STIR TREK 2024

Why Your API Needs a Spec and How to Get it One

Liz Johnson, Software Engineer

Hi! - I'm Liz

Senior Fullstack Dev

I am a full stack software engineer who has worked for companies of various sizes and at various stages of maturity. I am passionate about building scalable software because it enables people to get more done faster and that feels good! I love learning about helpful technologies and patterns that amazing people have created to make our lives as engineers better. And I also love sharing what I've learned with others.

Liz Johnson

Software Engineer

Get ready!
The Presentation Is

The Project

What We'll Cover

01 What is an API spec?

02 Why do you care?

03 How do you interact with them?

04 How do you implement them?

05 How can you migrate to them?

Why We're Talking About It

Enable clients and API engineers to build quickly, easily, and with confidence.

This supports a positive developer experience and helps businesses ship better features faster!

What is an API Spec

01

An API specification defines patterns for interaction with an API. Implementing an API to a specification means that anyone interacting with your API can expect to interact with it in a specific way.

{ json:api }

JSON:API

What is JSON:API?

A specification for building APIs in JSON.

It provides guidelines and conventions for structuring JSON responses from APIs. It aims to simplify the process of consuming APIs and reducing the need for custom code on the client side. It defines rules for resource objects, relationships between resources, and includes for fetching related resources in a single request.

Check it Out: The Docs

This is a sample JSON:API request.

```
www.example.com/api/v1/students?include=students_class
```


```
{
  data: [
    {
      id: "1",
      type: "students",
      attributes: {
        first_name: "Brok",
        last_name: "Lee",
        birthdate: "2014-01-15"
      },
      relationships: {
        students_class: {
          data: {
            type: "students_classes",
            id: "1"
          }
        }
      }
    }
  ],
  included: [
    {
      id: "1",
      type: "students_classes",
      attributes: {
        teacher_first: "Julie",
        teacher_last: "Lindsay",
        room: "1-A"
      },
      relationships: {
        students: {
          meta: {
            included: false
          }
        }
      }
    }
  ],
  meta: {}
}
```

GraphQL

What is GraphQL?

A query language for your API

It enables clients to request exactly the data they need from the server, eliminating over-fetching and under-fetching of data.

Check it Out: The Docs

This is a sample GraphQL request.

```
query {
  students(lastName: "Flower") {
    id
    firstName
    lastName
    studentsClass {
        room
    }
  }
}
```

```
{
    "data": {
        "students": [
            {
                "id": "2",
                "firstName": "Cauli",
                "lastName": "Flower",
                "birthdate": "2014-01-16",
                "studentsClass": {
                    "room": "1-A"
                }
            }
        ]
    }
}
```

“Ask for what you need, get exactly that”

Why do you care

02

Client needs are often diverse, especially as your application grows

We may initially build an endpoint for a single page, but that endpoint grows in complexity as we implement more features.

Student Information

Client’s various needs

- Students with detailed information and their teacher
- Students and their pickup information – carpool, bus, after school care etc.
- Students and their grade reports

Evolution of the students endpoint

Student directory information

```
{
  "students": [
    {
      "firstName": "julie",
      "lastName": "jones",
      "address": "123 5th street",
      "phoneNumber": 5551234567,
      "caregiverFirst": "katie",
      "caregiverLast": "jones",
      "teacher": {
        "firstName": "lindsey",
        "lastName": "johnson",
        "phone": 5556667777
      }
    }
  ]
}
```

Evolution of the students endpoint

Student pickup information

```
{
  "students": [
    {
      "firstName": "julie",
      "lastName": "jones",
      "address": "123 5th street",
      "phoneNumber": 5551234567,
      "caregiverFirst": "katie",
      "caregiverLast": "jones",
      "teacher": {
        "firstName": "lindsey",
        "lastName": "johnson",
        "phone": 5556667777
      },
      "pickup": {
        "zone": "1-A",
        "type": "carpool",
        "days": "all",
        "pickupTag": 386
      }
    }
  ]
}
```

