Using Your Data Structure to Generate Your Web API

Benjamin Pfalzgraf Martin

March 16, 2018

Abstract

A Web API is a common component of many websites and even internal systems. For a company to compete in the world market they need to adapt quickly to evolving requirements from their clients and potential clients. Traditional approaches to building Web APIs lack resilience to change and do not naturally support multiple versions. This project attempts to answer the question: Is it possible to generate an API from your data structure that will both provide a well-designed web API and allow for rapid development cycles while preserving existing services? To do this, I created a production like environment by deploying Kuberenets, an automated deployment, scaling and containerised management system, on Amazon Web Services with internal load balancing. Then on that cluster, I created a proof of concept Web API that implements the RealWorld spec where the only bespoke component was the Structured Query Language (SQL) in a Postgresql database cluster. This proof of concept illustrates both that it is feasible to create an ideal Web API with a generic back end and how to securely do this in a production setting.

Contents

Glossary

Amazon Web Services "Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow." [aws17]. 2

API Application Programming Interface. 4

AWS Amazon Web Services. 2, 4, 10, 11, 55, 74

back end is any aspect of a Web API that is run on a server or cluster of servers. For this project it means both the API and the database layer.. 1, 3, 7, 52, 54

canonical name is a DNS record that points to another DNS record. 2, 55

CNAME canonical name. 55

code base is a self contained repository of code. 6

Continuous Deployment is often integrated within the same service as Continuous Integration, but is the component that deploys the results of the build somewhere. For this project we build the project inside a Docker container and deploy the resulting binary in another Docker container to Amazon Web Services (AWS) docker registry. 3

Continuous Integration is where a separate service from the developer's local development environment builds and runs the software's testing suit. 2, 3

DDOS is a distributed denial of service attack. Essentially the aim is to flood a service will so much traffic that it is unable to handle legitimate connections.. 2

Docker is a way to put software into a container that should only has the necessary libraries and setting required to run that particular software [doc]. 2, 3, 11, 55, 56

Domain Specific Language is a language designed to solve a specific problem or set of problems. 2, 4

DSL Domain Specific Language. 4, 6

EC2 Elastic Compute Cloud. 74, 76

EC2 Container Service is a way to host Docker containers on AWS so that other services on AWS can access those Docker containers.. 55, 74

Elastic Compute Cloud is AWS's virtual machine compute service. 2, 74

Elastic Load Balancer is AWS's service that not only load balances, but also handles SSL and a number of DDOS attacks. 2, 74

ELB Elastic Load Balancer. 55, 56, 74

front-end is any client or user facing interface. 3, 4, 7, 44, 52

git version control system. 55

GitLab is an open source unified interface for issues, code review, Continuous Integration, Continuous Deployment and your git repositories [git]. 55, 74

GitLab Runner Runs tests and sends results back to GitLab. 55, 74

Grafana is a visualization suite used for building monitoring dashboards [gra17a]. 3, 74, 75

GraphQL "is a query language for your API which lets developers describe the complex, nested data dependencies of modern applications" [gra17b]. 10, 13, 42, 47

Haskell is a purely functional language with a Turing complete type system. 6

Ingress Controller monitors ingress of resources to a system. 3

JavaScript Object Notation is a language independent data-interchange format that uses concepts from many C-family languages[jsoa]. 3, 4

JSON JavaScript Object Notation. 4, 15, 17, 48

JWT short for JSON Web Tokens is a method for representing claims securely between two parties. [JSOb]. It is an extension of OAuth 2.0. 15, 16, 18–22, 28, 30, 35, 47

k8 Kubernetes. 10, 11, 13, 14, 52, 55, 56, 75, 76

kubectl Command line tool to both deploy docker containers on Kubernetes and access the components in a Kubernetes cluster. 55

Kubernetes is an open source, distributed, micro-service oriented Docker orchestration platform that handles: distributing secrets, replicating application instances, horizontal auto-scaling, naming and discovery, rolling updates and other services not relevant to this paper [Wha]. 3, 6, 10, 74

Makefile is a file where the requirements to build a project are written. In my case I use them as glorified bash scripts. 55

NGINX is only used as an Ingress Controller in this project. It does: SSL termination, Path-based rules and load-balancing [ngi]. 55, 74, 75

OAuth 2.0 Authentication protocol [OAu]. 3

PostgreSQL is an object-relational database system made up of PostgreSQL Clusters [Posd]. 6, 9, 10, 12–17, 21, 22, 28–30, 35, 47–49, 52, 55, 74, 75

PostgreSQL Cluster is a single PostgreSQL instance with any number of PostgreSQL Databases [Posf]. 3

PostgreSQL Database is a container for one or more Schemas and Roles [Posf]. 3

PostgREST "PostgREST is a standalone web server that turns your PostgreSQL database directly into a RESTful API. The structural constraints and permissions in the database determine the API endpoints and operations." [Posa]. 6, 9–18, 26, 30, 42, 47–50, 52, 55, 74, 75

Postman is a GUI to help with Web API development and testing. 14, 16

Prometheus is a monitoring solution designed to work with Grafana [Pro]. 74, 75

RDS Relational Database Service. 74

RealWorld is a project to showcase multiple front-end and back end frameworks on the same specification, see `https://github.com/gothinkster/realworld` for more details.. 1, 14

Relational Database Service is a distributed relational database service by AWS.. 3, 74

Representational State Transfer is an architectural style for constructing Web API [Fie00]. 4

REST Representational State Transfer. 4, 7, 9, 10, 13, 47

role "PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant membership in a role to another role, thus allowing the member role to use privileges assigned to another role." [Pose]. 3, 22

Route 53 is a DNS service by AWS.. 14, 55, 74

S3 Simple Storage Service. 74

Schema is a grouping of named objects, data types, functions and operators. [Posf]. 3, 4, 6, 10–12, 14

Simple Storage Service is bit like dropbox for AWS. Its a secure cloud storage medium.. 4, 74

sqitch "is a database change management application" [sqi]. 9, 12–15, 74

SQL Structured Query Language. 1, 6, 9, 10, 12, 15, 22, 29, 35, 47, 48, 54

SSL is Secure Socket Layer that provides encryption between client and server. 2, 11, 15, 55

Structured Query Language is a Domain Specific Language (DSL) for expressing queries on structured data. 1, 4

sub-domain is the text before the domain name, but after the schema. For example in http://v1postgrest.gasworktesti the v1postgrest is the subdomain.. 55

Swagger Codegen generates Web API client libraries for a wide variety of front ends including mobile and web frameworks [swa17]. 52

Swagger Specification is a language agnostic way of describing Representational State Transfer (REST)ful Web API in either YAML Ain't Markup Language (yaml) or JavaScript Object Notation (JSON) [swa]. 14, 52

Swagger UI is the interface for interacting with a swagger-spec by running queries on the underlying Web API.. 14, 18, 52, 74, 75

table "Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type" [Pos17a]. 4, 10

type check means ensuring that the types of expressions are consistent.. 6

URI short for Uniform Resource Identifier, it is a syntax to uniquely address a resource with an optional query [Mas11]. 10, 47

view is a saved query associated with a given Schema that is treated like a table when used in a query [Pos17b]. 10–13, 15, 16, 23, 24, 29, 35, 37–39, 42–47

Virtual Private Cloud is a subnet that is protected from the global address space. To make something accessible from the outside world it must explicitly set as so.. 4, 74

VPC Virtual Private Cloud. 55, 56, 74

Warp is a Haskell web server [Sno11]. In this project it is used to force SSL. 55, 74, 75

Web API is an Application Programming Interface (API) served over HTTP or HTTPS. 1–4, 6–18, 22, 24, 29, 30, 33, 37, 38, 42, 46, 47, 51, 52, 54

yaml YAML Ain't Markup Language. 4, 55

YAML Ain't Markup Language "YAML is a human friendly data serialization standard for all programming languages" [yam]. 4, 5

Chapter 1

Introduction

My research question initially was how you can have the types from one code base propagate through to other parts of the whole Web API. This question comes from the fact that Haskell lets you encode both your Schema and your SQL queries in Haskell as DSLs that are type checked by the compiler [Sno17]. The effect is that the database Schema and SQL queries no longer get out of sync, since the compiler tells you when the types don't match up during type checking. However, this means that the database becomes a simple data storage mechanism with weaker data validation than the Haskell code serving the Web API. More importantly, the database is tied to a single version of the Web API. That is the core of the problem that arises when the database layer is merged into the Web API layer. The result is each version of the Web API has to migrate data from older versions without being able to coexist at the same time.

It is possible to handle both layers separately as has been done in traditional approaches, but that also comes with the problem that changes to the Web API do not propagate through to the database, which means features have to be implemented twice. This reduces overall agility and introduces subtle bugs from the discrepancies between the database layer and Web API layer. Which is why approaches such as the ones in Haskell have tried to merge the database layer into the Web API layer [Sno17].

The next obvious question is what happens when the Web API layer is merged into the database layer. This paper investigates that question by building a production ready example using a Kubernetes cluster for deployment and PostgREST as a generic Web API sitting on top of a PostgreSQL cluster. The Web API is effectively coded in PostgreSQL with SQL because it is the only bespoke component in this setup.

This discussion starts with a quick intro into web API design to frame the conceptional discussion and proof of concept. Following from those I evaluate the resulting design and reflect on the process as a whole.

Chapter 2

Web API Design

Web API design takes into account a wide variety of concerns, the full scope of which are beyond this project. Instead I provide a brief introduction to: design of RESTful Web APIs, security concerns and scalability only to frame the discussion about the conceptual discussion and the proof of concept I created. Each of these topics on their own have a plethora of books and papers written about them that are beyond what is needed to evaluate my approach.

In very general terms, the purpose of a Web API is to provide an interface for either external or internal developers to build another layer of abstraction on top of that Web API. That abstraction could be a front-end in Swift made to run on iOS or another back end layer that provides its own Web API. As a result the Web API needs to be able to serve a wide variety of use cases and do so in a way that the developer using the Web API does not need to be concerned about scalability and security of the underlying implementation. At the same time the Web API needs to be easy for a developer without domain knowledge to use the Web API. Fundamentally a Web API is built for other developers and its quality is determined by how well other developers can use the Web API to accomplish their goals.

