



Search for articles...

All Collections > API > API v2 walkthrough

API v2 walkthrough

Read through an example of a series of API calls that illustrate the use of API v2 capabilities.

Updated over 9 months ago

Table of contents



Introduction

The purpose of this tutorial is to provide a walkthrough of the functionality found in NationBuilder's V2 API, which follows the JSON:API specification. Say we have created a signup named Grace using this payload to `POST /api/v2/signups`

```
{
  "data": {
    "type": "signups",
    "attributes": {
      "first_name": "Grace",
      "last_name": "Hopper",
      "email": "grace.hopper@example.com",
      "support_level": 1,
      "home_address_attributes": {
        "address1": "123 Sesame St.",
        "city": "New York",
        "state": "NY"
      },
      "custom_values": {
        "programming_language": "COBOL",
```

```

        "bugs_found": 1
    }
}
}

```

This documentation will be walking through new concepts in API V2, using data associated with Grace as the center of walkthrough.

Sparse fields

The signup resource contains many attributes, some of which may not be necessary for all API client usecases. In service of supporting slimmer response bodies, sparse fields can be specified using the query parameter `field[resource_type]=comma,delimited,list`. Say we only wanted signup names, email, and support level for our API functionality, the call would look as follows:

```
/api/v2/signups?fields[signups]=first_name,last_name,custom_values
```

Which would return the following payload—abridged to 2 entries for brevity:

```

{
  "data": [
    {
      "id": "18",
      "type": "signups",
      "attributes": {
        "first_name": "Grace",
        "last_name": "Hopper",
        "custom_values": {
          "programming_language": "COBOL",
          "bugs_found": 1.0
        }
      }
    },
    {
      "id": "19",
      "type": "signups",
      "attributes": {
        "first_name": "Joan",
        "last_name": "Cooney",

```

```

        "custom_values": {
            "programming_language": null,
            "bugs_found": null
        }
    }
}
]
}

```

Extra Fields

If we make a request for Grace via `GET /api/v2/signups/18` you'll notice that `home_address` is not in the response body. This is because `home_address` is an extra field. By default, extra fields are not included in response bodies, they must be explicitly requested by the caller. Simialr to sparse fields, extra fields are requested via a query parameter following the form `extra_fields[resource_type]=comma,delimited,list`. Extra fields defined for a resource are documented under the resource's query string parameters in NationBuilder's OpenAPI reference documenation. See [here](#) for the signup resource query parameters documenation. Building upon our previous request using sparse fields, a request including extra fields is demonstrated below:

```

/api/v2/signups/18?
fields[signups]=first_name,last_name,custom_values&extra_fields[signups]=home_address

```

```

{
  "id": "18",
  "type": "signups",
  "attributes": {
    "first_name": "Grace",
    "last_name": "Hopper",
    "home_address": {
      "address1": "123 Sesame St.",
      "city": "New York",
      "state": "NY"
    },
    "custom_values": {
      "programming_language": "COBOL",
      "bugs_found": 1.0
    }
  }
}

```

```
}  
}
```

Filtering

Basic filtering on resources is supported in API V2. Given the two signups used thus far, we can filter on our signups' `support_level` field using a `filter` query parameter following the form `filter[attribute_name][clause]=comma,delimited,list`. The complete list of filter clauses can be viewed [here](#). Here, we will filter for `support_level` greater than 1.

```
/api/v2/signups?fields[signups]=first_name,last_name,support_level&filter[support_level]  
[gt]=1
```

```
{  
  "data": [  
    {  
      "id": "19",  
      "type": "signups",  
      "attributes": {  
        "first_name": "Joan",  
        "last_name": "Cooney",  
        "support_level": 2  
      }  
    }  
  ]  
}
```

As noted earlier, filtering also accepts as comma-delimited list of values in most cases.