Evolution of the students endpoint

Student grade information


```json
{
  "students": [
    {
      "firstName": "julie",
      "lastName": "jones",
      "address": "123 5th street",
      "phoneNumber": 5551234567,
      "caregiverFirst": "katie",
      "caregiverLast": "jones",
      "teacher": {
        "firstName": "lindsey",
        "lastName": "johnson",
        "phone": 5556667777
      },
      "pickup": {
        "zone": "1-A",
        "type": "carpool",
        "days": "all",
        "pickupTag": 386
      },
      "grades": [
        {
          "history": {
            "assignments": [
              {
                "overallPercentage": 89,
                "name": "paper1"
              },
              {
                "overallPercentage": 90,
                "name": "test1"
              }
            ]
          }
        },
        {
          "math": {
            "assignments": [
              {
                "overallPercentage": 89,
                "name": "hw1"
              },
              {
                "overallPercentage": 90,
                "name": "test1"
              }
            ]
          }
        }
      ]
    }
  ]
}
```

```ruby
def index
  @students = Student.include(:grades, :teacher, :pickup).all
  render @students
end
```

This endpoint now does more than any one client call needs. In all cases it's making JOIN statements that it doesn't need to and returning data it doesn't need to.

```ruby
def students_teacher
  @students = Student.include(:teacher).all
  render @students
end

def student_pickup
  @students = Student.include(:pickup).all
  render @students
end

def student_grades
  @students = Student.include(:grades).all
  render @students
end
```

We could also create endpoints for each of those different requests to avoid getting data that you don't need, but this pattern would likely result, eventually, in an unmanageable amount of endpoints

```ruby
class StudentsController
  def index
    @students = Student.all
    render @students
  end
end

class GradesController
  def index
    @grades = Grade.all
    render @grades
  end
end
```

Maybe we just don't have any JOINS. We can have controllers for each endpoint and we will ask the client to make all the calls it needs and then stitch together a response

```ruby
def index
  @students = Student.include(:teacher, :pickup).all
  render @students
end

def student_grades
  @students = Student.include(:grades).all
  render @students
end
```

Another possible way this goes is that the student's pickup information gets included with the index endpoint, but a different engineer decides grade information should be separate.

```ruby
def index
  @students = Student.include(:teacher, :pickup).all
  render @students
end

def student_grades
  @students = Student.include(:grades).all
  render @students
end
```

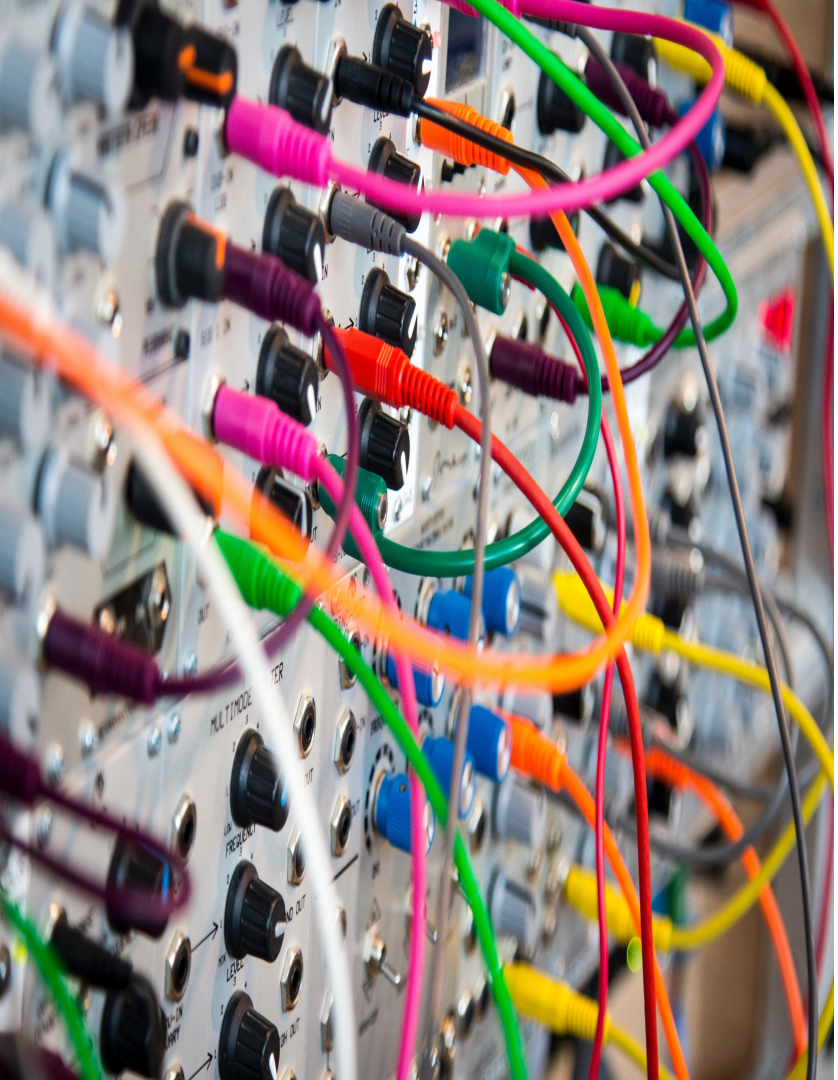
Now let's say a brand new engineer to the team is asked to support a fourth piece of functionality that lists a students and their elective classes for the current semester. Where should the engineer put this data?

```ruby
def index
  @students = Student.include(:teacher, :pickup).all
  render @students
end

def student_grades
  @students = Student.include(:grades).all
  render @students
end
```