2.1 RESTful Web-API Design

RESTful Web APIs have three main aspects that are relevant to my project: Objects are named with nouns and accessed using single slug with parameters after the slug that narrows down which part of an object the developer wants access to [Mas11]. Actions are operations on the data that return no information; they are named with verbs [RES11]. Finally, the collection of these objects and actions should allow for the developer to accomplish any task that is a valid use of the data. These are the criteria I will use to judge the expressiveness of the proof of concept.

2.2 Data Security

There are three levels to data security. The most important is unauthorized access to data, where an individual is able to bypass security measures. The second is authorized access to data a user should not have access to. Finally, there is access to data that a user should have, but that access results in unintentionally breaking other parts of the system without notification. These three layers can be addressed separately, once you guarantee that there cannot be unauthorized access to your data you don't need to revisit the issue while making sure authorized users are appropriately restricted and so on. As such these layers will be addressed separately in the proof of concept.

A specific exploit I will talk about is mitigating SQL injection. This exploit uses the fact that any layer sitting on top of a database must communicate with the database in some format that will be executed on the database. If an attacker can modify what information is being sent to the database they can effectively gain access to anything the Web API has access to in the database. Usually this flaw is only exploitable if the Web API does not validate the input it receives or does not access the database using prepared statements.

2.3 Scalability

There are two aspects to scalability that matter here: is it possible to distribute the Web API across multiple machines and how much computing power per request is used. In a nutshell, can the system handle any number of requests if given enough machines to and how many of those machines will the system need for a given number of requests per second. The later is hard to test, since the load depends on the type of request and many other factors may contribute to the overall performance. For the purpose of this paper only the first concern will be addressed.

2.4 Summary

A Web API is designed for other developers to solve their problems. It should use a combination of nouns and verbs to access objects and functions. Security can be addressed in layers and specifically SQL injection should be mitigated. Finally, scalability is an aspect of the system's design that is important in distributing the system across multiple machines.

Chapter 3

Concept

The concept is: put all data requirements in a PostgreSQL database which is using SQL for what it does best, expressing requirements of and relationships between data. Then use PostgREST to act as the conduit between the PostgreSQL cluster and the outside world. That way the database is never out of sync with the Web API since it is the Web API. Plus there is no need to write bespoke code to provide access to the data in the database. To do this effectively, though, the database itself needs version control for the same reasons any Web API needs version control. I do this with sqitch which gives me the power to describe my changes to the database incrementally, verify that a change worked appropriately when added and revert back to a previous configuration. The overview of this system is in figure 3.1.

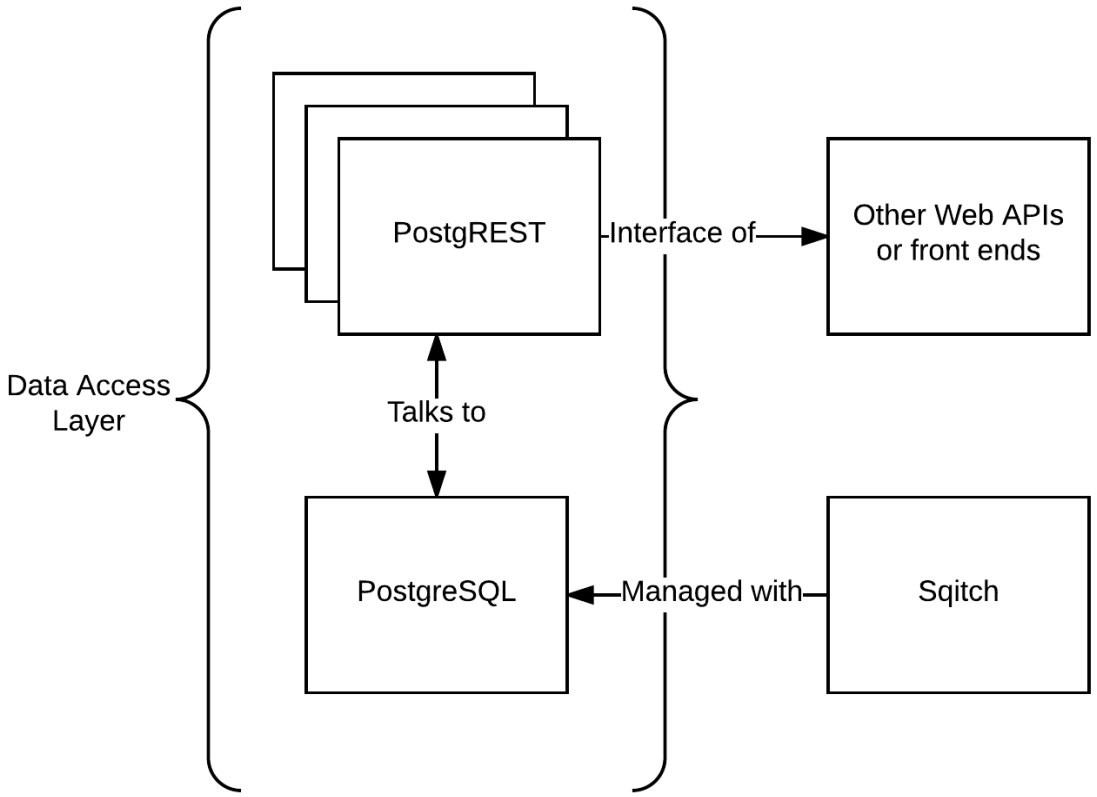

Figure 3.1:

Before talking about the deployment and advantages of this system there are a few concerns that need to be addressed: how does this system's RESTful Web API give access to the data in PostgreSQL? how will this system address the three levels of security? how will this system scale?

3.1 PostgREST's RESTful Interface

PostgREST not only follows the REST guidelines, it also augments it with some influences from GraphQL. PostgREST creates routes for each view, table and function accessible through the Schema it is provided. It's up to the developer to name functions as verbs and tables/views as nouns, but that is good practice for SQL as well. To specify which row within a view or table to access the developer sets parameters after the object's slug in the URI.

```
profiles?username=eq.test3&select=favorited
```

The above would only return rows where the `username` was equal to "test3" in the `profiles` table and within those rows only return the "favorited" attribute. The slug in this case is `profiles`. This would be the case if the GET HTTP verb was used, but if instead PATCH was used and there was a body with updates for any attribute in `profiles` then the request would try to modify that object, returning either just 204 or an error. In fact the same URI can have always have up to four different meanings. If paired with DELETE then that tries to delete all matching rows, while POST tries to add rows for each object in the requests body. Importantly, PostgREST is only trying to preform each request and its up to the PostgreSQL database to decide if they should be able to complete that request.

Not only does PostgREST provide the normal RESTful access it also understands references to other tables. In this example, the `article`'s table has a reference to the profile that created the article in the `profile` table. So the developer can make a request like so:

```
articles?select=*,profiles{*}
```

This gets all attributes of the `article` table and traverses the reference to get the profile of the author for each article. This is very much like how GraphQL Web API was designed to function. However, unlike in any other bespoke implementation, PostgREST was not modified for this particular Web API. Instead it was used as a generic tool to expose these routes to other developers.

The full range of possibilities that PostgREST can do can be found at the citation [Posb].

3.2 PostgreSQL's security features

The three main aspects that make the whole system possible are: row security policy, view's `"security_barrier"` and specifying different attribute level permissions for each type of action. The row level security adds two boolean functions, one to check for what rows can be viewed for a given action and one for what rows can be modified by a given action. Naturally only SQL's UPDATE needs both, but this allows for very fine grained requirements. Combined with `"security_barrier"` views can be safely added as further restrictions on top of the underlying table with row level security. Finally the developer can specify different sets of columns for SELECT, UPDATE and INSERT. The only caveats are that the views must be created under a role without superuser powers that has access to the underlying table.

The figure 3.2 illustrates what PostgREST has access to: one Schema that has a collection of views and functions where each view has an RSP, row security policy. Each view and function are created with an `accessor` role that can access the underlying tables and functions that implement the functionality. That way the attack surface is limited to anything in the view and whatever the `accessor` can touch. The `accessor` cannot bypass row security policies nor can it use the privileges of the functions that it has access to. That means the superuser can create functions to give the `accessor` restricted access to protected tables securely and any tables that the `accessor` can access are still restricted at the row level. This design was one of the critical, but missing details from the original PostgREST design I added.

3.3 Kubernetes Scalability

To prove that the design is scalable the proof of concept is deployed in a Kubernetes (k8) cluster on AWS. The full design is shown in figure 8.1 and the details of the deployment are found in the appendix 8. Ironically, designing and implementing the deployment was more complicated and

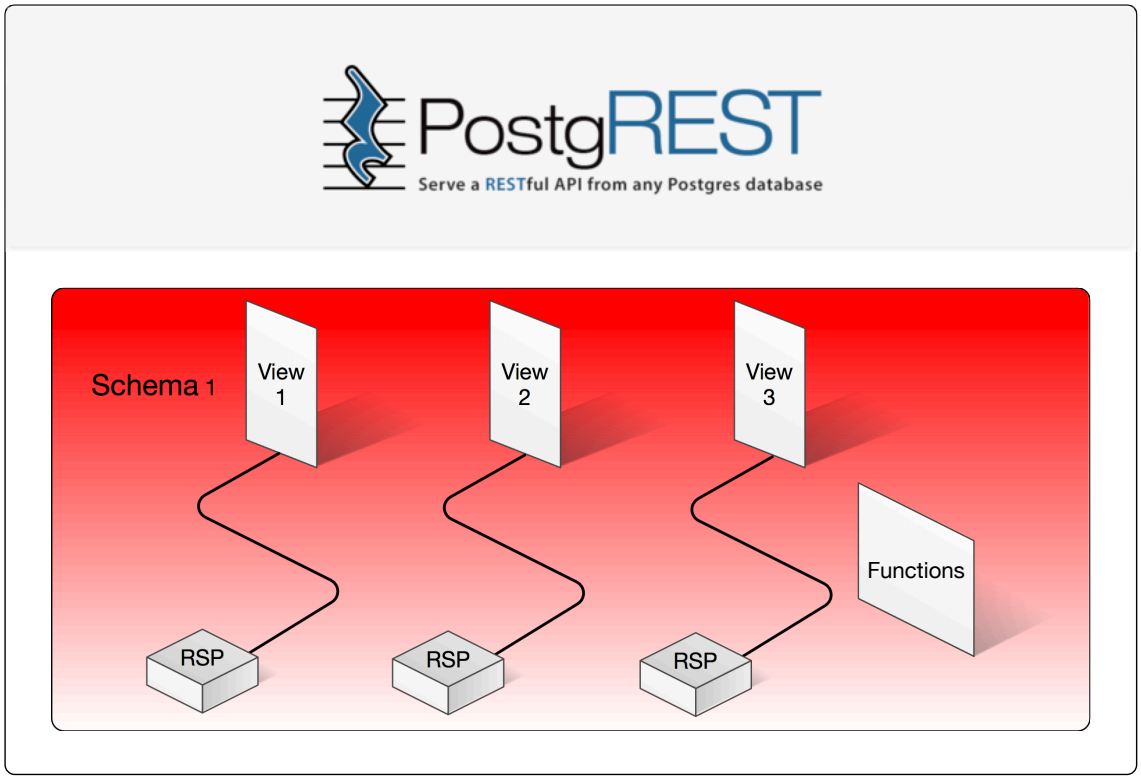

Figure 3.2:

difficult than designing and implementing the Web API to provide secure, restricted access to the underlying tables. Deploying anything to production is a nightmare. However, the deployment is responsible for SSL, load balancing, scaling up resources, monitoring, rolling updates and doing all of this in a secure environment. The proof of concept could not have multiple versions or have secure authentication without relying on the deployment.

