**Intro To Databases**

Mongoose is a Node package that interacts with a running MongoDB database. Before learning about Mongoose, let’s define a couple of key terms:

* *Data*
* *Database*

**What is Data?**

Data in the context of software and web development is digital information.

**What is a Database?**

A database is a structured set of data held in a computer.

Databases support storage and manipulation of data. For a web application to have persistence, a developer uses a database to store data. The developer can write methods to create, read, update, and delete information in the database.

# Mongoose Collections & Documents

Mongo stores data in ‘binary’ JSON (BSON) documents. BSON documents have a similar structure to JavaScript objects.

MongoDB stores documents in a collection. A MongoDB database is made up of these collections of documents.

A Mongo collection is like a table in a spreadsheet or relational database — each document is like a row in the spreadsheet.

Documents contain one or more key/value pairs. Each key has a corresponding value of a specified data type, like array, number, or string. MongoDB organizes documents with similar structure into collections.

**Schemas**

Mongoose is a JavaScript library that provides methods to interact with a MongoDB database. Mongoose translates JavaScript objects (JSON) to BSON data in a MongoDB database, and vice versa.

Mongoose interactions are based on Schema and Model declarations.

* A *Schema* defines the shape of the documents within that collection.
* A *Model* maps to a MongoDB collection and its documents.

**Mongoose Schema**

Remember, each record in a MongoDB database is a document with key/value pairs as entries. Using Mongoose’s Schema, we can set the structure of those documents dynamically.

Imagine you were creating a Schema for the database of a web-based poetry application where you could both write and publish poems. Each key in our poemSchema will define a property in our documents which will be cast to its associated SchemaType.

const poemSchema = new mongoose.Schema({

title: String

})

Each document that is derived from the poemSchema above will have a title property with a string saved to it.

Mongoose will cast mismatched data types to the specified SchemaType. For example, if we entered the number 1 as a title for a poem, Mongoose would cast the entry so that it entered that database as a string "1". Using casting, Mongoose ensures that string properties are assigned strings values.

**Instructions**

**1.**

Imagine you are developing a role-playing game where users will be able to access magic items from a database. Create a Schema for the following magic items:

* Create a const variable named magicItemSchema
* Using new mongoose.Schema(), set magicItemSchema to create a path with the name item and the schema type for item to String

Hint

You can use the following structure to achieve this:

const magicItemSchema = new mongoose.Schema({

path: value

})

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: String

})

**Paths**

The key-value pair in a schema is called a path. Paths define the name and type of fields in a MongoDB document.

const poemSchema = new mongoose.Schema({

title: String,

body: [String],

published: Boolean,

})

The schema above has three paths: title, body, and published. Each path is set to a different schema type. The [String] schema type, assigned to body, means a document that is derived from the poemSchema schema can store an array of strings to the body field.

[Paths in Mongoose can have many data types](http://mongoosejs.com/docs/schematypes.html). Besides data types like string, integer, boolean, and array, Mongoose also provides:

* Timestamp − timestamp. This can be handy for recording when a document has been modified or added.
* Object ID − This datatype is used to store the document’s ID.

**Instructions**

**1.**

Add the following paths and matching SchemaTypes to magicItemSchema:

* magicalProperty its schemaType is String
* unitCost its schemaType is Number
* totalUnits its schemaType is Number

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: String,

  magicalProperty: String,

  unitCost: Number,

  totalUnits: Number

})

# Validators

In our original poetry application example we declared the schema type for our title as a String:

const poemSchema = new mongoose.Schema({

title: String

})

Often, we want to specify more than just the type of a path — we can use validators to ensure other aspects of a document’s input value.

### Validators

Data validation is intended to provide guarantees about user input. Mongoose has several built-in validators.

You can add required validators to our Schema in an object that you pass to the path:

const poemSchema = new mongoose.Schema({

title: {

type: String,

required: true

},

body: {

type: [String],

required: true

},

published: {

type: Boolean,

required: true

},

})

In the example above, we set the schema type and required attributes for the title, body, and published paths. If the required property is true, then it is a required field when you save to the database.

If you save a document with an invalid path value, you will receive this error message Path `title` is required.. You can define a custom error message like this:

const poemSchema = new mongoose.Schema({

title: {

type: String,

required: 'Title is required!'

}

})

[You can learn more about mongoose validators in their documentation.](http://mongoosejs.com/docs/validation.html)

**Instructions**

**1.**

Refactor your schema to use the object syntax, and add a required validator with a value of true to each of the paths.

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: {

    type: String,

    required: true

  },

  magicalProperty: {

    type: String ,

    required: true

  },

  unitCost: {

    type: Number,

    required: true

  },

  totalUnits: {

    type: Number,

    required: true

  }

})

**Models**

To use our poemSchema definition:

const poemSchema = new mongoose.Schema({

title: String,

});

we need to convert our poemSchema into a Model we can work with. Schemas provide the definition for a model. A model maps to a collection in your MongoDB database.

Models are defined by passing a Schema instance to mongoose.model like this:

mongoose.model(modelName, schema):

The first argument is the singular name of the collection your model is for. The second argument is your previously created Schema. In the case of our poetry web app, turning our schema into a model would look like this:

const Poem = mongoose.model('Poem', poemSchema);

Models are constructors that we define based on our Schema. They represent documents which can be saved and retrieved from our database. All document creation and retrieval from the database is handled by these models.

**Instructions**

**1.**

Use mongoose.model() to create a model named MagicItem with the magicItemSchema Schema.

Hint

* Create a const variable named MagicItem and set it equal to mongoose.model('MagicItem', nameOfSchema);

Replace ‘nameOfSchema’ with the name of your schema.