Additionally, the client doesn't have a pattern to follow now. They don't know where they will be getting this information.

If you want your API to scale well, it’s helpful to establish patterns early on and stick to them as long as possible. These patterns should be ones that allow your API to feel manageable and performant long term.



How can GraphQL or JSON:API help?

- The client can ask for what they need and get exactly that
- The client has a standardized way to interact with the backed
- Developers have established patterns to follow which saves a ton of time and can help with complexity

How do you interact with them

03

We've already seen a couple examples but let's revisit those. Let's start with JSON:API. JSON:API is RESTful and you request specific data using query params. Here is an example where records are filtered by a field on the root resource.

```
http://localhost:3000/api/v1/students?filter[last_name]=Lee
```

```
{
  data: [
    {
      id: "1",
      type: "students",
      attributes: {
        first_name: "Brok",
        last_name: "Lee",
        birthdate: "2014-01-15"
      },
      relationships: {
        students_class: {
          meta: {
            included: false
          }
        }
      }
    }
  ]
}
```

In this example we retrieve related records along with the root resource.

```
http://localhost:3000/api/v1/students?include=students_class
```


```
{
  data: [
    {
      id: "1",
      type: "students",
      attributes: {
        first_name: "Brok",
        last_name: "Lee",
        birthdate: "2014-01-15"
      },
      relationships: {
        students_class: {
          data: {
            type: "students_classes",
            id: "1"
          }
        }
      }
    }
  ],
  included: [
    {
      id: "1",
      type: "students_classes",
      attributes: {
        teacher_first: "Julie",
        teacher_last: "Lindsay",
        room: "1-A"
      },
      relationships: {
        students: {
          meta: {
            included: false
          }
        }
      }
    }
  ],
  meta: {}
}
```

Creating a record and associating it to a related record can be handled with a POST request that looks like this.

```
http://localhost:3000/api/v1/students
```

```
{
 "data": {
  "type": "students",
  "attributes": {
    "first_name": "clayton",
    "last_name": "johnson"
  },
  "relationships": {
    "students_class": {
      "data": {
        "id": 1,
        "type": "students_class"
      }
    }
  }
 }
}
```

Let's look at some examples using GraphQL. GraphQL is not RESTful and uses only POST requests. Here's a request where we filter records by a field on our root resource.

```
query {
  students(lastName: "Flower") {
    id
    firstName
    lastName
    studentsClass {
        room
    }
  }
}
```

```
{
    "data": {
        "students": [
            {
                "id": "2",
                "firstName": "Cauli",
                "lastName": "Flower",
                "birthdate": "2014-01-16",
                "studentsClass": {
                    "room": "1-A"
                }
            }
        ]
    }
}
```

Here’s an example where we retrieve fields from a related record.

```
query {
  students {
    id
    firstName
    lastName
    studentsClass {
        room
    }
  }
}
```

```
{
    "data": {
        "students": [
            {
                "id": "1",
                "firstName": "Brok",
                "lastName": "Lee",
                "birthdate": "2014-01-15",
                "studentsClass": {
                    "room": "1-A"
                }
            },
            {
                "id": "2",
                "firstName": "Cauli",
                "lastName": "Flower",
                "birthdate": "2014-01-16",
                "studentsClass": {
                    "room": "1-A"
                }
            }
        ]
    }
}
```

Creating a record and associating it to a related record can be handled with a request that looks like this.

```
mutation {
  createStudent(firstName: "clayton",
                lastName: "johnson",
                birthdate: "05-17-1992",
                studentsClassId: 1 ) {
    errors
    student {
      firstName
      lastName
      studentsClass {
        teacherFirst
        teacherLast
        room
      }
    }
  }
}
```

```
{
    "data": {
        "createStudent": {
            "errors": [],
            "student": {
                "firstName": "clayton",
                "lastName": "johnson",
                "studentsClass": {
                    "teacherFirst": "Julie",
                    "teacherLast": "Lindsay",
                    "room": "1-A"
                }
            }
        }
    }
}
```

How do you implement them

04

Real Life Example: Modernization project.