k8 provides another important feature, each cheaply run unused or rarely used versions. Since k8 runs Docker containers for each service instead of dedicated virtual machines, the cost of keeping a version accessible is the cost of running at least one Docker container on the cluster. That cost is much lower than an entire virtual machine for each version. Also, the side effect of having k8 is that it can run other bespoke components of a system along side the Web API that can interact with the Web API or provide extra functionality that is not derivable from the data. Overall k8 provides the flexibility to solve the developer's problem with whatever solution makes sense for them, by providing a platform that make orchestrating Docker containers easy.

3.4 Multiple Versions

Having multiple versions of the Web API is handled in two steps: First there needs to be a separate PostgREST service on k8 for each version of the Web API. Second the database needs a new Schema with views of the data that should be accessible, re-exports of the functions and new roles for each type of user. Its important to note that existing users do have to be manually added to the new groups, this is something that could be automated in the future.

The first step is made easy with a working k8 cluster and some scripting with AWS's command line tool. Essentially to have a new PostgREST service a new k8-deployment is created, with its own k8-service and k8-ingress, see Appendix 8 for details. Once one version is working duplicating it for other versions is trivial.

The second step assumes that there is already a version of the Web API deployed in production. If this is the first version to be deployed then the underlying tables and authentication need to be setup as well. Otherwise its a mater of copying all files from the previous version's Schema, renaming everything to the new version, switching to new group roles and then making any new changes or additions that are necessary.

An example of what this might look like in practice is in figure 3.3. All tables are hidden in their own private schema and can only be accessed through a view. New versions can have extra views of the same underlying data, expose new tables that were not accessible in the previous version, remove access to columns by not putting them in the new views or alter existing tables to have more columns. Those columns must have a default and be "nullable" to be compatible with existing view. In the figure 3.3 it shows "View 2" and "View 3" are shared over the same Schema, although this is the result of making a new version no views can be shared between versions because that would require roles to access the older version's Schema and thus break the complete separation between the two versions.

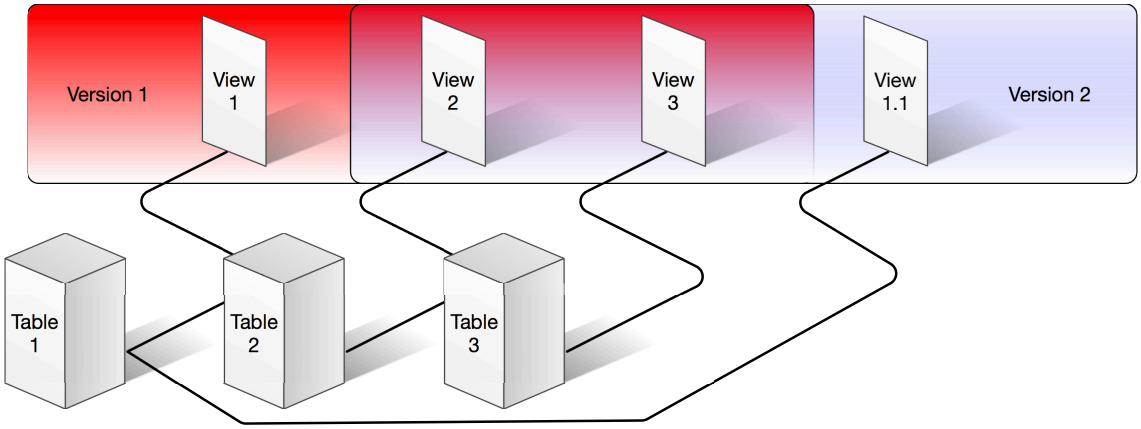

Figure 3.3:

The complete separation is not only between versions, but also between the underlying tables and all functions. The `accessor` role keeps the separation between the underlying table structure and the Web API's interface because it is a role PostgREST does not have access to and is the only role that can preform operations on tables and execute functions. The only way PostgREST has any access to data is through views and functions the `accessor` creates. However, no roles that PostgREST can switch to are granted access to those views and functions, instead each user's role is a member of a role that has restricted access to those views and functions. Importantly each function the `accessor` has access to is a re-export of a function from a private schema. This means functions can have different implementations in each version and yet still have the same names. The result is both fine granted access controls on a per group basis and all access is being done through an underprivileged role that must respect row security policies.

The above design I created is to both enforce security and provide as much flexibility in what can change between each version as possible. The only aspects that cannot change between versions are the row level policies on each table and adding columns that cannot be null or do not have a default to tables that are exposed as a view in a previous version. However, everything else can be changed. It is even possible to add stronger row level policies by adding a "where" in the new exposed views. As a result the design could be built without row level security and depend entirely on views to restrict data access. This flexibility is quite profound and allows each new version to have access to the same data that all previous versions had access to.

3.5 Database Versions

The methods to have multiple versions working off of the same data are complex and could easily be written incorrectly, so not only having versions of the database's design are important, but also verifying the design adheres to its requirements is critical. Using sqitch makes creating new versions of the Web API that operate on the same underlying data practical. In practice sqitch works by splitting the deploy, revert and verify components into separate files of SQL as seen in figure 3.4. Each time a feature or change is needed a new blank deploy, revert and verify file is created. A developer can then write each script and test them locally on PostgreSQL database with a copy of the online data. This means each change is considered immutable once it reaches production and can only be reverted by adding another feature. So, creating a new version of the Web API is a matter of creating the scripts for the new components and any alterations for tables.

Rolling back to a previous version on a database in production becomes reliable and predictable as long as the scripts are properly tested and considered immutable when already in production, all that sqitch has to do is run the revert script between the current state and the desired state.

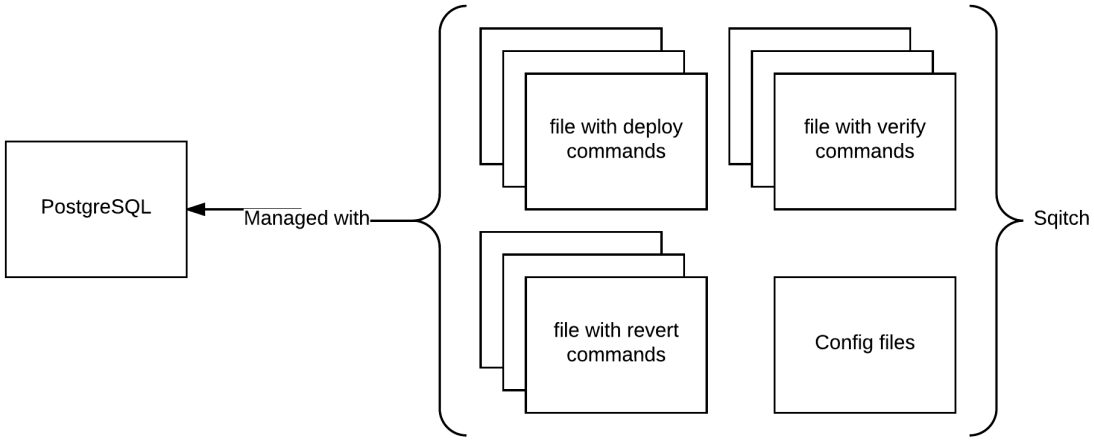

Figure 3.4:

3.6 Players in This Space

There are several companies and individuals chasing the dream of getting a Web API from the structure of the underlying data. DreamFactory which can be found at: `https://www.dreamfactory.com/`, provides an incredible amount of features including: Generating a RESTful Web API from any MySQL or NoSQL database. A GUI for user management, interactive documentation and monitoring/rate limiting routes. On top of that it handles the business logic by allowing the developer to add Python, PHP and JavaScript [Dre]. However, the project is still merging the database layer and Web API layer in the other direction. The project, written in PHP, is a stateful program in and of itself that treats the database as just a data store. This makes horizontal scaling essentially impossible. The reason PostgREST is easy to scale on k8 is because PostgREST is stateless.

Another interesting attempt that is not open source is GraphCMS found at: `https://graphcms.com/`. They provide an interface to design your data structure in a constrained manner, but allow quires in GraphQL which is very expressive [Gra]. They say it is secure and scalable, but there is no evidence they provide that shows this is true.

Finally, PostGraphQL, similar to PostgREST is an open source project that provides a GraphQL interface to a PostgreSQL database while leveraging the built in security features of PostgreSQL [posc]. Its written in JavaScript/TypeScript, but the authentication model is missing how to create role's dynamically, complete separation between the user table and the outside interface and modularity in general. Their design would benefit greatly from the concepts described in this paper.

3.7 Summary

A core concept in this paper is how to design a database structure in a secure modular way that allows for multiple views on the same data. Everything from the deployment to PostgREST are just tools to make that data available with secure communication and in a scalable environment. Even sqitch is just there to help deploy and revert that design. In the Proof Of Concept you will see this effect in practice and how important the underlying concepts of the design are.