Demonstrating this again with `support_level` using the `eq` clause:

```
/api/v2/signups?fields[signups]=first_name,last_name,support_level&filter[support_level]  
[eq]=1,2
```

```
{  
  "data": [  
    {  
      "id": "18",  
      "type": "signups",
```

```

      "attributes": {
        "first_name": "Grace",
        "last_name": "Hopper",
        "support_level": 1
      }
    },
    {
      "id": "19",
      "type": "signups",
      "attributes": {
        "first_name": "Joan",
        "last_name": "Cooney",
        "support_level": 2
      }
    }
  ]
}

```

Filtering for values with special characters

There are attributes which may contain special characters, such as a comma in a note, which we wish to query on. Given a signup with the note: "fundraiser, volunteer", we would filter for this value using `{{value}}` to escape our special characters. Our filter would look like this:

```
/api/v2/signups?filter[note][contains]={{fundraiser, volunteer}}
```

Which will give us all signups containing the exactly the value "fundraiser, volunteer". As with other filter values, escaped values may be used in comma delimited lists like so:

```
/api/v2/signups?filter[last_name][contains]={{, jr}},{{, esq.}}
```

Filtering on hash attributes

Some attributes in the API, such as the signup resource's `custom_values`, are a hash of values. The API supports equality filtering on attributes in this format using the `eq` clause. Given this request `GET /api/v2/signups?filter[custom_values][eq]={ "programming_language": "COBOL" }` we will get a response of all resources with where the custom value of slug "programming_language" is "COBOL". Note that the value to filter on must be properly formatted JSON.

Say we wanted to find signups with multiple `programming_language` values. This is done by passing comma delimited values like so:

```
GET /api/v2/signups?filter[custom_values][eq]={ "programming_language": "COBOL" },{
"programming_language": "Ruby" }
```

```
{
  "data": [
    {
      "id": "18",
      "type": "signups",
      "attributes": {
        "first_name": "Grace",
        "last_name": "Hopper",
        "custom_values": {
          "programming_language": "COBOL",
          "bugs_found": 1.0
        }
      }
    },
    {
      "id": "30",
      "type": "signups",
      "attributes": {
        "first_name": "Yukihiro",
        "last_name": "Matsumoto",
        "custom_values": {
          "programming_language": "Ruby",
          "bugs_found": null
        }
      }
    }
  ]
}
```

Managing relationships between resources

In NationBuilder's V2 API, resources can have relationships with other resources. These relationships can be read and mutated (e.g. creating, updating, or deleting associations

between resources). Fetching related resources is referred to as **sideloading**; mutation of these resources is referred to as **sideposting**.

In the following sections, we'll walk through sideloading relationships using Grace's signup. We will also examine the different sideposting methods used to apply changes to a resource's relationships.

Sideloading

After signup creation, Grace signed multiple petitions published on the nation's website, resulting in multiple petition signatures. We can request resources that have relationships with Grace's signup data, referred to as **sideloading data**. This is done via a query parameter in the format `include=relationship,another_relationship`. Requesting Grace's recruiter data and her petition signatures would look like this:

```
GET /api/v2/signups/18?include=recruiter,petition_signatures
```

```
{
  "data": {
    "id": "6",
    "type": "signups",
    "attributes": {
      "first_name": "Grace",
      "last_name": "Hopper",
      "email1": "grace.hopper@example.com"
    },
    "relationships": {
      "petition_signatures": {
        "links": {
          "related": "/api/v2/petition_signatures?filter[signup_id]=18"
        },
        "data": [
          {
            "type": "petition_signatures",
            "id": "2"
          }
        ]
      }
    }
  },
  "included": [
    {
```

```

    "id": "2",
    "type": "petition_signatures",
    "attributes": {
      "page_id": "11",
      "petition_page_id": "1",
      "recruiter_id": null,
      "signup_id": "18",
      "is_private": false,
      "comment": "I, Grace Hopper, support you!",
      "created_at": "2024-01-01T13:54:36-04:00",
      "updated_at": "2024-01-01T13:54:36-04:00"
    },
    "relationships": {
      "petition": {
        "links": {
          "related": "/api/v2/petitions/1"
        }
      },
      "signup": {
        "links": {
          "related": "/api/v2/signups/18"
        }
      }
    }
  },
  {
    "id": "555",
    "type": "signup",
    "attributes": { ... }
  },
  "meta": {}
}

