



APIs & You – Pt. 2

HUNTER SKRASEK - @HSKRASEK

Endpoint Planning Continued

Controllers

- ▶ We need to list Users and Sups? Easy, UsersController and Sups Controller
- ▶ Everything in REST is a resource, so each resource needs a controller... right?
- ▶ Not necessarily, some things are only sub-resources of Users, and can simply be a method on the UsersController
- ▶ These rules are flexible though

Example

Action	Endpoint	Route
CREATE	POST /users	Route::post('/users', 'UserController@create');
READ	GET /users/X	Route::get('/users/{id}', 'UserController@show');
UPDATE	PUT /users/X	Route::put('/users/{id}', 'UserController@update');
DELETE	DELETE /users/X	Route::delete('/users/{id}', 'UserController@destroy');
LIST	GET /users	Route::get('/users', 'UserController@index');
SUPS	GET /users/X/sups	Route::get('/users/{id}', 'SupsController@index');
DEVICES	GET /users/X/devices	Route::get('/users/{id}', 'UserController@devices');
FRIENDS	GET /users/X/friends	Route::get('/users/{id}', 'UserController@friends');

Input/Output Theory

AKA: REQUESTS AND RESPONSES

Requests

- ▶ Input to an API, is simply an HTTP request, such as:
GET /users?lat=40.5465&lon=-46.16567 HTTP/1.1
Host: api.getsup.com
- ▶ Posting is just as simple:
POST /users HTTP/1.1
Host: api.getsup.com
Authorization: Bearer aAbAsds123X-0
Content-Type: application/json

{ "first_name": "Hunter" }
- ▶ Define your Headers, define the body with appropriate formatting, send and done.

Responses

- ▶ Much like HTTP Request, the response will wind up as plain text.

- ▶ Unless its SSL, but lets not get to carried away just yet

HTTP/1.1 200 OK

Server: nginx

Content-Type: application/json

```
{ "id": 1, "first_name": "Hunter" }
```

- ▶ That is basically all an API is.

Supporting Formats

- ▶ Avoid application/x-www-urlformencoded
- ▶ `$_GET` != GET and `$_POST` != POST
- ▶ Why no x-www-urlformencoded?
 - ▶ Everything must be strings: `foo=something&bar=1&baz=0`
 - ▶ Data-types are important, so why just throw them away for “ease of access”?
- ▶ `Input::json('foo')` is available in Laravel, or you could use `file_get_contents('php://input')` if such a function is not available

Supporting Formats

```
{  
  "sup": {  
    "user_id": 2,  
    "type": "Scary Movie"  
  }  
}
```

- ▶ Perfectly valid HTTP Request
- ▶ Makes documentation easy, as well as error generation
- ▶ “Why in the world did you send a ‘sup’ to the ‘users’ endpoint??”

Supporting Formats

- ▶ Here is the same request in form data:

`checkin[user_id]=2&checkin[type]=Scary+movie`

- ▶ This could look even worse, but I didn't want to type it all out

- ▶ Do NOT mix JSON with form data:

```
json="{  
  \"checking\": {  
    \"user_id\": 2,  
    \"type\": \"Scary Movie\"  
  }  
}"
```

- ▶ Why? Just why would you make anyone do this? ☹

Supporting Formats

JSON

```
{  
  "sup": {  
    "id": 1,  
    "type": "Scary Movie",  
    "received": true,  
    "image": null,  
    "comment": ""  
  }  
}
```

XML

```
<sups>  
  <sup>  
    <id>1</id>  
    <type>Scary Movie</type>  
    <received>0</received>  
    <image />  
    <comment />  
  </sup>  
</sups>
```




Content Structure

LETS LOOK AT WHAT SOME BIG PLAYERS DO

Twitter

Item

```
{  
  "name": "Hunter Skrasek",  
  "id": "654894864135"  
}
```

Collection

```
[  
  {  
    "name": "Hunter Skrasek",  
    "id": "654894864135"  
  },  
  {  
    "name": "Terry Matula",  
    "id": "6547315795"  
  }  
]
```


Twitter

Pros

- ▶ Minimalistic
- ▶ Pretty much everything can understand it

Cons

- ▶ No way to add pagination or other meta-data

Facebook

Item

```
{  
  "name": "Hunter Skrasek",  
  "id": "654894864135"  
}
```

Collection

```
{  
  "data": [  
    {  
      "name": "Hunter Skrasek",  
      "id": "654894864135"  
    },  
    {  
      "name": "Terry Matula",  
      "id": "6547315795"  
    }  
  ]  
}
```


Facebook

Pros

- ▶ Simplistic response
- ▶ Namespace in collections for meta-data

Cons

- ▶ No meta-data for single items, without embedding within resource

My Preference

Item

```
{
  "data": {
    "name": "Hunter Skrasek",
    "id": "654894864135"
  }
}
```

Collection

```
{
  "data": [
    {
      "name": "Hunter Skrasek",
      "id": "654894864135"
    },
    {
      "name": "Terry Matula",
      "id": "6547315795"
    }
  ]
}
```




Status Codes, Errors, and Messages

OH MY!

HTTP Status Codes

- ▶ 2xx is all forms of success
 - ▶ The requested action was successful
- ▶ 3xx is all forms of redirection
 - ▶ Lets send the requester elsewhere for fun
- ▶ 4xx is all forms of client errors
 - ▶ The client has done something invalid
- ▶ 5xx is all forms of service errors
 - ▶ The service is borked. Oops

Some Example Codes to Use

- ▶ 201 – Created Ok
- ▶ 202 – Accepted but processing async
- ▶ 400 – Bad Request (technically bad syntax, but some use this for validation errors)
- ▶ 401 – Unauthorized
- ▶ 403 – Forbidden
- ▶ 404 – URL is invalid, or resource doesn't exist
- ▶ 410 – Data was deactivated, deleted, etc.
- ▶ 500 – The API screwed up
- ▶ 503 – API is unavailable, please leave a message after the beep

Error Codes and Messages

- ▶ Error codes and message help explain what happened
- ▶ Using HTTP status codes exclusively usually doesn't work
- ▶ Status codes are more like hints, error codes give more details

Error or Errors



Common Pitfalls

- ▶ 200 OK and Error Code
 - ▶ Do not do this. This is bad. Seriously, might have to smack you
 - ▶ Not joking
- ▶ 404 Abuse
 - ▶ 404 is used for, “never existed”, “you cant view it”, “no longer exists”, which is too vague
 - ▶ 404, along with 403 and 410 can help, but still can be too vague
 - ▶ Complement these with error codes explain the situation



Questions?

Concerns, Complains, Praise?