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: {

    type: String,

    required: true

  },

  magicalProperty: {

    type: String ,

    required: true

  },

  unitCost: {

    type: Number,

    required: true

  },

  totalUnits: {

    type: Number,

    required: true

  }

})

const MagicItem = mongoose.model('MagicItem', magicItemSchema);

**Create**

Our model is a class with properties that we define in our schema. We can construct documents as instances of our model. Creating documents and saving them to the database can be done by calling .create() on our model. In the poetry app example it would look like this:

const Poem = mongoose.model('Poem', poemSchema);

const poemProperties = {

title: "Rewrite Reality" ,

body: ["Re-imagine the consumption of the stagnant status quo",

"No matter how nice you dress",

"The emperor is still wearing no clothes"],

published: false

}

runWithDatabase(async () => {

// Create and save a document

await Poem.create(poemProperties);

});

The runWithDatabase function is designed to accept a method as input, and run it after we connect to a database and before we disconnect from it. If you want to learn more about the methods in **database.js**, read the documentation for:

* [Connect](http://mongoosejs.com/docs/api.html#index_Mongoose-connect)
* [Drop](https://docs.mongodb.com/manual/reference/method/db.dropDatabase/)
* [Disconnect](http://mongoosejs.com/docs/api.html#index_Mongoose-disconnect)

This would create a new document in our database, with the paths and properties defined in the code above. In the next exercise, we will learn how to query the database and confirm our application is successfully sending and retrieving information from our MongoDB collection of documents.

**Instructions**

**1.**

Create an entry in a database. Create a const variable named properties and set it equal to an object with the paths:

* item: "cloak"
* magicalProperty: "invisibility"
* unitCost: 25
* totalUnits: 100

Inside runWithDatabase(), call .create() on MagicItem and pass the properties variable as an argument.

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: {

    type: String,

    required: true

  },

  magicalProperty: {

    type: String ,

    required: true

  },

  unitCost: {

    type: Number,

    required: true

  },

  totalUnits: {

    type: Number,

    required: true

  }

})

const MagicItem = mongoose.model('MagicItem', magicItemSchema);

const properties = {

  item: "cloak" ,

  magicalProperty: "invisibility" ,

  unitCost: 25 ,

  totalUnits: 100

}

runWithDatabase(async () => {

  await MagicItem.create(properties)

})

**Queries**

At this point, we will start creating instances in a MongoDB database, then query the database for the values we saved. All of the method calls and queries will be passed to runWithDatabase().

If we wanted to search for the poem that we saved to the database, we could write a Mongoose query and call .findOne() on our Poem model:

runWithDatabase(async () => {

Poem.create(poemProperties)

const poemMatch = await Poem.findOne({ title: 'Rewrite Reality' });

console.log(`Found poem: ${poemMatch.body}`);

});

.findOne() returns a document that has a title path with the value 'Rewrite Reality'. We confirm this by using console.log() to see the value of the path body for the returned document.

This is good when we are looking for one document. What if we wanted to find all the documents that matched a specified criteria? We can use .find(), which returns an array of all the documents that match the argument passed to it.

Imagine we had a collection of documents with the following values:

const manyPoems = [

{

title: "Rewrite Reality" ,

body: ["Re-imagine the consumption of the stagnant status quo",

"No matter how nice you dress",

"The emperor is still wearing no clothes"],

published: false ,

},

{

title: "Phasing",

body: ["I imagine myself a second moon;",

"waning in and out of sequence with the tides"],

published: true,

},

{

title: "One of Those Days",

body: ["My shirt and spirit”,

"are bent inside-out-backwards"],

published: true,

}

]

In the example below, we create the above documents using a Poem model, then we use the find method to return all poems with a published property equal to true:

runWithDatabase(async () => {

Poem.create(manyPoems);

let publishedPoems = await Poem.find({ published: true })

// Check that it works by logging the number of returned documents

console.log(`Published Poems: ${publishedPoems.length}`)

});

To the right, we’ve added **items.js** to your project. It contains an array of items that we will save to the MongoDB database. In **exercise.js**, we import the items from **items.js** and use MagicItem.create() to save the items as documents to our database.

**Instructions**

**1.**

Follow the steps below to write a query using .findOne() to locate the document in your database with the item: "cloak".

* Inside runWithDatabase() and below .create(), add a let variable named finder and set it equal to await and then your query.
* Before running your script in the terminal, add this on the following line:

console.log(`Found one: ${finder.item}`);

* Run your database query using the following command in the terminal: node exercise.js.

Checkpoint 2 Passed

Hint

You can use the following example as a model for the syntax you would use:

let finder = await ModelName.findOne({ pathName: value });

**2.**

Add a second query to your code, which uses .find() to locate all magicObject documents that have a unitCost less than 50.

You can use the mongoose query operator $lt. $lt selects the documents where the value of the field is less than (i.e. <) the specified value. The syntax is:

.find({field: {$lt: value} }).

* Create a let variable named cheapObjects and set it equal to await and then your query.
* Before running your script in the terminal, you can add this line after your code:

console.log(`Found ${cheapObjects.length} magic objects`);

* Run your database query using the following command in the terminal: node exercise.js.

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: {

    type: String,

    required: true

  },

  magicalProperty: {

    type: String ,

    required: true

  },

  unitCost: {

    type: Number,

    required: true

  },

  totalUnits: {

    type: Number,

    required: true

  }

})

const MagicItem = mongoose.model('MagicItem', magicItemSchema);

runWithDatabase(async () => {

  await  MagicItem.create(manyItems);

  let finder = await MagicItem.findOne({ item: 'cloak' });

  console.log(`Found one: ${finder.item}`);

  let cheapObjects = await MagicItem.find({ unitCost: { $lt: 50 }})

  console.log(`Found ${MagicItem.length} magic objects`);

});

**Methods**

Mongoose supports the creation of methods on both instances of documents and collections of documents (the model).

* .statics() adds static “class” methods to the model.
* .methods() adds an instance method to documents.

**Model Methods — .statics()**

For example, in our poetry app we could use .statics() to create a method named firstAlphabetically.

const poemSchema = new mongoose.Schema({

...

)}

poemSchema.statics.firstAlphabetically = function(callback) {

return this.findOne({}).sort('title').exec(callback);

}

The method is part of the model, which in this example would return the first document, after sorting the values of the title path alphabetically.