Chapter 4

Proof of Concept

This chapter illustrates one way to deploy a system as described in chapter 3 and an example of the system in use. The overall deployment is shown in figure 8.1 and the detailed view of the k8 cluster is shown in figure 8.2. The deployed system is generic in that it does not impose any restrictions on what is put in the PostgreSQL instance, how many PostgreSQL services there are, what the PostgREST services provide as an Web API or what other services run on the cluster. To illustrate what this system is capable of version 2 of the Web API has equivalent functionality to the RealWorld Web API spec 8.1.3 using the generic system I setup.

The following sections cover the result of generating the RealWorld Web API from the PostgreSQL database's structure and functions. To know how the system was deployed and details on how the internals work, see the appendix 8. This chapter is broken down into two halves: The first includes sections on the external access and adding features to the Web API and authentication. The second is concrete examples of the Web API in practice. Each section is about how the RealWorld Web API is implemented using the system described in the Concept 3.

4.1 External Access

In terms of external interface there are two main types of routes. First there is `https://v1postgrest.gasworktesting.com`, the domain "gasworktesting" is not special, it is just a domain I had access to on Route 53, and `https://v1swagger-ui.gasworktesting.com/`. The first is the endpoint for the Web API and the second is the ingress for Swagger UI that lets you explore the Web API. There can be as many versions as desired, see 8 for how to deploy more versions.

Unfortunately, Swagger UI by default does not let you add headers to your requests, requiring you to write your own custom interface to add it in. Also, Swagger UI does not include all possible queries that PostgREST can handle, missing the vast majority of the Web API ability. However, it does show the user what responses can look like, gives access to the portions of the Web API that do not require authentication and a full description of all the objects that the Web API handles. Therefore, Swagger UI is still useful, but I still have to use Postman to supplement Swagger UI which can also interpret the Swagger Specifications.

When accessing the `https://v1postgrest.gasworktesting.com` the request is routed to a PostgREST instance that then handles the query by possibly making requests to a PostgreSQL instance, for details see 8. That endpoint, with no extra parameters, returns the Swagger Specification generated from the exposed schema in the PostgreSQL cluster. To expose a Schema the PostgREST instance just needs a role with login credentials that has access to that Schema.

4.2 Adding Features to the Web API

Without sqitch creating, testing and reverting changes to a PostgreSQL database is a nightmare. All database administration GUI fail to have versions for each change to the database. They may have rollback features on a single change, which just abuses PostgreSQL built in commit feature, but none allow fine grained deployment, reverting and verification for every change to the database. The reason sqitch is critical to the whole system I built is because it not only has the

above features, but also allows for tagging, keeps track of what it has deployed on which database and makes incremental development very smooth.

The work flow with sqitch starts with:

```
sqitch add <schema>.<table/function> -requirement <anything directly
used in either the deployment or verification> -n '<type: purpose>'
```

for example:

```
sqitch add private.articles -r private -r private.profiles -n
'table: stores article attributes>'
```

This creates a deploy/private.articles.sql, revert/private.articles.sql and verify/private.articles.sql files and sets the requirements for this feature to have the "private" schema and "private.profiles" table be deployed beforehand. Both the "private" schema and "private.profiles" table would have been created in a similar fashion. The idea to name the files based on what schema they were in came about because I knew there would also need to be "v1.articles", "v2.articles" and so on so each version could have their own view of the data.

Once you complete each of the files with the necessary SQL commands to create the new feature running

```
sqitch deploy
```

will run each file in the deploy folder that has not already been deployed in the configured database and run each corresponding file in the verify folder. If anything fails, the revert files are run to bring the database back to a consistent state. Sometimes the revert fails because of an error in the revert commands, but this happens rarely and in my experience been very easy to fix.

Once all the necessary changes are deployed to the database sqitch lets you tag the release so you can revert to in at a later stage. Add on the fact that these are just normal files that work perfectly with version control systems like git. The effect is version control on the schema, tables structure, roles, functions, triggers and data validation of the entire database.

4.3 Authentication

To understand this complex process, I first start with how to make sure a user is who they say they are. This requires that when a request is made to the Web API there needs to be a way for the request to claim who they are and a way for Web API to check that claim. Using JWT we can give the client a signed token with claims that the Web API can check for authenticity. The login process becomes: send an identifier, usually an email or user name, and a password to a login function in the PostgreSQL database. That function hashes and salts the sent password with the stored salt before checking it against the value saved in the password field. The reader should note that the password has to be sent in plain text, so SSL must be used between the client and the Web API. If their values are the same, then the login function creates a JWT of the user's role, an expiration date and the user's user name that is signed with a secret in the PostgreSQL database. This JWT is then sent back to the client.

This process is accomplished with PostgREST by exposing a login function in the PostgreSQL database to the schema that the PostgREST instance has access to. These functions then can be accessed with

```
/rpc/function_name
```

after the base url as a POST request where the functions arguments are a JSON object in the body of the request, see figure 4.1.

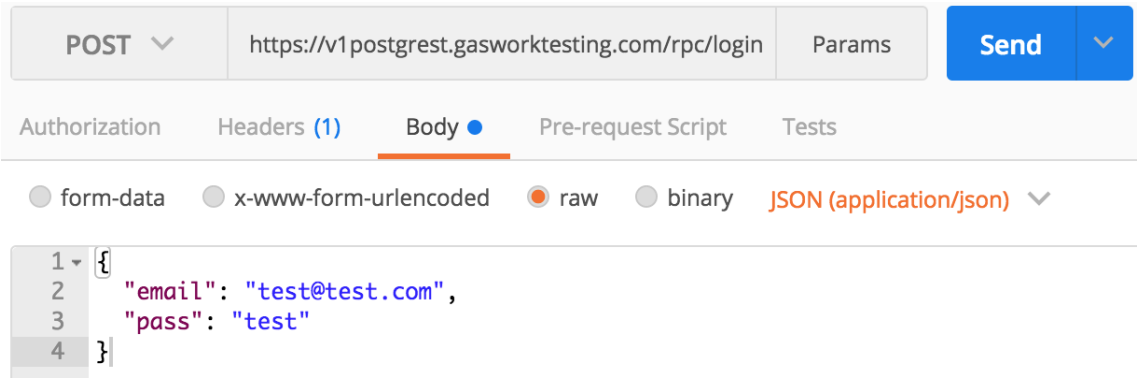

Figure 4.1:
Example of login procedure using Postman.

An example of what this would return is:

```
[
  {
    "token":
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoicm9sZTk3Z
     DAwYzM5Y2Y2ODQ4NTA4ZTU5ZjYzYzZjMzIxZWE2IiwiZW1haWwiOiJ0ZXN0
     QHRlc3QuY29tIiwiZXhwIjoxNDk1NjExMjI1LCJ1c2VybmFtZSI6InRlc3Q
     ifQ.ZStoE7BDD1i7Pv4wqCz1R8w7cpLB8WscGmMvbyzz1kI"
  }
]
```

The value of the token is the JWT.

Now that the client has an authentication token, the JWT, they can send that token in the header of their requests under the key, "Authorization" to validate themselves. The Web API then uses the same secret that the PostgreSQL database used to sign the JWT to check if the JWT has been tampered with. If the signature is valid then the Web API knows only the PostgreSQL database could have set the role value, expiration date and user name. The PostgREST instance can then switch to the provided role and execute the given query. That way it's up to the PostgreSQL database to determine if the role (user) should have access to the data they are requesting.

Now comes the next issue: how does the PostgreSQL database know what each role should have access to, given each user will need to have their own role. If done naively where each role is set to the user's user name, there is the possibility of clashing with built in roles. Plus each role would have to be added to every view it needs access to when it was created. Instead each role is given a unique identifier that serves as its role and when the role is created it is added to groups that have access to the correct views. Its important to note that the way the PostgreSQL database allows for an underprivileged role to both login and create new users is by providing access to functions that can do things the underprivileged role cannot. So the requirements for a function to create a new role are: create a unique role name, add the role to a group that has access to the correct views and add the role to the group that the PostgREST instance logs-in as.

There a few problems with this approach: every time a new group is created that login function has to change, plus it does not handle removing the created groups and roles when a user is deleted. Both of these can be better handled with triggers. The idea is for each group, when adding the group you also add a trigger on the users table that adds each user to group if they match the requirements. This still keeps the requirement that additions to the underlying tables do not affect the data currently there. If there was a need to update the data already in the tables, then that would be up to the database administrator on how that should be handled.

Now finally it is possible to add row security policies since there are claims in the JWT the tables can rely on for security. This is where SQL injection is essentially handled and or at the view level. The idea is, no mater what query a user is able to execute they cannot get past these barriers. To help with this, each user's role is given the least number of privileges possible to still access the features they need. Plus the role the PostgREST logs in as has a few privileges as possible to mitigate any vulnerability of PostgREST itself.

Now, each of the following examples illustrate the above concepts. I am running them in Postman for clarity, but each request could have been made with any program that can make HTTPS requests, like: curl, wget or even just in your browser.

4.4 Concrete Examples of the Web API in practice

4.4.1 Quick Introduction to Postman

Postman provides a nice interface to making requests to Web APIs. Below are two screen shots of Postman's interface that I have annotated with numbers. Each number corresponds to one component in the interface that is relevant to the examples in the next section.

At 1.0 in figure 4.2 you can set the HTTP Verb of the request. PostgREST only uses the verbs, GET, POST, PATCH and DELETE which nicely correspond to SELECT, INSERT, UPDATE and DELETE actions in a PostgreSQL database. PostgREST requires that all requests to functions in the database be POST requests and be under the `/rpc/`slug. It makes sense that all functions are POST requests since the JSON object sent in the body can be used to give the function's arguments.

At 2.0 in figure 4.2 you have the request URL, which includes both the base URL,

```
"https://v1postgrest.gasworktesting.com"
```

and the slug, "articles" or any other parameters for the request. 3.0 in figure 4.2 is to send the request; in all of my examples I have already sent the request before making the screen shot.

At 4.0 in figure 4.2 we have the header section which is where the authorization token goes, 4.4 in figure 4.2. The token itself is after the word "Bearer", but the full token is cut off by the window.