We wanted to modernize an old PHP backend while adding new features.

The frontend clients were React clients, an Android app, and an IOS App.

- Backend teams and client teams worked separately
- We had to add a lot of functionality fast!
- Our new API was a Rails application and we made it JSON:API compliant using a library called Graphiti

Graphiti

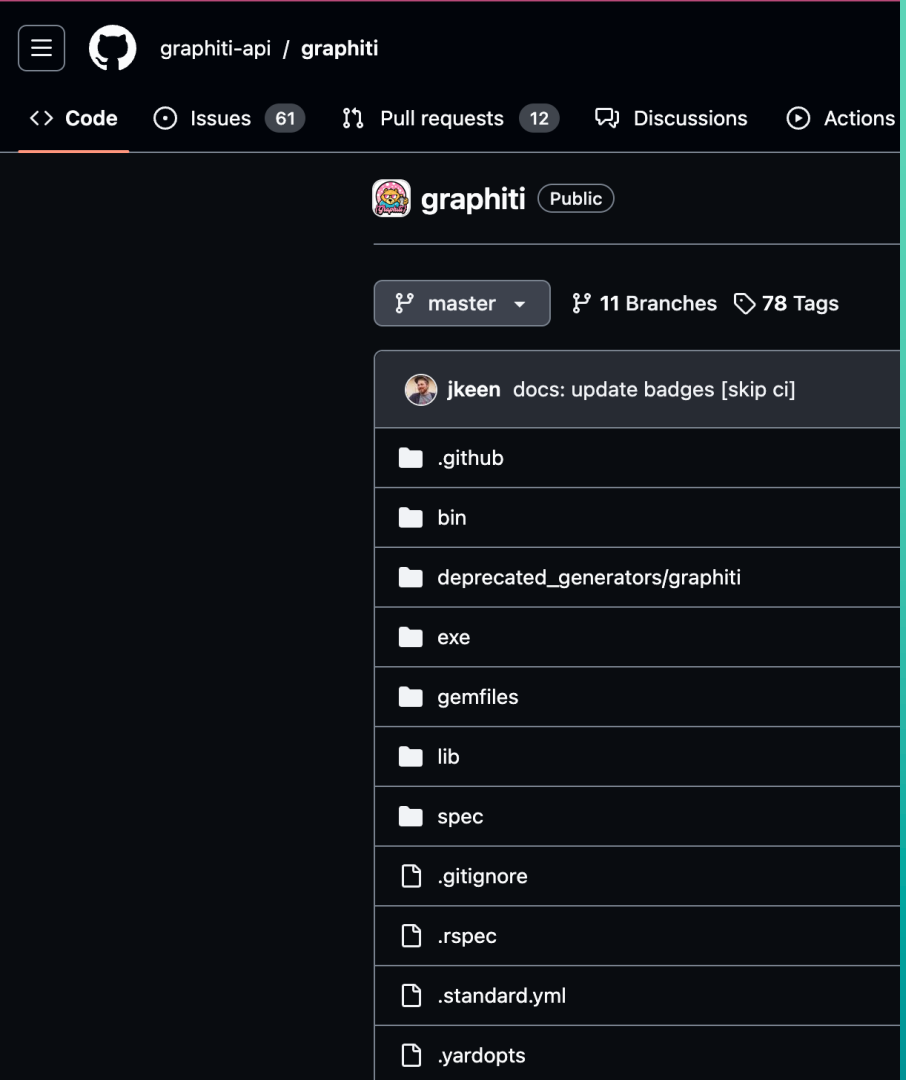
What is Graphiti?

A Ruby library that allows you to create JSON:API compliant graph APIs quickly.

Why Graphiti?

It supports building Graph APIs that are ALSO RESTful.

Check it Out: The Docs



Pro Tip: Use an open-source library.