In order to use the new .firstAlphabetically method, we would call it inside runWithDatabase() on the Poem model like this:

const Poem = mongoose.model('Poem', poemSchema);

runWithDatabase(async () => {

Poem.create(properties));

const firstAlpha = await Poem.firstAlphabetically();

console.log(`The first poem alphabetically is: ${firstAlpha.title}. It goes like this: ${'\n'} ${firstAlpha.body}`);

});

The console.log() logs the following in the terminal:

The first poem alphabetically is: One of Those Days. It goes like this:

My shirt and spirit, are bent inside-out-backwards

**Document Methods — .methods()**

Instances of a model are documents. Documents have many of their own built-in instance methods. It is also possible to create custom document instance methods. Below, we create a method for our poem example to change the published path value to true on any document in the database.

const poemSchema = new mongoose.Schema({

...

)}

poemSchema.methods.publish = function(callback) {

this.published = true

return this.save();

}

.save() writes the current JavaScript object as a MongoDB document.

Inside runWithDatabase() we could call .publish() on a document like this:

const Poem = mongoose.model('Poem', poemSchema);

runWithDatabase(async () => {

Poem.create(properties));

...

const publishIt = await Poem.findOne({ title: 'Rewrite Reality' });

console.log(publishIt.published)

await publishIt.publish();

console.log(`${publishIt.title} has had its publish field changed to ${publishIt.published}`)

});

Here we use .findOne to locate the poem, and then console.log() the initial publish value, which is false. We call .publish() on it and console.log() again to see that it changed. The output in the terminal for this code would look this:

false

Rewrite Reality has had its publish field changed to true

**Instructions**

**1.**

Create a static method for your schema called findMostExpensive and set it equal to a function with a callback argument. Add this in the function’s body:

return this.findOne({}).sort('unitCost').exec(callback);

Create a method for your schema using the syntax magicItemSchema.methods.use and set it equal to a function with a callback argument. Add this in the function’s body:

this.totalUnits -= this.unitCost;

return this.save();

Checkpoint 2 Passed

**2.**

You can add the following code in runWithDatabase(), then enter node exercise.js in the terminal to see these methods in action:

const mostExpensive = await MagicItem.findMostExpensive();

console.log(`The most expensive object is the ${mostExpensive.item}`);

console.log(`The ${mostExpensive.item} started with ${mostExpensive.totalUnits} charges.`);

console.log(`Using ${mostExpensive.item}...`);

await mostExpensive.use();

console.log(`The ${mostExpensive.item} has ${mostExpensive.totalUnits} charges left.`);

const {mongoose, runWithDatabase} = require('./database');

const manyItems = require('./items');

const magicItemSchema = new mongoose.Schema({

  item: {

    type: String,

    required: true

  },

  magicalProperty: {

    type: String ,

    required: true

  },

  unitCost: {

    type: Number,

    required: true

  },

  totalUnits: {

    type: Number,

    required: true

  }

})

magicItemSchema.statics.findMostExpensive = function(callback) {

  return this.findOne({}).sort('unitCost').exec(callback);

}

magicItemSchema.methods.use = function(callback) {

  this.totalUnits -= this.unitCost;

  return this.save();

}

const MagicItem = mongoose.model('MagicItem', magicItemSchema);

runWithDatabase(async () => {

  await  MagicItem.create(manyItems);

  const mostExpensive = await MagicItem.findMostExpensive();

  console.log(`The most expensive object is the ${mostExpensive.item}`);

  console.log(`The ${mostExpensive.item} started with ${mostExpensive.totalUnits} charges.`);

  console.log(`Using ${mostExpensive.item}...`)

  await mostExpensive.use();

  console.log(`The ${mostExpensive.item} has ${mostExpensive.totalUnits} charges left.`);

  let finder = await MagicItem.findOne({ item: 'cloak' });

    console.log(`Found one: ${finder.item}`);

    let cheapObjects = await MagicItem.find({ unitCost: { $lt: 50 }})

    console.log(`Found ${MagicItem.length} magic objects`);

});

# Review

In this lesson, you learned how to use Mongoose to interact with a MongoDB database. Let’s review some of the topics that we covered:

* Mongoose is a Node package that interacts with a running MongoDB database.
* MongoDB stores documents in collections and collections of documents in databases. Each document has key-value pairs as entries.
* Using a Schema, we can set the structure of documents dynamically, using paths with schema types and validators.
* Models are JavaScript classes that we compile from our Schema definitions.
* You can use models to create, read, update, and delete documents from a database.
* You can query a database using .find() and .findOne(). [Mongo also provides query operators to allow for more complex queries.](https://docs.mongodb.com/v3.4/reference/operator/query-comparison/)

Mongoose also allows for the creation of methods associated with a database:

* .statics() adds static “class” methods to the Models itself.
* .methods() adds an instance method to documents.

**Introduction**

A chat app manages messages, users, and chat rooms. A restaurant app manages customers, tables, and orders. How do applications define these entities and their interactions? The answer is the *model layer*.

Models represent the entities and interactions in a web application’s problem domain: the area of knowledge surrounding a problem. A chat app’s problem domain includes messages, users, and chat rooms; a restaurant’s includes customers, tables, and orders. A model can define each entity, describe the shape of the data stored for each entity, validate the data, store it in a database, and interact with it.

In this lesson, you will learn TDD techniques to develop a model layer using JavaScript with the Mongoose node package and a MongoDB database. For testing, you’ll be using a Mocha test framework and the Chai assertion library.

To better understand the concept of a *model*, take this example: a full-stack web application manages the inventory of a zoo. It can add animals, remove animals, count animals, and store that information for later use. The app can be divided into three layers:

* Front-end: a webpage with buttons to allow users to add and remove animals. Could be implemented with HTML and CSS.
* Server: an application to handle HTTP requests and responses. It routes requests, like the addition and removal of animals, and defines responses, like the count of animals after addition/removal. Could be implemented with Express.
* Database and Models: storage and shape of the animal data. The data is grouped by animal, each with properties like species name, count, and risk-level. These fields and the methods to interact with them are defined by models, and the storage is managed by a database. Could be implemented with Mongoose models and a MongoDB database.

Just like any other software, you can develop models using Test-Driven Development (TDD). The following exercises will help you write tests specific to the model layer.

