);

ford.has(‘type’);
// true

ford.has(‘year’);
// false
```

Model Events

- Models raise events when their state changes
- To detect a change to a model listen for the 'change' event

```
ford.on('change', function () {});
```

- Or listen to a change to a property

```
ford.on('change:color', function () {});
```

Custom Model Events

- It is possible to define, trigger and observe custom model events
- Events are identified by string identifiers
- Use the ‘on’ method to bind to an event

```
ford.on(‘retired’, function () {});
```

- Use the ‘trigger’ method to trigger an event

```
ford.trigger(‘retired’);
```

Model Identity

- **The 'id' property represents the model's persistent identity. It is undefined until the model has been saved.**

```
var ford = new Vehicle();
ford.id;
// undefined
```

Model Identity (cont.)

- The ‘cid’ property is a temporary identifier used until a model is assigned its ‘id’.

```
var ford = new Vehicle();
ford.cid;
// c1
```

Defaults

- The ‘defaults’ property specifies default values for attributes that are not set in the constructor

```
var Vehicle = Backbone.Model.extend({
    defaults: {
        ‘color’: ‘white’,
        ‘type’: ‘car’
    }
});

var car = new Vehicle();
car.get(‘color’); // white
car.get(‘type’);  // car
```

Validation

- **Backbone exposes model validity through two methods**
  - validate
  - isValid
- **Validate is called by backbone prior to performing ‘set’ or ‘save’ operations**

toJSON

- **Converts a model's attributes to a JavaScript object**

```
var ford = new Vehicle();
ford.set('type', 'car');
ford.toJSON(); // { type: 'car' }
```

save, fetch, destroy

- Models have save, fetch and destroy methods for synchronizing with the server
- Save performs insert and update operations, depending upon the state of the model
- Fetch updates the model with the server-side state
- Destroy deletes the model from the server

Summary

- Models hold your applications data
- get / set
- new / initialize
- Events
- Identity
- Defaults
- Validation
- toJSON()
- Save, fetch and destroy