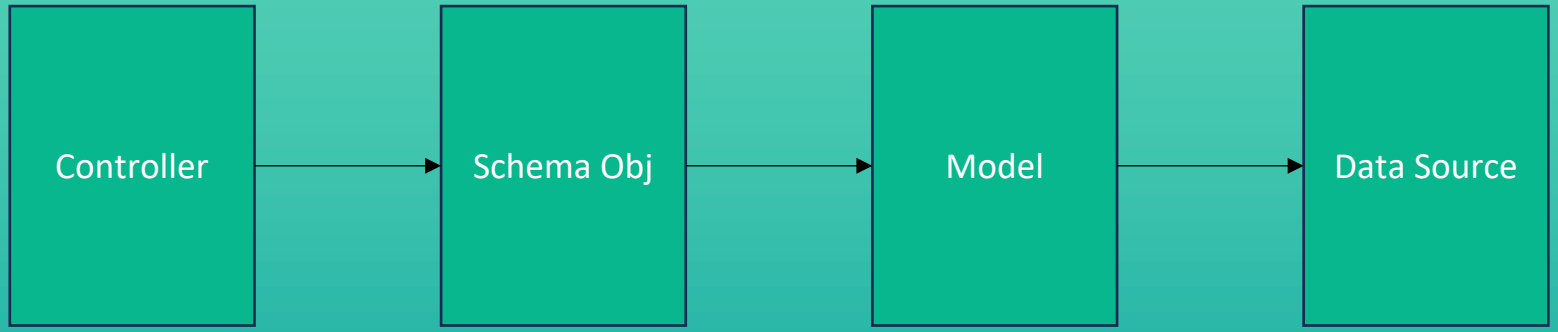
Hand rolling implementations of these specifications is complex and error prone.

Don’t solve solved problems.

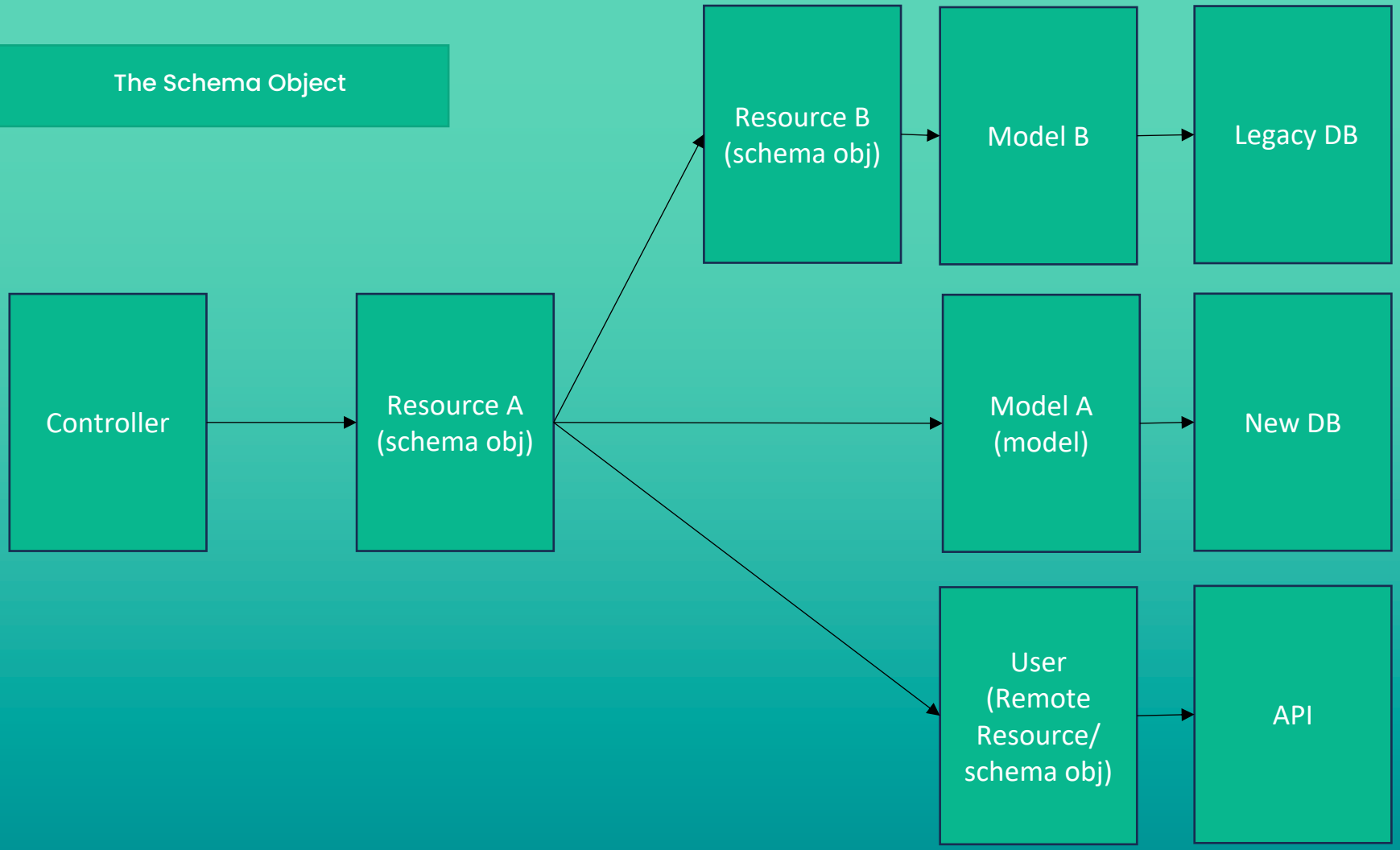
JSON:API and GraphQL are implemented differently but let's talk about at the structure of the applications that they create.