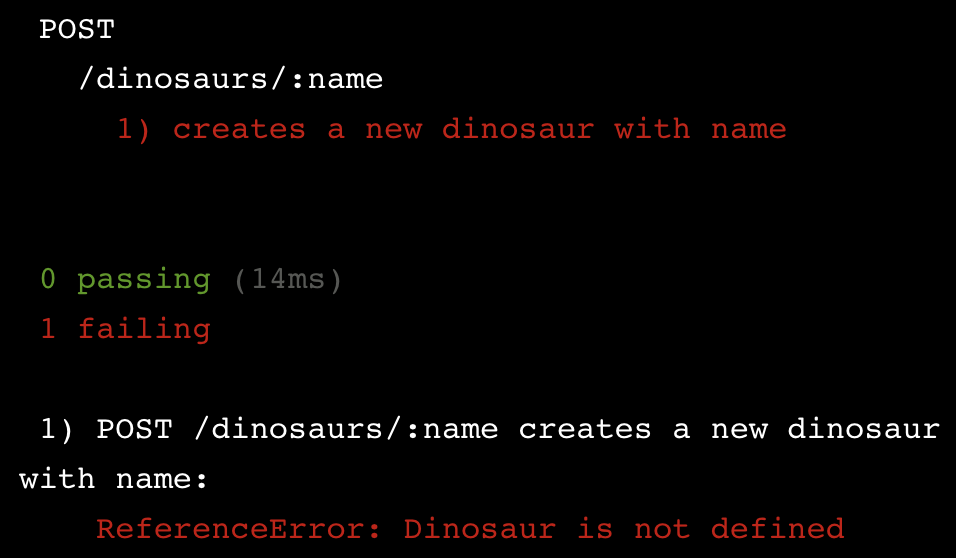
**Instructions**

The zoo for this application keeps a particular type of animal. Get familiar with the data displayed on the right. You will be implementing a model for that data throughout this lesson.



**Path Definition**

In test-driving the zoo application, you receive this error message in the server layer:



You’re in the red! To get to green you have to drop to the model layer and define the Dinosaur model.

You’ll need multiple model tests to satisfy this server test. Since they don’t touch HTML/CSS selectors nor HTTP actions/status codes, model tests are typically faster than feature-level and server-level tests. Driving the Dinosaur implementation with model tests — rather than feature or server tests — will make your test suite run faster. The model tests will confirm that:

1. the Dinosaur model is defined
2. the Dinosaur model has a path called name

Your first test will cover conditions 1 and 2 by creating an instance of a Dinosaur model with a name, then asserting that the name path (also referred to as field or property) can be retrieved.

You can review the [Mongoose guide](http://mongoosejs.com/docs/guide.html) for defining a schema and the model documentation for creating models and instances.

**Instructions**

**1.**

Write your first model test! In **dinosaur-test.js** under it('is a String', construct a new instance of the Dinosaur model. Use the constructor new Dinosaur() and store the result in a variable dino.

Checkpoint 2 Passed

**2.**

Run the test and see the error message

ReferenceError: DinosaurSchema is not defined

Checkpoint 3 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**3.**

The error directs you to define a Dinosaur schema. In **dinosaur.js** define a Dinosaur schema named DinosaurSchema without any paths. Use the constructor new Schema().

Checkpoint 4 Passed

**4.**

Run the test and see green!

Checkpoint 5 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**5.**

But the test doesn’t check the name path. In **dinosaur-test.js** within the Dinosaur constructor, add a name path with value 'T-rex'.

Don’t forget to use braces { }.

Checkpoint 6 Passed

Hint

If you need help constructing a model instance with paths, use the Mongoose documentaton at: [mongoosejs.com/docs/models.html](http://mongoosejs.com/docs/models.html).

**6.**

Assert that the name of the dino is strictly equal to 'T-rex'.

Use Chai’s [assert.strictEqual(actual, expected)](http://chaijs.com/api/assert/#method_strictequal) method.

Checkpoint 7 Passed

Stuck? Get a hint

**7.**

Run the test and see the error message

AssertionError: expected undefined to equal 'T-rex'

Checkpoint 8 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**8.**

The error directs you to define a name path. In **dinosaur.js** define a name path in the Schema constructor of type String.

Don’t forget to use braces { }.

Checkpoint 9 Passed

Hint

If you need help defining a schema, use the Mongoose guide at: [mongoosejs.com/docs/guide.html](http://mongoosejs.com/docs/guide.html).

**9.**

Run the test and see green!

Checkpoint 10 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

const Dinosaur = require('../../models/dinosaur');

const {assert} = require('chai');

const {mongoose, databaseUrl, options} = require('../../database');

describe('Dinosaur', () => {

  describe('#name', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex'

      });

      assert.strictEqual(dino.name, 'T-rex')

    });

  });

});

**Hooks**

Before getting any further, let’s recall some good TDD practices:

* make your tests *expressive* by writing them in four phases
* make your tests *isolated* with setup and teardown phases
* follow the [red, green, refactor cycle](https://www.codecademy.com/articles/tdd-red-green-refactor)

In this lesson you will be writing your setup and teardown phases in beforeEach and afterEach hooks provided by [Mocha](https://mochajs.org/).

Before each test, your beforeEach hook will [connect](http://mongoosejs.com/docs/api.html#index_Mongoose-connect) to the database and [drop](https://docs.mongodb.com/manual/reference/method/db.dropDatabase/) any old data using these method calls:

await mongoose.connect(databaseUrl, options);

await mongoose.connection.db.dropDatabase();

After each test, your afterEach hook will [disconnect](http://mongoosejs.com/docs/api.html#index_Mongoose-disconnect) from the database with

await mongoose.disconnect();

You can refactor these hooks by wrapping the three calls in two helper functions: connectAndDrop and disconnect. In your test file, import those functions and add them to your hooks.

**Instructions**

**1.**

A #save test has been added to further drive development. Run the tests by entering npm test in the terminal.

Checkpoint 2 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**2.**

You should get a Timeout error — Mocha is waiting for a connection that doesn’t exist yet. Add this under the first describe:

beforeEach( async () => {

await mongoose.connect(databaseUrl, options);

});

afterEach( async () => {

await mongoose.disconnect;

});

Checkpoint 3 Passed

**3.**

Run the tests and see green.

