

Ruby/Rails Developer Challenge

---------------------------------

Directions: Build a minimal API, using Rails and MongoDB,

for our products (please see attached).

The API should be RESTful, serve JSON as the output, and contain the following CRUD functions:

CREATE

1. Creates a product

READ

1. Shows ALL products

2. Shows ONE product that best matches a given length/width/height/weight query (For example,

if I make an API request for a product with the following dimensions: 48”l X 14”w X 12”h (@ 42lbs)

the API should send me back “Golf - Small”.)

UPDATE

1. Updates a product

DESTROY

1. Deletes a product

---------------------------------

Other Requirements:

- Use proper HTTP error codes

- Validate data

- Create a script to populate the DB with the products.json file

- Provide Minimal API Documentation (a sample request with its parameters will suffice)

---------------------------------

Helpful Hints:

-With shipping packages, you can always go bigger, but you can’t go smaller

(i.e. if an item is 5”x5”x5", you will need the 6”x5”x6" package, not the 4”x5”x5” package).

This is also the case for weight.

In terms of interpreting the data,

think of dimensions and weight of a product as MAX dimensions and MAX weight.

---------------------------------

\*Full Stack Bonus (optional): Build a view that consumes the API with JavaScript,

and style it using Twitter Bootstrap

[**http://requiremind.com/riding-rails-4-along-with-mongoid-and-ruby-2-dot-0**](http://requiremind.com/riding-rails-4-along-with-mongoid-and-ruby-2-dot-0) **# OpenSSL errors and Rails  
#Unicorn**[**https://github.com/laserlemon/figaro**](https://github.com/laserlemon/figaro)[**http://railsapps.github.io/rails-composer/**](http://railsapps.github.io/rails-composer/)[**http://railsapps.github.io/rails-bootstrap/**](http://railsapps.github.io/rails-bootstrap/)[**https://github.com/mongoid/echo**](https://github.com/mongoid/echo)[**https://github.com/RailsApps/rails3-mongoid-devise**](https://github.com/RailsApps/rails3-mongoid-devise)[**http://railscasts.com/episodes/238-mongoid?view=asciicast**](http://railscasts.com/episodes/238-mongoid?view=asciicast)

[**https://spring.io/guides/gs/consuming-rest-jquery**/](https://spring.io/guides/gs/consuming-rest-jquery/)  
[**http://www.codeproject.com/Articles/664157/Consume-RESTful-Service-using-jQuery-in-Simple-S**](http://www.codeproject.com/Articles/664157/Consume-RESTful-Service-using-jQuery-in-Simple-S)[**https://spring.io/guides/gs/consuming-rest-jquery**/](https://spring.io/guides/gs/consuming-rest-jquery/)  
[**https://spring.io/guides/gs/consuming-rest-backbone**/](https://spring.io/guides/gs/consuming-rest-backbone/)  
[**http://blog.miguelgrinberg.com/post/writing-a-javascript-rest-client**](http://blog.miguelgrinberg.com/post/writing-a-javascript-rest-client)[**https://carminecarella.wordpress.com/2014/03/20/accounts-manager-a-sample-web-application-with-twitter-bootstrap-backbone-js-restful-j2ee-and-jpa**/](https://carminecarella.wordpress.com/2014/03/20/accounts-manager-a-sample-web-application-with-twitter-bootstrap-backbone-js-restful-j2ee-and-jpa/)  
[**https://www.twilio.com/blog/2013/12/votr-part-5-angularjs-crud-restful-apis.html**](https://www.twilio.com/blog/2013/12/votr-part-5-angularjs-crud-restful-apis.html)[**https://github.com/crtr0/votr-part5**](https://github.com/crtr0/votr-part5)[**https://spring.io/blog/2015/01/12/spring-and-angular-js-a-secure-single-page-application**](https://spring.io/blog/2015/01/12/spring-and-angular-js-a-secure-single-page-application)[**http://www.sitepoint.com/creating-crud-app-minutes-angulars-resource**/](http://www.sitepoint.com/creating-crud-app-minutes-angulars-resource/)  
[**https://github.com/jsprodotcom/source/blob/master/movieApp.zip**](https://github.com/jsprodotcom/source/blob/master/movieApp.zip)Single Page Application

**#rails generate rspec:install  
#rails g scaffold product name:string length:float width:float t** *#bundle exec rspec  
#bundle binstubs rspec-core  
# Prepare the database and add the default user to the database by running the commands:  
# $ rake db:seed  
#  
# $ rake db:test:prepare  
# Use rake db:reseed* ***if*** *you want to empty and reseed the database. Or you can use rake db:drop and rake db:setup .  
# The equivalent task for Rails with ActiveRecord is rake db:reset which will be available in Mongoid 4.0.  
# rake spec  
# rake cucumber*

[**http://docs.seattlerb.org/minitest/**](http://docs.seattlerb.org/minitest/)

[**https://github.com/rspec/rspec-rails**](https://github.com/rspec/rspec-rails)

[**https://github.com/rspec/rspec-mocks**](https://github.com/rspec/rspec-mocks)

**gem install rspec # for rspec-core, rspec-expectations, rspec-mocks**

**gem install rspec-mocks # for rspec-mocks only**

[**http://martinfowler.com/articles/mocksArentStubs.html**](http://martinfowler.com/articles/mocksArentStubs.html)

**Mocks Aren't Stubs**

**Test Double** as the generic term for any kind of pretend object used in place of a real object for testing purposes. The name comes from the notion of a Stunt Double in movies. (One of his aims was to avoid using any name that was already widely used.) Meszaros then defined four particular kinds of double:

**Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.

**Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an [in memory database](http://www.martinfowler.com/bliki/InMemoryTestDatabase.html) is a good example).

**Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

**Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

public interface MailService {

public void send (Message msg);

}

public class MailServiceStub implements MailService {

private List<Message> messages = new ArrayList<Message>();

public void send (Message msg) {

messages.add(msg);

}

public int numberSent() {

return messages.size();

}

}

We can then use state verification on the stub like this.

class OrderStateTester...

public void testOrderSendsMailIfUnfilled() {

Order order = new Order(TALISKER, 51);

MailServiceStub mailer = new MailServiceStub();

order.setMailer(mailer);

order.fill(warehouse);

assertEquals(1, mailer.numberSent());

}

Of course this is a very simple test - only that a message has been sent. We've not tested it was sent to the right person, or with the right contents, but it will do to illustrate the point.

Using mocks this test would look quite different.

class OrderInteractionTester...

public void testOrderSendsMailIfUnfilled() {

Order order = new Order(TALISKER, 51);

Mock warehouse = mock(Warehouse.class);

Mock mailer = mock(MailService.class);

order.setMailer((MailService) mailer.proxy());

mailer.expects(once()).method("send");

warehouse.expects(once()).method("hasInventory")

.withAnyArguments()

.will(returnValue(false));

order.fill((Warehouse) warehouse.proxy());

}

}

|  |  |
| --- | --- |
| up vote185down voteaccepted | I believe the biggest distinction is that a stub you have already written with predetermined behavior. So you would have a class that implements the dependency (abstract class or interface most likely) you are faking for testing purposes and the methods would just be stubbed out with set responses. They wouldn't do anything fancy and you would have already written the stubbed code for it outside of your test.  A mock is something that as part of your test you have to setup with your expectations. A mock is not setup in a predetermined way so you have code that does it in your test. Mocks in a way are determined at runtime since the code that sets the expectations has to run before they do anything.  Tests written with mocks usually follow an initialize -> set expectations -> exercise -> verify pattern to testing. While the pre-written stub would follow an initialize -> exercise -> verify. The purpose of both is to eliminate testing all the dependencies of a class or function so your tests are more focused and simpler in what they are trying to prove. |

Ruby/Rails Developer Challenge

---------------------------------

Directions: Build a minimal API, using Rails and MongoDB, for our products (please see attached). The API should be RESTful, serve JSON as the output, and contain the following CRUD functions:

CREATE

1. Creates a product

READ

1. Shows ALL products

2. Shows ONE product that best matches a given length/width/height/weight query (For example, if I make an API request for a product with the following dimensions: 48â€l X 14â€w X 12â€h (@ 42lbs) the API should send me back â€œGolf - Smallâ€.)

UPDATE

1. Updates a product

DESTROY

1. Deletes a product

---------------------------------

Other Requirements:

- Use proper HTTP error codes

- Validate data

- Create a script to populate the DB with the products.json file

- Provide Minimal API Documentation (a sample request with its parameters will suffice)

---------------------------------

Helpful Hints:

-With shipping packages, you can always go bigger, but you canâ€™t go smaller (i.e. if an item is 5â€x5â€x5", you will need the 6â€x5â€x6" package, not the 4â€x5â€x5â€ package). This is also the case for weight. In terms of interpreting the data, think of dimensions and weight of a product as MAX dimensions and MAX weight.

---------------------------------

\*Full Stack Bonus (optional): Build a view that consumes the API with JavaScript, and style it using Twitter Bootstrap

The meta name="viewport" comes from Bootstrap. It makes the page scale according to the browser dimensions, be it a browser on a desktop PC or one in a smartphone. What follows is the Bootstrap CSS file, which we import from a CDN (content delivery network) so that we don't have to host it ourselves. Next come the Javascript files for jQuery, Bootstrap and Knockout, also imported from CDNs.

In the body of the page we first have a top bar with the application title, created using Bootstrap's CSS styles. Then we have the main content area, where we will insert the application data. And finally, we will have our application code at the bottom, keeping in mind that for larger projects the application code should likely go into one or more independent Javascript source files.

Features of the Home Page

Salient features include:

Some CSS imported in the <head>, one placeholder for a file that doesn’t yet exist, but is named suggestively (“angular-bootstrap.css”) and one inline stylesheet defining the “ng-cloak” class.

The “ng-cloak” class is applied to the content <div> so that dynamic content is hidden until Angular JS has had a chance to process it (this prevents “flickering” during the initial page load).

The <body> is marked as ng-app="myapplication" which means we need to define a JavaScript module that Angular will recognise as an application called “myapplication”.

All the CSS classes (apart from “ng-cloak”) are from Twitter Bootstrap. They will make things look pretty once we get the right stylesheets set up.

The content in the greeting is marked up using handlebars, e.g. {{greeting.content}} and this will be filled in later by Angular (using a “controller” called “home” according to the ng-controller directive on the surrounding <div>).

Angular JS (and Twitter Bootstrap) are included at the bottom of the <body> so that the browser can process all the HTML before it gets processed.

We also include a separate “hello.js” which is where we are going to define the application behaviour.

We are going to create the script and stylesheet assets in a minute, but for now we can ignore the fact that they don’t exist.

This controller module is represented as a simple JavaScript function that is given AngularJS’s $scope and $http components. It uses the $http component to consume the REST service at "/greeting".

If successful, it will assign the JSON returned back from the service to $scope.greeting, effectively setting a model object named "greeting". By setting that model object, AngularJS can bind it to the application page’s DOM, rendering it for the user to see.