- Schema objects exist that declare the API schema like how database schema files declare a database schema.
- The API engineers declare the schemas. They create the entry points, filtering abilities, and expose certain pieces of data.
- One or many objects exist that connect the schema object to the data that it supports.

The Schema Object



The Schema Object



What do the schema objects look like and how do they fit into the application?

GraphQL

A simple GraphQL API uses a single controller as the entry point and executes a query based on the POST body. A schema supports the GraphQL controller. This schema usually contains a reference to a root QueryType and a root MutationType.

```
def execute
    variables = prepare_variables(params[:variables])
    query = params[:query]
    operation_name = params[:operationName]
    context = {
      # Query context goes here, for example:
      # current_user: current_user,
    }
    result = GraphqlJsonapiComparisonSchema.execute(query, variables: variables, context:
context, operation_name: operation_name)
    render json: result
  rescue StandardError => e
    raise e unless Rails.env.development?
    handle_error_in_development(e)
  end
```

GraphQL

The QueryType class is where you will support the ability to query for certain records.

```
module Types
  class QueryType < GraphQL::Schema::Object

    field :students, [Student], 'Returns all students' do
      argument :lastName, String, required: false
    end

    def students(lastName:)
      return ::Student.all unless lastName
      ::Student.where(last_name: lastName)
    end
  end
end
```

JSON:API

A simple JSON:API backend will look more similar to the REST APIs that you may have built in the past. Your controllers will take in request params and build schema objects (here they're called resources).

```ruby
class StudentsController < ApplicationController
  def index
    students = StudentResource.all(params)
    render jsonapi: students
  end

  def show
    student = StudentResource.find(params)
    render jsonapi: student
  end
end
```

JSON:API

The schemas declare the fields and relationships that we want our API to have accessible. They have a supporting object (in this case an Active Record Model).

```
class StudentResource < ApplicationResource
  self.autolink = false
  self.model = Student
  belongs_to :students_class
  attribute :first_name, :string
  attribute :last_name, :string
  attribute :birthdate, :string
end
```

Here is a simpler resource model to show everything that this resource has. If our resource looks like the one on the left, our library adds the code that you see on the right. The full resource looks like the one on the right.

```ruby
class StudentResource < ApplicationResource
  attribute :first_name, :string
end
```

```ruby
class StudentResource < ApplicationResource
  filter :first_name do |scope, value|
    eq do |scope, value|
      scope.where(first_name: value)
    end
  end

  sort :first_name do |scope, dir|
    scope.order(first_name: dir)
  end

  paginate do |scope, current_page, per_page|
    scope.page(current_page).per(per_page)
  end

  def base_scope
    Student.all
  end

  def resolve(scope)
    scope.to_a
  end
end
```

JSON:API - Testing

When feature level testing JSON:API endpoints, it will look very similar to request specs for any other endpoint because it is RESTful. The specific library I chose has a lot of test helpers for parsing the JSON response. I'd recommend using these.

```ruby
RSpec.describe "students#show", type: :request do
  let(:params) { {} }

  subject(:make_request) do
    jsonapi_get "/api/v1/students/#{student.id}", params: params
  end

  describe 'basic fetch' do
    let!(:student) { create(:student) }

    it 'works' do
      expect(StudentResource).to receive(:find).and_call_original
      make_request
      expect(response.status).to eq(200)
      expect(d.jsonapi_type).to eq('students')
      expect(d.id).to eq(student.id)
    end
  end
end
```