Checkpoint 4 Passed

**4.**

Run the tests again and see red!

AssertionError: expected 2 to equal 1

This error is telling you that there are 2 documents found, when 1 was expected. Your test is not *isolated* — the document added in the previous test run still exists in the database.

Checkpoint 5 Passed

**5.**

In the beforeEach hook after the call to connect, add

await mongoose.connection.db.dropDatabase();

This drops any old data in the database.

Checkpoint 6 Passed

**6.**

Run the test suite and see green!

Checkpoint 7 Passed

**7.**

Time to refactor. The two helper functions connectAndDrop and disconnect have been defined for you in **database.js**. Import the functions at the top of **dinosaur-test.js** by replacing {mongoose, databaseUrl, options} with {connectAndDrop, disconnect}.

Checkpoint 8 Passed

**8.**

Delete the contents of the beforeEach hook, including async () => {...}.

Pass the argument connectAndDrop to the beforeEach function.

Checkpoint 9 Passed

**9.**

Delete the contents of the afterEach hook, including async () => {...}.

Pass the argument disconnect to the afterEach function.

Checkpoint 10 Passed

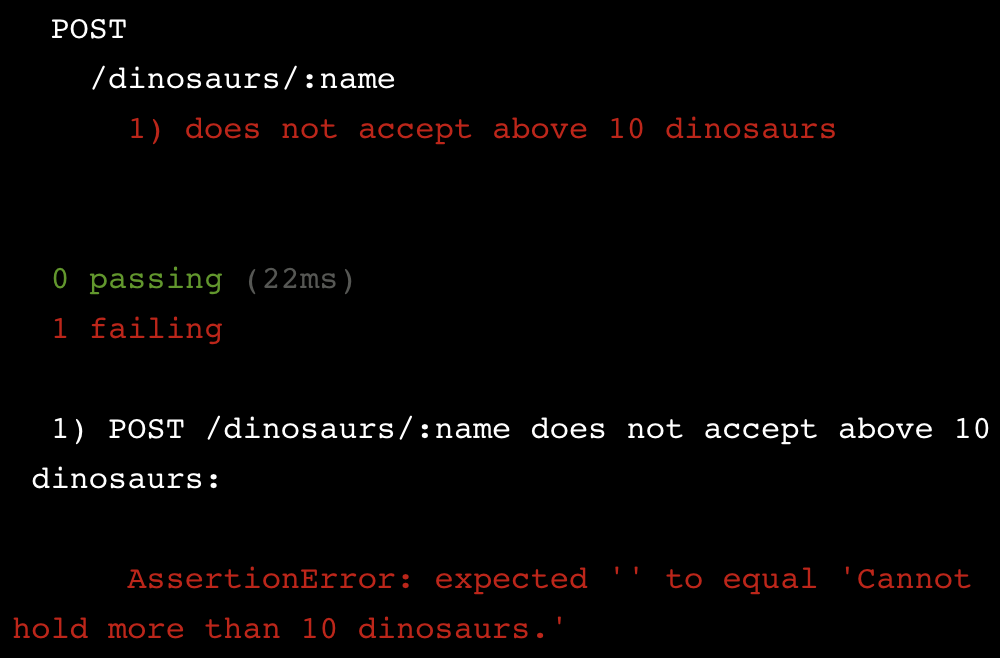
**10.**

Run the tests to confirm that you’re still in the green.

**Path Validation I**

According to the Department of Dino Zoo Control (DDZC), the zoo isn’t safe with more than 10 of any one type of Dinosaur.

At this point your Dinosaur model has name, count, and risk properties and you have some server tests written. One of those tests expects a warning message when adding more than 10 dinosaurs, but you receive the following error message:



This error occurred because the Dinosaur model has no validation! You can ensure the safety of the zoo by adding a validator function.

In this exercise you’ll be using a [custom validator function](http://mongoosejs.com/docs/api.html#schematype_SchemaType-validate). It receives the value to validate as its first argument. It returns a Boolean, which is false if the value fails validation. Avoid arrow notation () =>. Using function() notation preserves the proper binding of this. Here’s the syntax:

// Define validator

validate = function (value) {

...

}

// Add validator to Schema

const DinosaurSchema = new Schema({

count: {

type: Number,

validate: [validator, 'custom err msg']

}

});

Since validation is a model-level concern, you’ll need to test at the model layer. You can test validation like this:

1. Create an instance of a model with validators and execute the validations with the validateSync method. Any validation errors will be stored in [instance].errors.[path], like dino.errors.count.
2. Make assertions on [instance].errors.[path] and its properties.

For more information on validators visit the Validation section of the Mongoose guide: <http://mongoosejs.com/docs/validation.html>.

**Instructions**

**1.**

Using TDD, ensure that a dinosaur cannot have a count of more than 10. The count and risk paths have already been defined for you.

In **dinosaur-test.js** under it('is invalid with 11' construct a new instance of the Dinosaur model with

* name 'T-rex'
* count 11
* risk 'High'

and store it in a variable dino.

Checkpoint 2 Passed

Hint

Create a const variable called dino.

Use the new Dinosaur(obj) constructor where obj is an object with the above key-value pairs.

**2.**

Call dino.validateSync() after that.

Checkpoint 3 Passed

Hint

Call the function within the is invalid with 11 it block.

**3.**

If there are any errors, they will be stored in dino.errors.

* Use assert.ok to confirm that dino.errors exists
* Pass the custom error message 'model should have validation error' as a second argument to assert.ok

Checkpoint 4 Passed

Hint

Call assert.ok within the is invalid with 11 it block.

**4.**

Run npm test and see the error message

AssertionError: model should have validation error: expected undefined to be truthy

Checkpoint 5 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**5.**

The errors object is undefined, which means no error is thrown! In **dinosaur.js** add

validate: validator

to the count property. You’ll define the validator function soon.

Checkpoint 6 Passed

Hint

Add the property within the count path after the type property.