At 4.1 in figure 4.2 I have the Content-Type key that tells PostgREST what format I want the response in. It is always JSON for readability, but PostgREST can return XML as well.

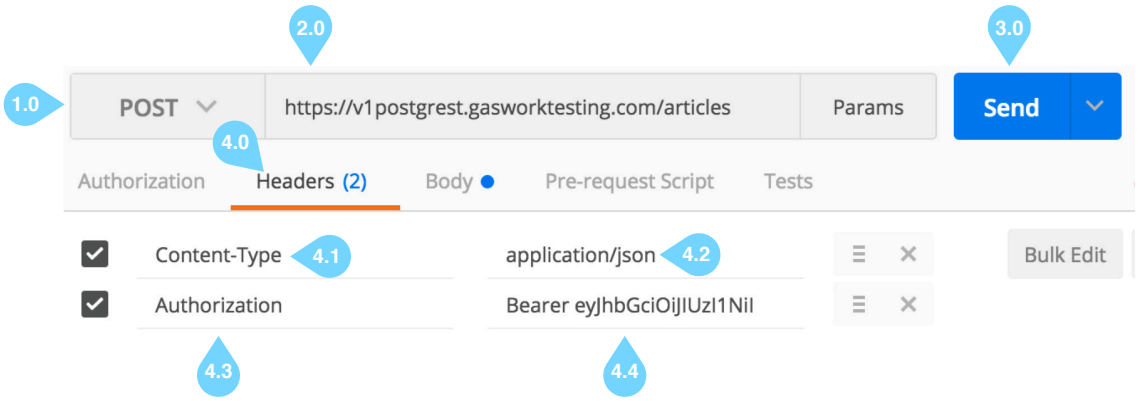

Figure 4.2:

At 1.0 in figure 4.3 Postman has the Body tab that lets us specify the body of the request. For all of the requests I use raw, as seen in 1.1 in figure 4.3, and set the content type to JSON, as seen in 1.2 in figure 4.3. Finally the body itself is written in 1.3 and the response I get back from the Web API is in 2 in figure 4.3 at the bottom.

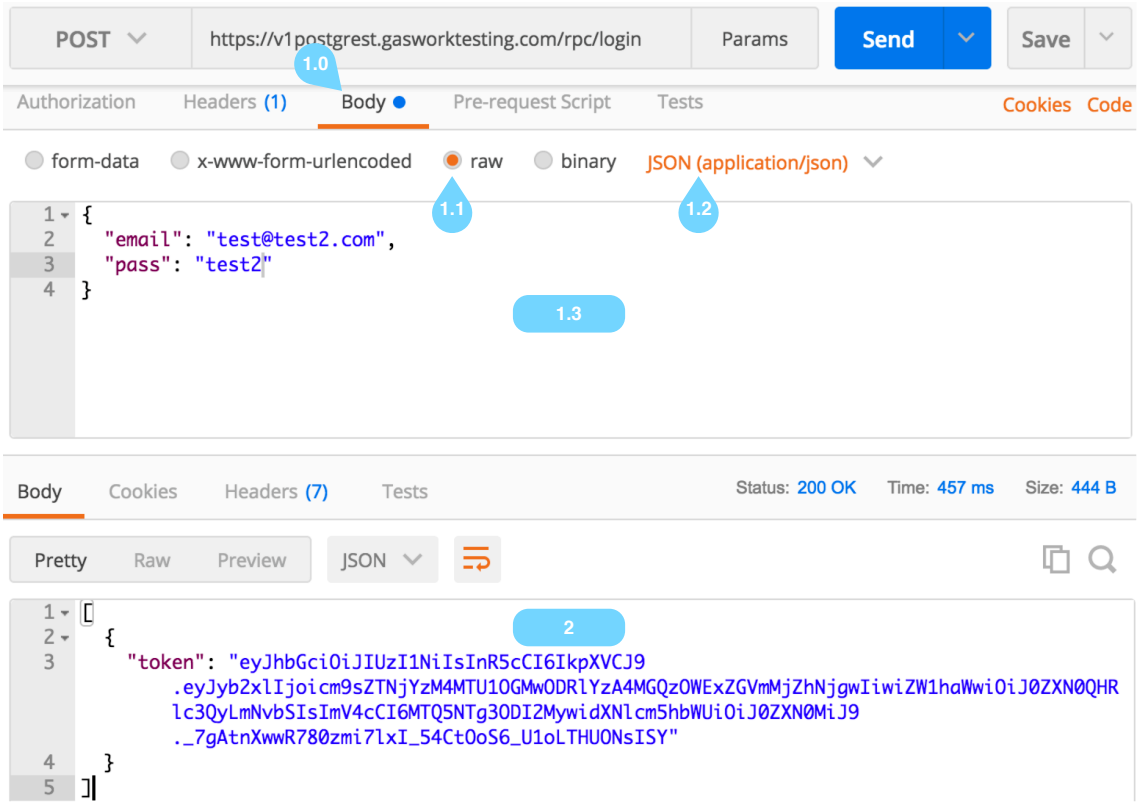

Figure 4.3:

4.4.2 Login and Posting an Article

The first step is to create an account as seen in figure 4.4. This request creates an entry in the users table with a generated, unique role name. Plus it validates that the email is in the correct format and that both the email and username are unique. If that insert was successful then the generated role is inserted into the `pg_authid` table, added to the `logged_in` group and the `authenticator` group. The authenticator group is what PostgREST logs into the database with and is set with "noinherit". It cannot access any tables or view, but can run "set role" to either "anonv1" if no JWT is supplied or the role specified in the provided in the request's JWT. Since you must make the request in figure 4.4 without authentication the `new_user` function is accessible to the `anonv1` role.

The `anonv1` is the anonymous role for the first version of the Web API. Whatever the `anonv1` role has access to is what the Swagger UI will show.

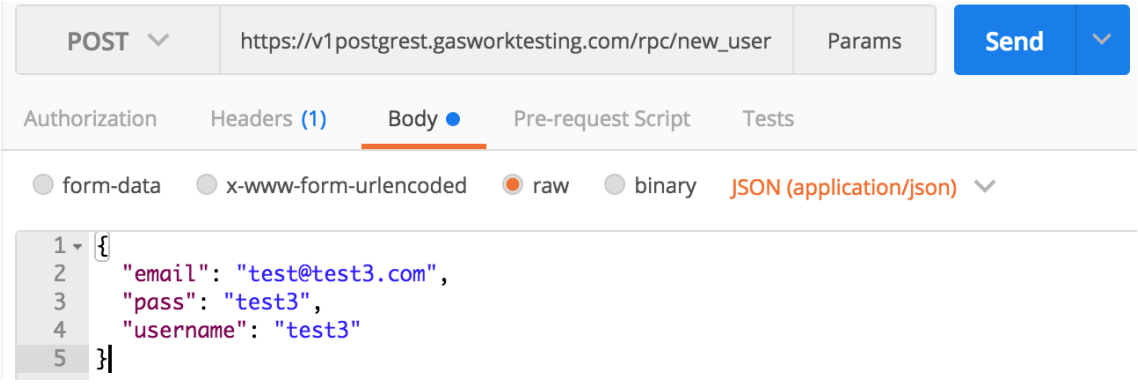

Figure 4.4:

Another trigger on the users table is to create a profile for the new user as seen in figure 4.5.

Profiles are accessible without authentication, so I can view them with just a GET request.

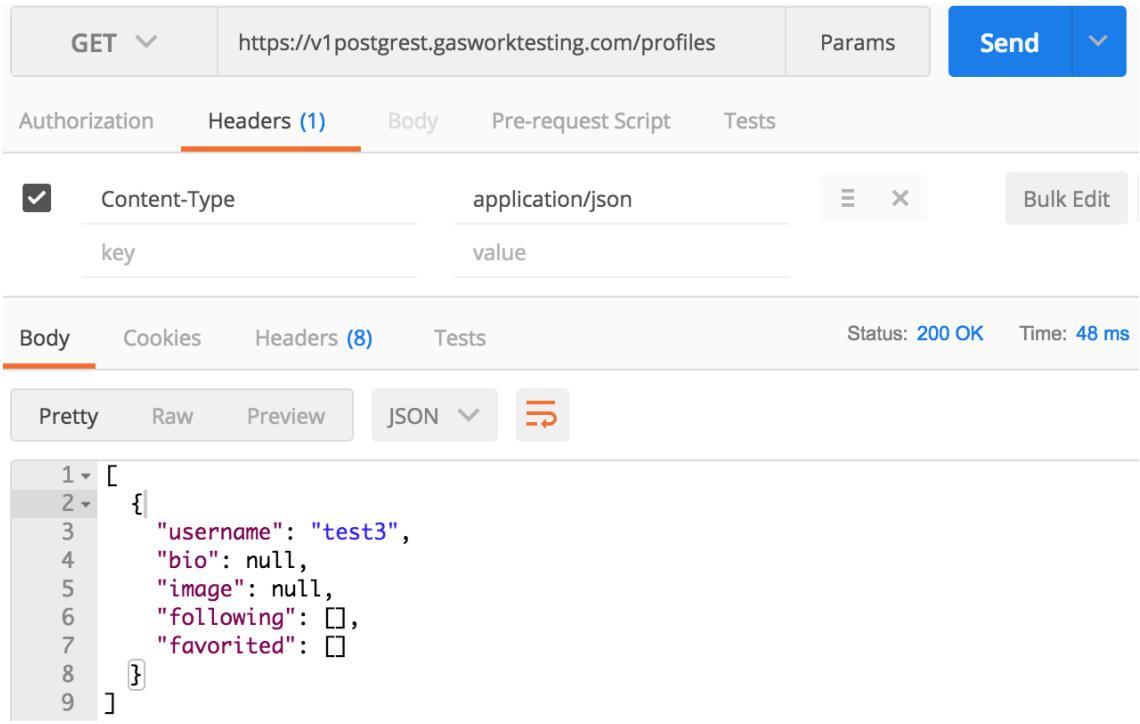

Figure 4.5:

In order to login I make a POST request to the login function as seen in figure 4.6. This function is also accessible to the `anonv1` role and without authentication. In the response I get back a JWT which is that long string of what seems like random characters after the "token" key. If you took that token over to `https://jwt.io/` you would find the payload is:

```
{
  "role": "role2da2a58dce7b4e92a18126792c6bb5f7",
  "email": "test@test3.com",
  "exp": 1495893690,
  "username": "test3"
}
```

That token is not encrypted, it just has a signature at the end to prevent the user from tampering with the contents. The important parts are the "role" which is what the `authenticator` switches roles to and the "username" which I use internally to determine if a user should have access to a given row.