JSON:API - Testing

You can test the resource directly. This is useful if you have complex filter functions that need to live on the resource.

```ruby
RSpec.describe StudentResource, type: :resource do
  describe 'filtering' do
    let!(:student1) { create(:student) }
    let!(:student2) { create(:student) }

    context 'by id' do
      before do
        params[:filter] = { id: { eq: student2.id } }
      end

      it 'works' do
        render
        expect(d.map(&:id)).to eq([student2.id])
      end
    end
  end
end
```

The Object Graph

- Just like the database tables have a structure of relations your API exposes objects in a specific structure
- The data is represented as objects. These are the nodes of the graph. The relationships between the objects are the edges of the graph.
- Your object graph can match your database schema, but it doesn’t need to

With graph APIs:

“you model your business domain as a graph by defining a schema; within your schema, you define different types of nodes and how they connect or related to one another”

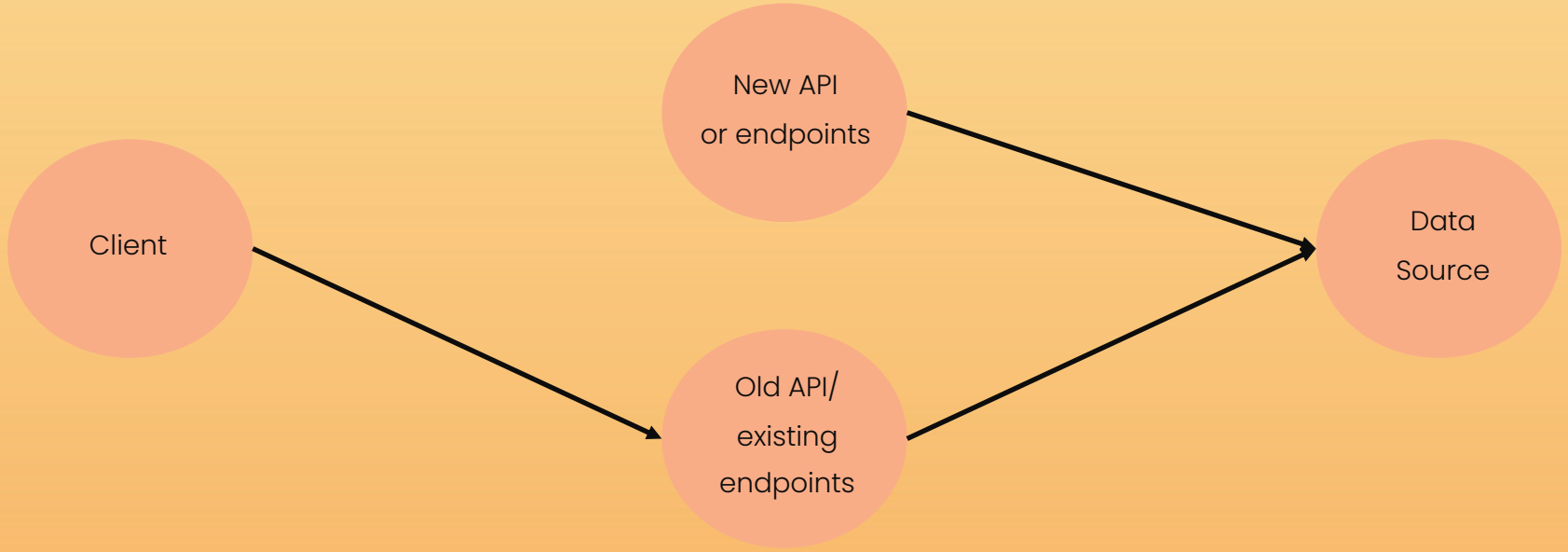
Over time this will make our API feel more consistent and cohesive.