**6.**

Run npm test and see the error message

ReferenceError: validator is not defined

Checkpoint 7 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**7.**

Above the DinosaurSchema declaration in **dinosaur.js**, define a function named validator with function(args) {} notation.

Checkpoint 8 Passed

Hint

Define a const variable called validator and set it equal to an empty function.

**8.**

Run npm test and see the AssertionError message reappear. The validator function needs to return false to raise a validation error.

Checkpoint 9 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**9.**

Make validator return false.

Checkpoint 10 Passed

Hint

Add return false to the body of the function.

**10.**

Run npm test and see green!

Checkpoint 11 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

const Dinosaur = require('../../models/dinosaur');

const {assert} = require('chai');

const {connectAndDrop, disconnect} = require('../../database');

describe('Dinosaur', () => {

  beforeEach(connectAndDrop);

  afterEach(disconnect);

  describe('#name', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex'

      });

      assert.strictEqual(dino.name, 'T-rex');

    });

    it('is required', () => {

      const dino = new Dinosaur({});

      const error = dino.validateSync();

      assert.equal(error.errors.name.message, 'Path `name` is required.');

      assert.equal(error.errors.name.kind, 'required');

    });

  });

  describe('#count', () => {

    it('is invalid with 11', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        count: 11,

        risk: 'High'

      })

      dino.validateSync();

      assert.ok(dino.errors, 'model should have validation error');

    });

  });

  describe('#risk', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        risk: 'High'

      });

      assert.strictEqual(dino.risk, 'High');

    });

  });

});

**Path Validation II**

Your test passes and the model technically satisfies the DDZC’s no-more-than-10 regulation. But as it currently exists, there are two issues:

* the validation error message is not helpful, and
* the validation function will invalidate a count of any value.

Before you write the code to fix the issues, you’ll need more failing tests.

Remember that validation error messages are defined in the schema like this:

age: {

type: Number,

validate: [validator, 'Age must be above 9.']

}

And you can assert the value of multiple properties of [instance].errors.[path] like message, path, kind, and name. You can write out multiple assertions or use [assert.include](http://chaijs.com/api/assert/#method_include):

const errorInfo = person.errors.age;

assert.include(errorInfo, {

message: 'Age must be above 9.',

path: 'age',

kind: 'user defined',

name: 'ValidatorError'

});

The complete list of validators are available here: <http://mongoosejs.com/docs/schematypes.html>.

**Instructions**

**1.**

In **dinosaur-test.js** at the end of the 'is invalid with 11' test, assert that dino.errors.count.message strictly equals 'Cannot hold more than 10 dinosaurs.'.

Checkpoint 2 Passed

**2.**

Run the test suite and see the error message:

AssertionError: expected 'Validator failed for path `count` with value `11`' to equal 'Cannot hold more than 10 dinosaurs.'

Checkpoint 3 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**3.**

Add the validation error message 'Cannot hold more than 10 dinosaurs.' in **dinosaur.js**. Don’t forget to use square [ ] brackets!

Checkpoint 4 Passed

**4.**

Run the test suite and see green. Time to solve the second issue.

Checkpoint 5 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**5.**

In **dinosaur-test.js** below the is invalid with 11 test, add the following test.

it('is valid with 10', () => {

const dino = new Dinosaur({

name: 'Triceratops',

count: 10,

risk: 'Low'

});

dino.validateSync();

assert.isUndefined(dino.errors, 'model should be valid');

});

Checkpoint 6 Passed

**6.**

Run the test suite and see red. This will drive the implementation of the validator function, which is currently invalidating every value.

Checkpoint 7 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**7.**

Change the validator function to look like this

const validator = function(value) {

return value <= 10;

};

Checkpoint 8 Passed

Hint

In **dinosaur.js**, make sure that validator accepts one argument called value.

**8.**

Run the test suite and see green!

Checkpoint 9 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**9.**

Time to refactor: The validator function can be replaced by a built-in validator provided by Mongoose, max. Replace the validate property with

max: [10, 'Cannot hold more than 10 dinosaurs.']

Checkpoint 10 Passed

Hint

Change the validate property in **dinosaur.js**:

* Replace validate with max
* Replace validator with10

**10.**

Run the test suite and see green!

Checkpoint 11 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

const Dinosaur = require('../../models/dinosaur');

const {assert} = require('chai');

const {connectAndDrop, disconnect} = require('../../database');

describe('Dinosaur', () => {

  beforeEach(connectAndDrop);

  afterEach(disconnect);

  describe('#name', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex'

      });

      assert.strictEqual(dino.name, 'T-rex');

    });

    it('is required', () => {

      const dino = new Dinosaur({ });

      const error = dino.validateSync();

      assert.equal(error.errors.name.message, 'Path `name` is required.');

      assert.equal(error.errors.name.kind, 'required');

    });

  });

  describe('#count', () => {

    it('is invalid with 11', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        count: 11,

        risk: 'High'

      });

      dino.validateSync();

      assert.ok(dino.errors, 'model should be invalid');

      // Add assertion here

      assert.strictEqual(

        dino.errors.count.message,

        'Cannot hold more than 10 dinosaurs.')

    });

    // Add next test here

    it('is valid with 10', () => {

      const dino = new Dinosaur({

        name: 'Triceratops',

        count: 10,

        risk: 'Low'

      });

      dino.validateSync();

      assert.isUndefined(dino.errors, 'model should be valid');

    });

  });

  describe('#risk', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        risk: 'High'

      });

      assert.strictEqual(dino.risk, 'High');

    });

  });

});