```

When we sideload related resources, there are two things that change about our response. First, the data included in the relationships portion of our primary payload changes. In this case, our signup resource, Grace, will see have changes to the `petition_signatures` key of the `relationships` portion of the response. Where previously, our `petition_signatures` relationship would look like so:

```

"relationships": {
  "petition_signatures": {
    "links": {

```



```

    "related": "/api/v2/petition_signatures?filter[signup_id]=18"
  }
}

```

It will now contain data about the sideloaded petition signatures:

```

"relationships": {
  "petition_signatures": {
    "links": {
      "related": "/api/v2/petition_signatures?filter[signup_id]=18"
    },
    "data": [
      {
        "type": "petition_signatures",
        "id": "2"
      }
    ]
  }
}

```

The added data key within a relationships array is necessary for associating requested resources with the sideloaded resources that are found within the array of data with the "included" key.

Filtering and sparse fields on sideloaded data

Were Grace to have a very large number of petition signatures, some of which are less relevant to use as they're older. When sideloading her signatures, we can apply filters to the `petition_signatures` resource to reduce the number of signatures in our response.

Remembering our filter query param format of `filter[resource_name][clause]`, a possible filter to apply here would be to request all signatures created since the beginning of 2024 like so:

```

GET /api/v2/signups/18?include=petition_signatures&filter[petition_signatures][created_at]
[gte]=2024-01-01

```

This same concept would apply to a request for all signups. This time, we'll also apply sparse fields to our petition signatures as we only want the comment on the signature:

GET /api/v2/signups?

fields[signups]=first_name&include=petition_signatures&filter[petition_signatures]
[created_at][gte]=2024-01-01&fields[petition_signatures]=comment

Which would give us a response looking like this:

```
{
  "data": [
    {
      "id": "16",
      "type": "signups",
      "attributes": {
        "first_name": "Grace"
      },
      "relationships": {
        "petition_signatures": [
          "data": [
            {
              "type": "petition_signatures",
              "id": "2"
            }
          ]
        }
      }
    },
    {
      "id": "30",
      "type": "signups",
      "attributes": {
        "first_name": "Yukihiro"
      },
      "relationships": {
        "petition_signatures": [
          "data": [
            {
              "type": "petition_signatures",
              "id": "8"
            }
          ]
        }
      }
    }
  ],
}
```

```

    ],
    "included": [
      {
        "id": "2",
        "type": "petition_signatures",
        "attributes": {
          "comment": "I support you!"
        }
      },
      {
        "id": 8,
        "type": "petition_signatures",
        "attributes": {
          "comment": "We can do it."
        }
      }
    ]
  }

```

Note that our "included" array of responses contains petition signature resources that are associated with both signup resources in the `data` portion of our payload. As mentioned previously, a resource's relationship data is used for determining which resources are associated with each other.

Sideposting

Just as the API supports the ability to request related resources, it also allows for creating, updating, and deleting related resources with a single request. These capabilities are referred to as `create`, `destroy`, `dissociate`, and `update` methods, and are used within a POST, PATCH, or PUT request body. The below payload is for a PATCH request and makes use of all four methods of applying changes to a resource's relationships.

```

{
  "data": {
    "id": "6",
    "type": "signups",
    "attributes": {
      "support_level": 2,
      "home_address_attributes": {
        "delete": true
      }
    }
  }
}