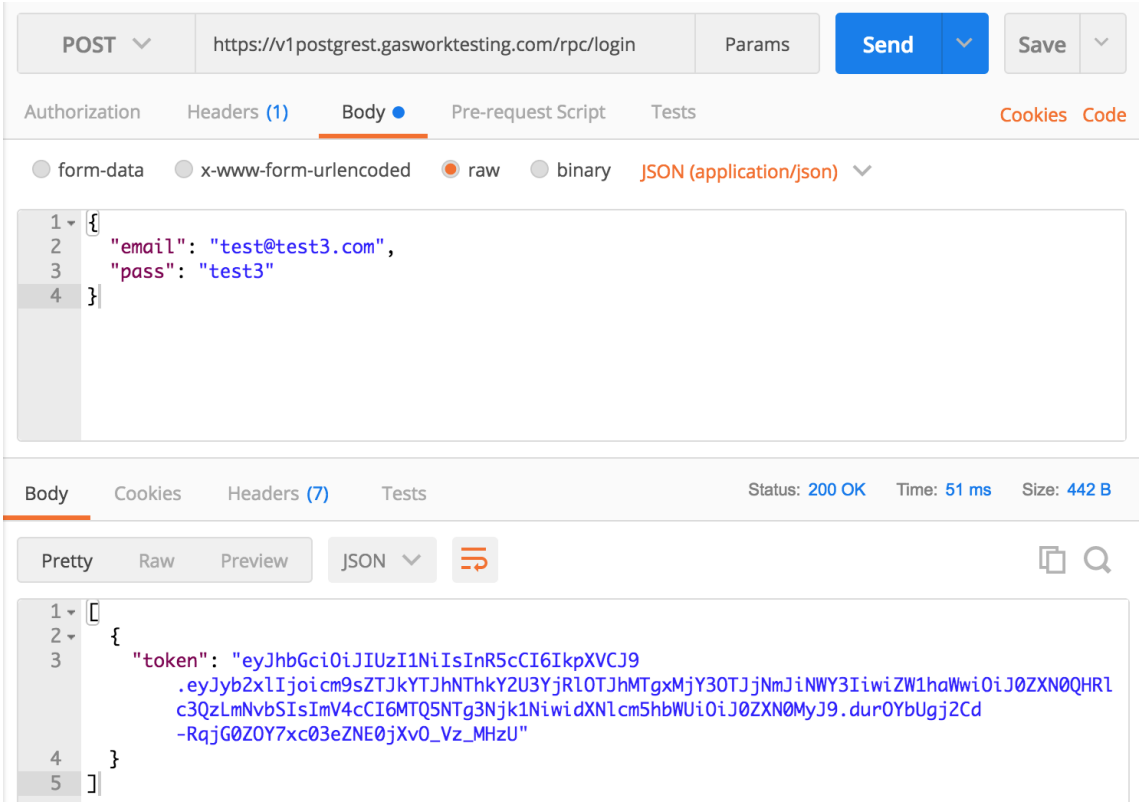

Figure 4.6:

Submitting articles requires a user to be logged in, which really stands for, having a valid JWT in the header of the request as seen in figure 4.7. The body of the request for creating a new article can be seen is figure 4.8.

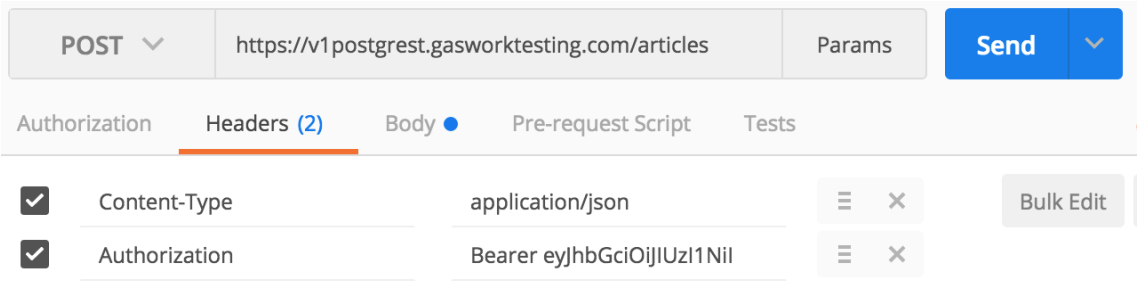

Figure 4.7:

Figure 4.8:

After submitting the new article we can get all articles by changing the HTTP verb from POST to GET as seen is 4.9. However, there are way more fields than we submitted before. The "taglist" has a default of empty if not provided, but the other fields cannot be sent in a request. In fact if you try to set any of the extra fields you would get back:

```
{
  "hint": null,
  "details": null,
  "code": "42501",
  "message": "permission denied for relation articles"
}
```

PostgreSQL lets me set access on a per field basis and per type of action. So any user may select all of the fields, but only a `logged_in` user may insert or update some of the fields. The reason is partly for convenience, but also for security. The createddat and updatedat are both populated from PostgreSQL's `current_timestamp` function, but "favorited", "favoritescount" and "author" are special. The first is the result of a function that checks if the `logged_in` user has favorited this article, but the other two are just normal fields. Later on I will go over the "favoritescount" field, but the "author" field is critical. That field is filled in by whatever "username" is in the JWT payload. This means I can be sure that the user who posted this article must have the "username", "test3".

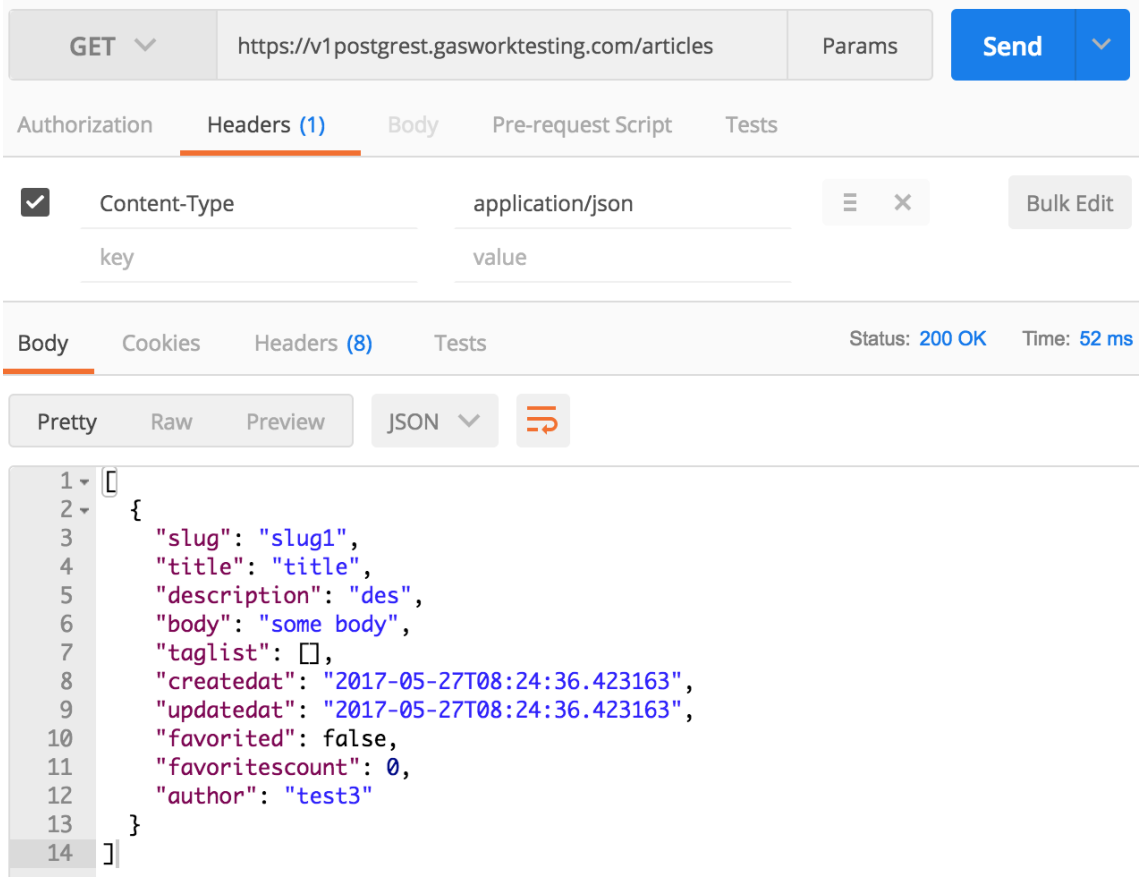

Figure 4.9:

In this example I covered the basics of how authentication works from the users' perspective and covered some details about the account creation and login process. This illustrates my proof of concepts authentication layer that ensures there is a clear separation between `logged_in` users and `anonv1` (anonymous) users. Its easy to restrict privileges in SQL based on what a role should have access to. As a result the complicated part of authentication comes from creating the JWT and putting roles into the correct groups, not setting what groups have access to what.

Note: groups and roles are identical in PostgreSQL since any role can become a group if another role is added as its member.

4.4.3 Row Level Security

This next section is about the second layer of security, what an authenticated user should have access to. I'll show you how row level security is used to restrict access to what the user can modify. It can also be used to restrict what a user can see, but this Web API coincidentally does not illustrate that.

In figure 4.10 I'll make another account.