**Methods I**

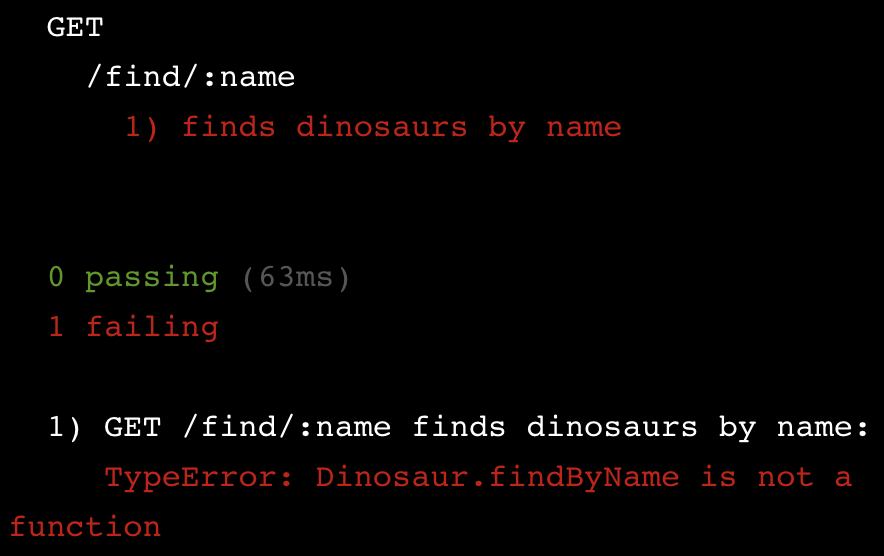
Now that you’ve tested the persistence of data, you’ll need methods to access that data.

Mongoose schemas support

* *static methods*: methods called by a model. They typically operate on a collection of documents (instances of the model).
* *instance methods*: methods called by an instance of a model. They typically operate on the document (model instance) itself.

From the previous exercise, you might recognize Dinosaur.findOne() as a static method and dino.save() as an instance method.

Sometimes you need to define additional methods for your application, like if you see a server-level error such as this:



This server test is failing because there is no model method to find Dinosaurs by name. You’ll need to drop to the model layer and write more tests.

The desired query is performed on a collection of documents, so it requires a static method, which is defined in [schema].statics and called according to the example below.

// static method - implementation

DinosaurSchema.statics.findByName = function(name, callback) {

return this.findOne({ name: name }, callback);

};

// static method - call the method

await Dinosaur.findByName('Velociraptor')

Use function() notation instead of arrow => notation to properly bind this.

You can test-drive the development of this method just like any other JavaScript method: Call the method and make assertions on its output.

**Instructions**

**1.**

Test-drive the static method findByName(name), which returns the first document that matches a given name.

In **dinosaur-test.js** under it('returns the first match on name' (scroll to the bottom of the file) add the following setup:

const fields = {

name: 'Pterodactyl',

count: 5,

risk: 'Low'

};

const dino = new Dinosaur(fields);

await dino.save();

Checkpoint 2 Passed

Hint

Scroll to the bottom of the file to find the it block.

Paste the provided code into that block.

**2.**

Call await Dinosaur.findByName('Pterodactyl') and store the result in a variable named stored.

Checkpoint 3 Passed

Hint

Make sure to:

* create a const called stored
* use await
* spell findByName correctly

**3.**

Using [assert.include](http://chaijs.com/api/assert/#method_include), assert that stored contains the same key-values pairs as fields.

Checkpoint 4 Passed

**4.**

Run npm test and see the error message

TypeError: Dinosaur.findByName is not a function

Checkpoint 5 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**5.**

In **dinosaur.js**, define an empty findByName method in DinosaurSchema.statics.

Use function() {} notation.

Checkpoint 6 Passed

**6.**

Run npm test and see the error message

AssertionError: Target cannot be null or undefined.

Checkpoint 7 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**7.**

In **dinosaur.js**, add the implementation code provided in the narrative above.

Make sure to add name and callback parameters to the function.

Checkpoint 8 Passed

Hint

The code for .findByName is provided in the narrative section above.

**8.**

Run npm test and see green!

Checkpoint 9 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

const Dinosaur = require('../../models/dinosaur');

const {assert} = require('chai');

const {connectAndDrop, disconnect} = require('../../database');

describe('Dinosaur', () => {

  beforeEach(connectAndDrop);

  afterEach(disconnect);

  describe('#name', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex'

      });

      assert.strictEqual(dino.name, 'T-rex');

    });

    it('is required', () => {

      const dino = new Dinosaur({ });

      const error = dino.validateSync();

      assert.equal(error.errors.name.message, 'Path `name` is required.');

      assert.equal(error.errors.name.kind, 'required');

    });

  });

  describe('#count', () => {

    it('is below 11 if risk is high', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        count: 11,

        risk: 'High'

      });

      dino.validateSync();

      assert.ok(dino.errors, 'model should be invalid');

      assert.equal(dino.errors.count.message, 'Cannot hold more than 10 dinosaurs.');

    });

    it('is valid with 10', () => {

      const dino = new Dinosaur({

        name: 'Triceratops',

        count: 10,

        risk: 'Low'

      });

      dino.validateSync();

      assert.isUndefined(dino.errors, 'model should be valid');

    });

  });

  describe('#risk', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        risk: 'High'

      });

      assert.strictEqual(dino.risk, 'High');

    });

  });

  describe('#save', () => {

    it('persists a dino', async () => {

      const fields = {

        name: 'Velociraptor',

        count: 3,

        risk: 'High'

      };

      const dino = new Dinosaur(fields);

      await dino.save();

      const stored = await Dinosaur.findOne({ 'name': 'Velociraptor' });

      assert.include(stored, fields);

    });

  });

  describe('.findByName', () => {

    it('returns the first match on name', async () => {

      const fields = {

        name: 'Pterodactyl',

        count: 5,

        risk: 'Low'

      };

      const dino = new Dinosaur(fields);

      await dino.save();

      const stored = await Dinosaur.findByName('Pterodactyl');

      assert.include(stored, fields);