```

```
    },
    "registered_address_attributes": {
      "address1": "123 Sesame St.",
      "city": "New York",
      "state": "NY"
    }
  },
  "relationships": {
    "author": {
      "data": {
        "type": "signups",
        "id": "42",
        "method": "update"
      }
    },
    "recruiter": {
      "data": {
        "type": "signups",
        "id": "555",
        "method": "disassociate"
      }
    },
    "petition_signatures": {
      "data": {
        "type": "petition_signatures",
        "id": "888",
        "method": "destroy"
      }
    },
    "signup_tags": {
      "data": [
        {
          "type": "signup_tags",
          "temp-id": "temp-signup-tag-id",
          "method": "create"
        },
        {
          "type": "signup_tags",
          "id": "123",
          "method": "update"
        }
      ]
    },
    "voter": {
      "data": {
```

```

        "type": "voters",
        "temp-id": "temp_voter_id",
        "method": "create"
      }
    }
  },
  "included": [
    {
      "type": "voters",
      "temp-id": "temp_voter_id",
      "attributes": {
        "is_absentee_voter": true,
        "is_active_voter": true,
        "is_early_voter": true,
        "is_permanent_absentee_voter": false
      }
    },
    {
      "type": "signup_tags",
      "temp-id": "signup-tag-1",
      "attributes": {
        "name": "updated_via_api"
      }
    },
    {
      "type": "signups",
      "id": "42",
      "attributes": {
        "support_level": 1
      }
    }
  ]
}

```

As you can see in the primary section of this payload, we are updating Grace's support level, removing her home address, and adding a registered address. This PATCH request also creates voter data for Grace, which is represented as a Voter resource. It creates a new signup tag, `updated_via_api`, that is then added to Grace and also tags Grace's signup with another, already existing signup tag. We're also updating Grace's author relationship and that resource's support level. Additionally, we're removing Grace's recruiter and a petition signature. In the following sections, we'll walk through each sidepost section of this payload.

The `create` method

In the above payload, we are creating two resources associated with Grace's signup. One voter resource and a new signup tag. Creating a resource requires the addition of two pieces of data in the payload. In the primary data payload, we add entries under the `relationships` key that look like this:

```
"relationships": {
  "voter": {
    "data": {
      "type": "voters",
      "temp-id": "temp_voter_id",
      "method": "create"
    }
  },
  "signup_tags": [
    {
      "type": "signup_tags"
      "temp-id": "temp-signup-tag-id",
      "method": 'create'
    },
    {
      # ... signup_tag update here
    }
  ]
}
```

Notice that our voter and signup tag entries differ based on the type of relationship the data has to our signup resource. Signups only have a single voter data resource, but may have many signup tags. This is differentiable by the single voter entry compared to they array of signup tag entries. Another unique aspect of resource creation via sideposting is the `temp-id` key that is used in both the relationship entries and the entries within the `included` array, which we will address here shortly. A resource's `temp-id` can be arbitrarily assigned by the client making the request, but these `temp-id` values must be unique across the request body. These are used to correctly map relationship entries to included entries.

The attributes for sideposted resources being created must be added to the `included` array. Below is the relevant entries for our newly created voter and signup tag resources:

```
# rest of payload...
"included": [
```

```

{
  "type": "voters",
  "temp-id": "temp_voter_id",
  "attributes": {
    "is_absentee_voter": true,
    "is_active_voter": true,
    "is_early_voter": true
    "is_permanent_absentee_voter": false
  }
},
{
  "type": "signup_tags",
  "temp-id": "signup-tag-1",
  "attributes": {
    "name": "updated_via_api"
  }
},
# ... remaining included resources here
]

```

The update method

The sideposting `update` method is used for updating both the relationship between resources, as well as updating attributes of related resources. We do the former in the above payload to update Grace's signup to have a signup tag for an already created tag with the ID of "123". For the latter, let's assume Grace already has an author relationship with ID "42". We are passing an update payload in the `included` array of data to update the recruiter signup with a `support_level` of 1.