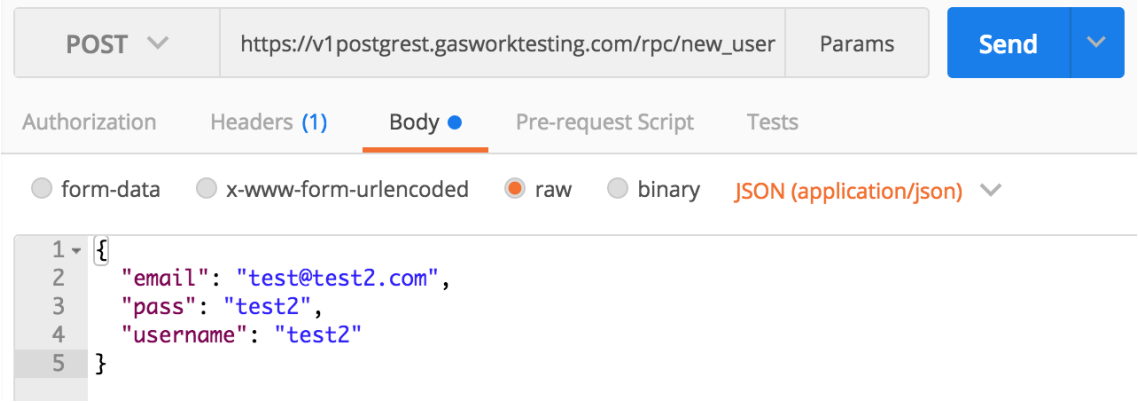

Figure 4.10:

The account we made in the previous example is still there as seen in figure 4.11. This GET request is not actually making a query directly on the underlying table. In fact, all of the previous GET and POST requests have all be operating on views of the tables. This creates an interesting problem, since views are run with the privileges of the user who created them. So they need to be created by an underprivileged user who still has access to the relevant tables. In order to make this request the `authenticator` first sets its role to `anonv1` which has only select access to the version 1 view of the profiles table. That table was created by the `accessor` role that has select, insert and update privileges on that table. So the query itself is run by `accessor`, but the `anonv1` role only can select from this table because it can only select from the view. Since neither a `logged_in` user or an `anonv1` should be able to delete their profile the `accessor` does not have that privilege.

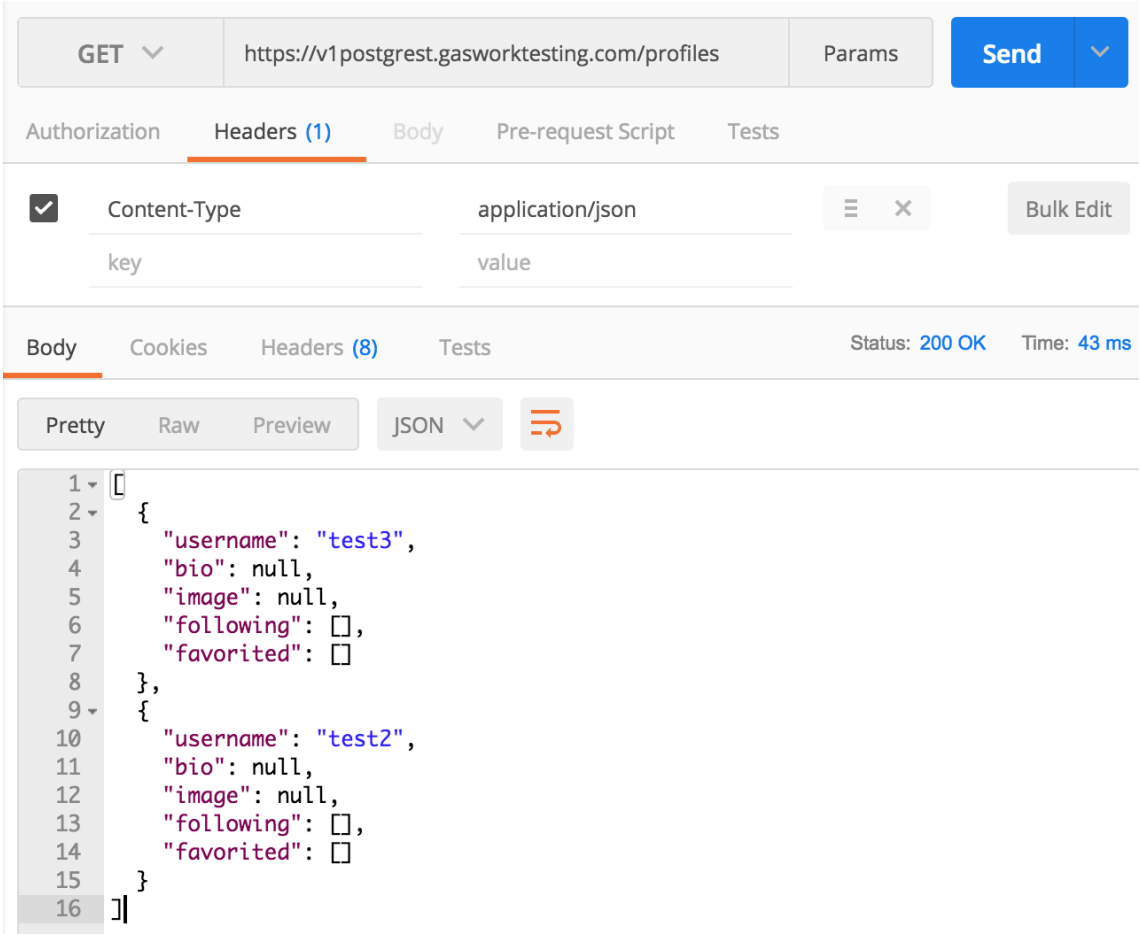

Figure 4.11: Showing all profiles.

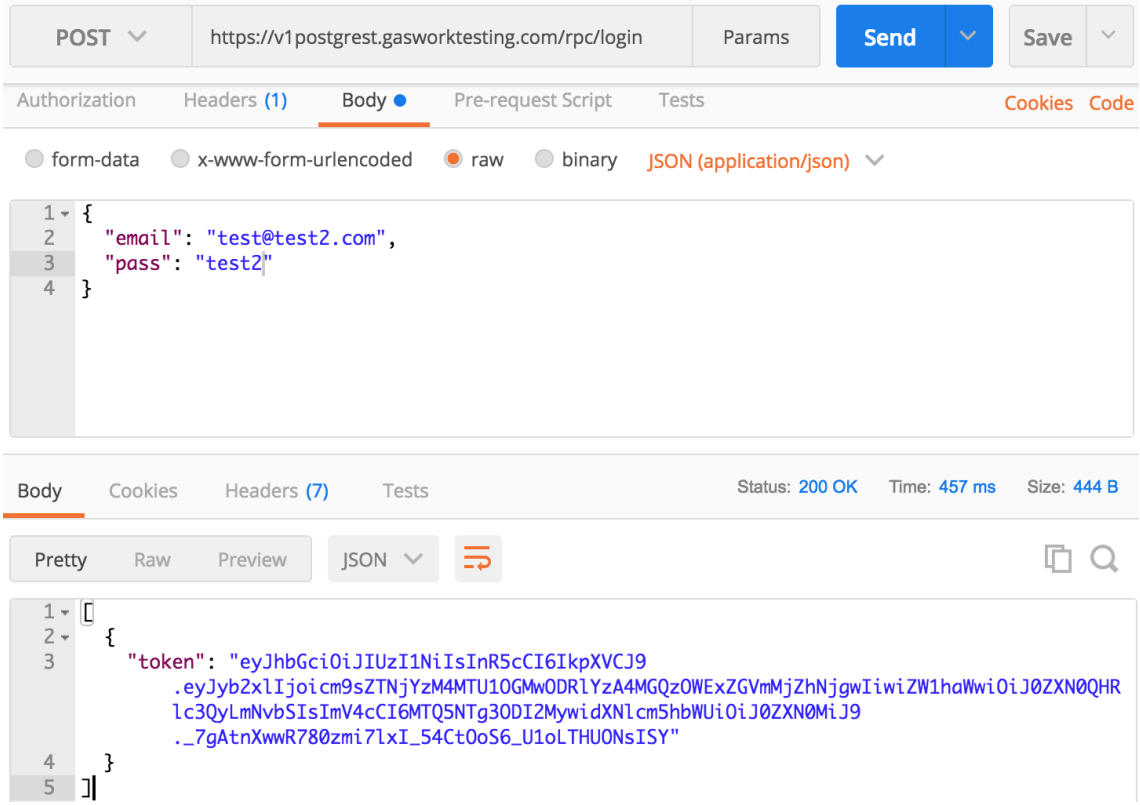

Figure 4.12: Here I login to the new account.

In figure 4.13 I'm showing you the `Authorization` token I got from the previous step. Then in both figure 4.14 and figure 4.15 I create two new articles with that token. Just like with the profiles table each request is being made through a view, the article's view is where `anonv1` and `logged_in` get their permissions to access the data. However, only the `accessor` is actually able to make the query. The `logged_in` role has its restrictions on inserting into the view, not the table itself. This is why each version of the Web API can easily have different access to the same underlying data without disturbing the previous versions.

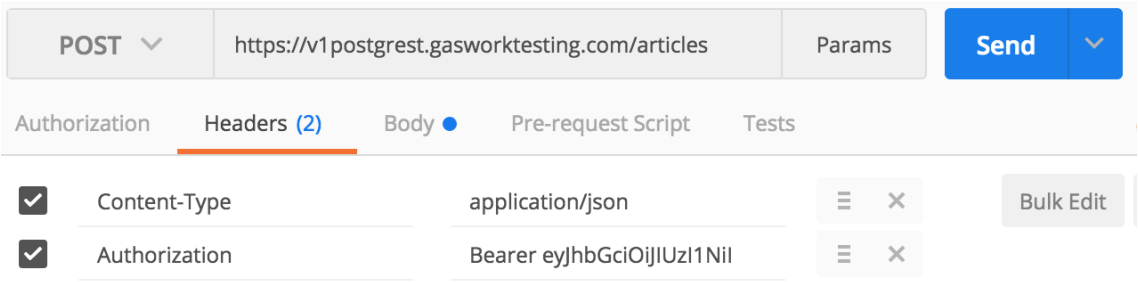

Figure 4.13:

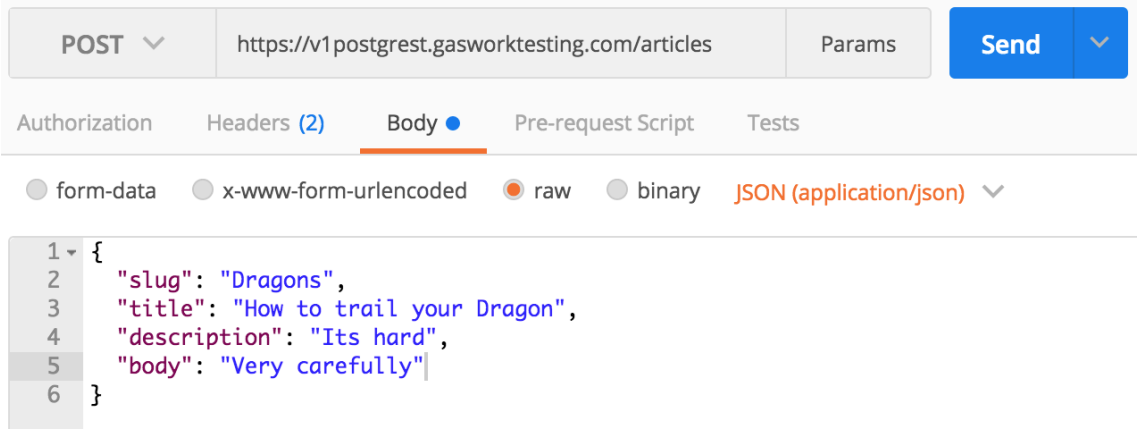

Figure 4.14:

Figure 4.15:

Now in figure 4.16 you can see all three articles. The first article is by previous user "test3", while the bottom two are the articles I just created in figures 4.14 and 4.15. Notice how the "author" has been set to "test2".