How can you migrate to them

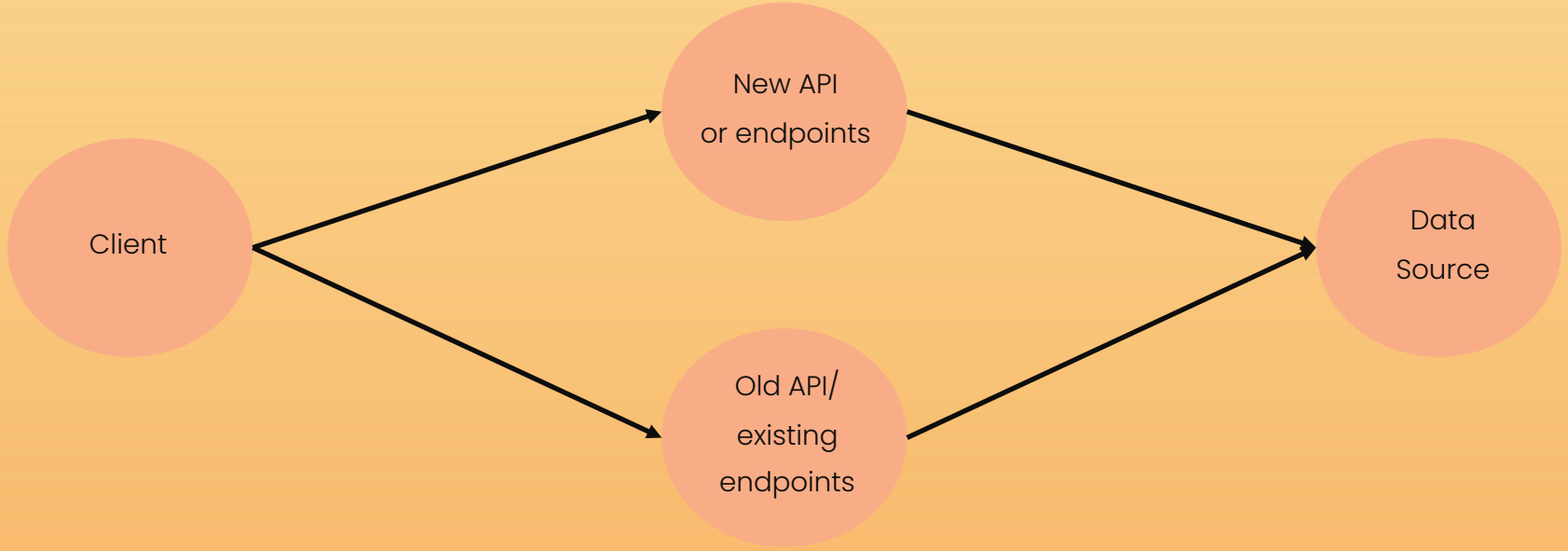
05

If you like this pattern but have an existing API that does not conform to these specifications - here are some ways you can move towards having a spec-compliant API.

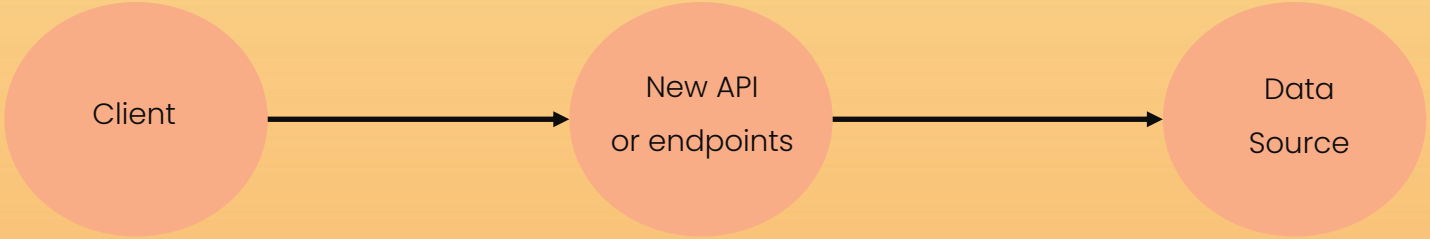
Your client can change



Your client can change



Your client can change



Twitter

- [Twitter Blog Article]
- [Twitter talk]

Which do you use?

{ json:api }

When to use JSON:API

1. When the client is leaning heavily into RESTful endpoints and would prefer to keep this pattern.

2. You want to take advantage of the default browser caching (or other REST-based functionality/tools)

When to use GraphQL

1. In the case of a migration of a standard REST API the client is able and willing to adjust a bit more on how they interact with your API.

2. Your client is React or mobile (a system like what GraphQL was developed) - GraphQL can work well in that ecosystem.

Caution: Exposing relationships means that clients can ask for them.

What we covered:

- What an API specification is
- Why we would want one
- How we would interact with one
- How we would implement one
- How we could migrate to one

All done!
The Presentation Is

Connect With Me

Thank you! ❤️

Email

lizdrennan8@gmail.com