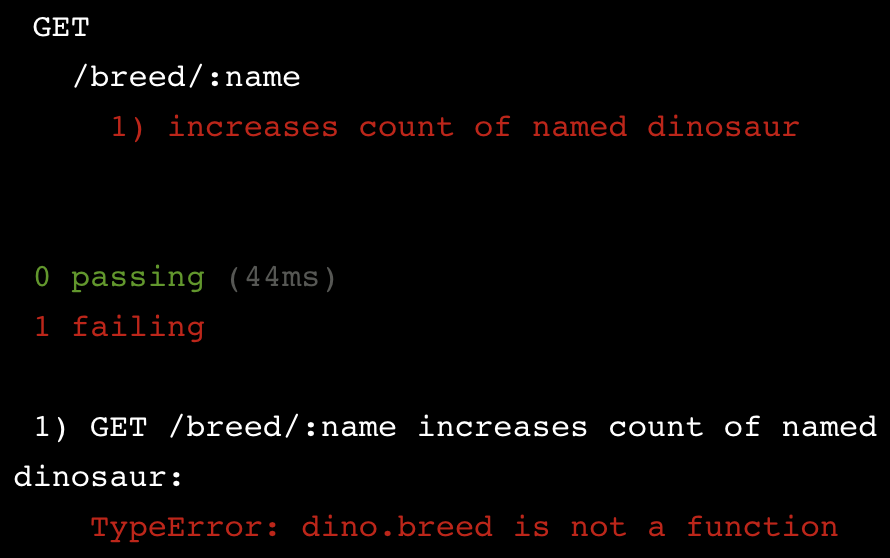
    });

  });

});

# Methods II

Sometimes you need an instance method for your application, like if you see a server-level error such as this:



This server test expects an increase to the dinosaur count, which is a responsibility of the Dinosaur model. You’ll need to drop to the model layer and test for a .breed() method.

.breed() will increase the count of one dinosaur. This kind of method is specific to an instance of a model, so you’ll need to define it as an instance method. Do this by storing it in [schema].methods as shown below.

// instance method - implementation

DinosaurSchema.methods.breed = function() {

this.count = this.count + 1;

};

// instance method - call the method

dino.breed()

Use function() notation instead of arrow => notation to properly bind this.

You can test-drive the development of this method just like any other JavaScript method: Call the method and make assertions on its output.

**Instructions**

**1.**

In **dinosaur-test.js** under it('increases count by 1' (scroll to the bottom of the file) add the following setup:

const start = 3;

const end = 4;

const dino = new Dinosaur({

name: 'Stegosaurus',

count: start,

risk: 'Low'

});

Checkpoint 2 Passed

Hint

Scroll to the bottom of the file to find the it block.

Paste the provided code into that block.

**2.**

Call dino.breed().

Checkpoint 3 Passed

Hint

Call dino.breed in the same it block.

**3.**

Assert that dino.count is strictly equal to end.

Checkpoint 4 Passed

Hint

Use assert.strictEqual. The first argument is dino.count and the second argument is end.

For this exercise, no variable name other than end is allowed.

**4.**

Run npm test and see the error message

TypeError: dino.breed is not a function

Checkpoint 5 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**5.**

In **dinosaur.js**, define an empty breed method in DinosaurSchema.methods.

Use function() {} notation.

Checkpoint 6 Passed

**6.**

Run npm test and see the error message

AssertionError: expected 3 to equal 4

Checkpoint 7 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

**7.**

In **dinosaur.js**, add the implementation code provided in the narrative above.

Checkpoint 8 Passed

Hint

In between the braces { } add

this.count = this.count + 1;

**8.**

Run npm test and see green!

Checkpoint 9 Passed

Hint

Run npm test in the terminal. Wait for output before checking work.

const Dinosaur = require('../../models/dinosaur');

const {assert} = require('chai');

const {connectAndDrop, disconnect} = require('../../database');

describe('Dinosaur', () => {

  beforeEach(connectAndDrop);

  afterEach(disconnect);

  describe('#name', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex'

      });

      assert.strictEqual(dino.name, 'T-rex');

    });

    it('is required', () => {

      const dino = new Dinosaur({});

      const error = dino.validateSync();

      assert.equal(error.errors.name.message, 'Path `name` is required.');

      assert.equal(error.errors.name.kind, 'required');

    });

  });

  describe('#count', () => {

    it('is below 11 if risk is high', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        count: 11,

        risk: 'High'

      });

      dino.validateSync();

      assert.ok(dino.errors, 'model should be invalid');

      assert.equal(dino.errors.count.message, 'Cannot hold more than 10 dinosaurs.');

    });

    it('is valid with 10', () => {

      const dino = new Dinosaur({

        name: 'Triceratops',

        count: 10,

        risk: 'Low'

      });

      dino.validateSync();

      assert.isUndefined(dino.errors, 'model should be valid');

    });

  });

  describe('#risk', () => {

    it('is a String', () => {

      const dino = new Dinosaur({

        name: 'T-rex',

        risk: 'High'

      });

      assert.strictEqual(dino.risk, 'High');

    });

  });

  describe('#save', () => {

    it('persists a dino', async () => {

      const fields = {

        name: 'Velociraptor',

        count: 3,

        risk: 'High'

      };

      const dino = new Dinosaur(fields);

      await dino.save();

      const stored = await Dinosaur.findOne({ 'name': 'Velociraptor' });

      assert.include(stored, fields);

    });

  });

  describe('.findByName', () => {

    it('returns the first match on name', async () => {

      const fields = {

        name: 'Pterodactyl',

        count: 9,

        risk: 'Low'

      };

      const dino = new Dinosaur(fields);

      await dino.save();

      const stored = await Dinosaur.findByName('Pterodactyl');

      assert.include(stored, fields);

    });

  });

  describe('#breed', () => {

    it('increases count by 1',  () => {

      const start = 3;

      const end = 4;

      const dino = new Dinosaur({

        name: 'Stegosaurus',

        count: start,

        risk: 'Low'

      });

      dino.breed();

      assert.strictEqual(dino.count, end)

    });

  });

});

# Review

You have developed a model using TDD!

* The model layer represents entities and interactions in a web app’s problem domain.
* Model paths can be test-driven using validators. Call validateSync and make assertions on the properties of [instance].errors.[path].
* The storage of data can be tested with [construction and updating methods](http://mongoosejs.com/docs/models.html) like save and update. Retrieval can be tested with [query methods](http://mongoosejs.com/docs/queries.html) like find, findOne, and findby.
* Static methods are stored in [schema].statics and instance methods are stored in [schema].methods. Both can be tested like any other JavaScript function.

To learn more about Mongoose, read the [guide](http://mongoosejs.com/docs/guide.html) and refer to the [API docs](http://mongoosejs.com/docs/api.html) as needed.