Our relationships entries for these updates are here:

```

# ... rest of payload
"relationships": {
  "author": {
    "data": {
      "type": "signups",
      "id": "42",
      "method": "update"
    }
  },
  "signup_tags": {
    "data": [
      {

```

```

        "type": "signup_tags",
        "id": "123",
        "method": "update"
    }
]
},
# ... rest of payload

```

And our included resources array looks like this:

```

# ... "data" payload
"included": [
    # ... other sideposted resources
    {
        "type": "signup_tags"
        "temp-id": "signup-tag-1"
        "attributes": {
            "name": "updated_via_api"
        }
    },
    {
        "type": "signups"
        "id": "42"
        "attributes": {
            "support_level": 1
        }
    }
]

```

Just as in the `create sidepost` method, the `type`, `temp-id`, and `id` values are used to map the full resource data from the `included` array, to the correct entries in the `relationships` portion of the payload.

The `destroy` method is used to delete a related resource via sideposting. This means that you can remove a resource that is related to the primary resource in a single API request. In the earlier payload, we used the `destroy` method to delete one of Grace's petition signatures. Here's the relevant portion of the payload:

```

jsonCopy code"relationships": { "petition_signatures": { "data": { "type": "petition_signat

```



In this example, we are deleting the `petition_signature` with ID "888" that is associated with Grace's signup. This instructs the API to remove the petition signature resource entirely. It is important to note that when using the `destroy` method, the resource specified will be deleted, not just unlinked from the primary resource. If you wish to only remove the association without deleting the resource, you should use the `disassociate` method instead.

The `disassociate` method

The `disassociate` method is used to remove the relationship between two resources without deleting either resource. This is useful when you want to unassign or detach a related resource from the primary resource.

In our earlier payload, we used the `disassociate` method to remove Grace's recruiter. Here's the relevant part of the payload:

```
jsonCopy code"relationships": { "recruiter": { "data": { "type": "signups", "id": "555", "r
```



In this example, we are removing the association between Grace's signup and the recruiter with ID "555". The recruiter resource remains intact in the system, but it is no longer linked to Grace's signup.

A caution on `destroy` and `disassociate` methods

When managing relationships between resources via sideposting, it's important to utilize `destroy` and `disassociate` methods judiciously, as they are destructive actions that can impact data integrity if mis-applied. Ensure that API functionality leveraging these methods are tested within a sandbox prior to utilizing this functionality in live environments.

Pagination

API V2 uses pagination to limit the amount of data returned in a single response. By default, endpoints that return collections of resources will paginate the results with pages of size 20. You can control pagination using the `page[number]` and `page[size]` query parameters. A page size of up to 100 is allowed.

For example, to retrieve the second page of signups with 10 signups per page:

```
GET /api/v2/signups?page[number]=2&page[size]=10
```

The response will include pagination metadata in the `meta` section, as well as `links` for navigating to the `first`, `last`, `prev`, and `next` pages. Here's an example of the `meta` and `links` sections:

```
{
  "data": [ /* array of signup resources */ ],
  "meta": {
    "total_pages": 5,
    "total_count": 50,
    "current_page": 2
  },
  "links": {
    "self": "/api/v2/signups?page[number]=2&page[size]=10",
    "prev": "/api/v2/signups?page[number]=1&page[size]=10",
    "next": "/api/v2/signups?page[number]=3&page[size]=10"
  }
}
```

By using the `links` provided in the response, you can navigate through the pages of results. Note that the current implementation of pagination will always provide a next link. When a request returns an empty array of resources, the final page of resources has been reached.

Related Articles

- NationBuilder v2 API Core Concepts >
- NationBuilder API QuickStart Guide >
- API Authentication Guide >
- Relationships between Resources in v2 API >
- Using parameters to interact with the NationBuilder API >

Did this answer your question?



[Demos & webinars](#) [Courses](#) [Website themes](#) [API documentation](#)