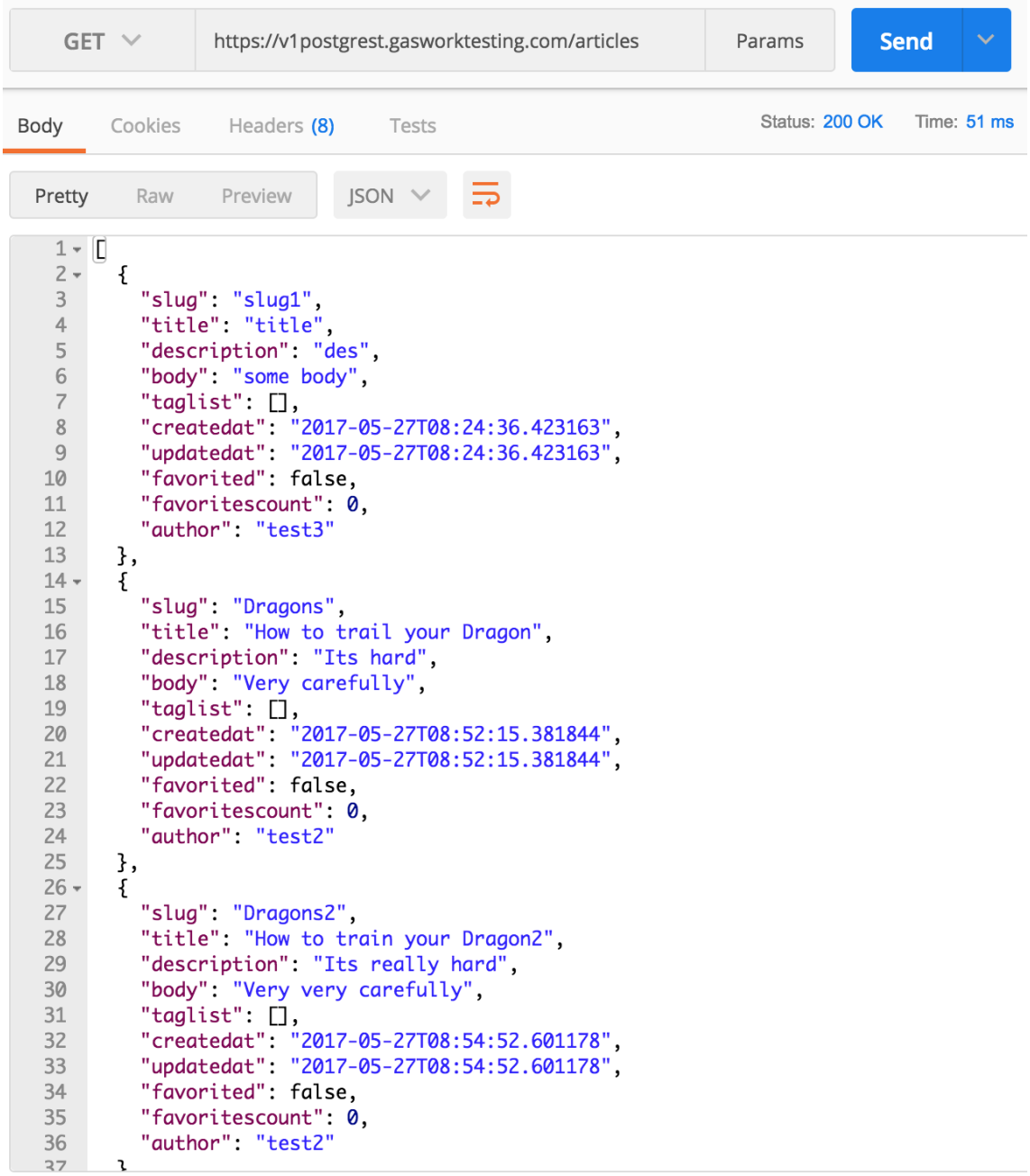

Figure 4.16:

However, I made a mistake when creating the title of the first article, it says "trail" instead of "train", let me fix that. In figure 4.17 you see there is more text after the "articles" slug. That narrows down the search results, or in this case narrows down what the POST request is applied to. The "?" allows for optional parameters such as specifying the requirement that key "slug" must equal "Dragons". The "eq." tells PostgREST that we are comparing using equality.

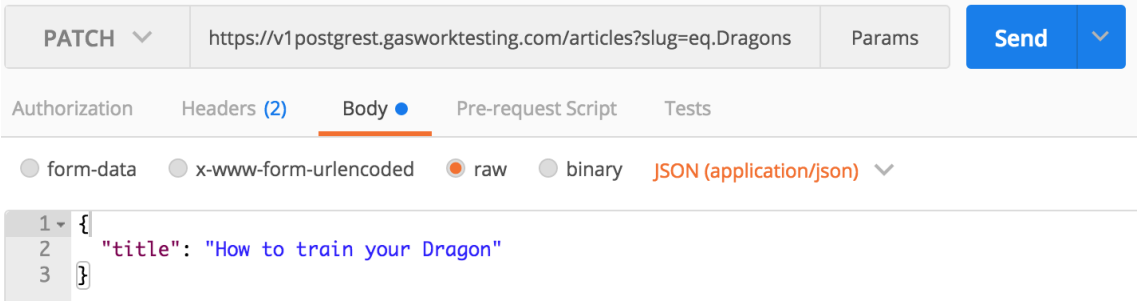

Figure 4.17:

We can change the verb to GET to see the effect of the previous query, see figure 4.18. Importantly, only this article will have been changed.

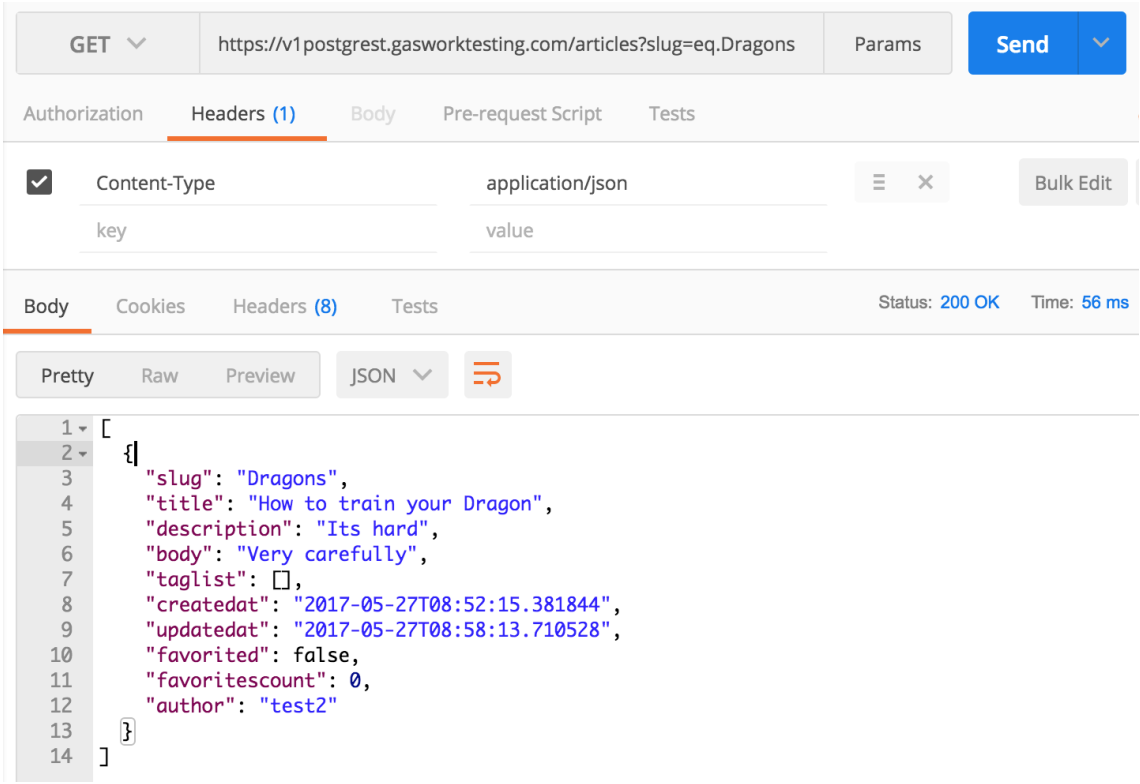

Figure 4.18:

What happens if we modify all articles? In the next PATCH I set all "descriptions" to "Dragons are hard to train", see figure 4.19. This query should set all articles to the same "description" since there is nothing to filter what articles should be modified. It would be equivalent to doing:

```
update articles set description = "Dragons are hard to train";
```

Figure 4.19:

However, when I request all the articles again, only the ones that "test2" owns were modified. This is because the row level security only provides access to the articles where the author name is equal to the username in the JWT. In reality the PostgreSQL database is running the query I wrote before, just in a different context.

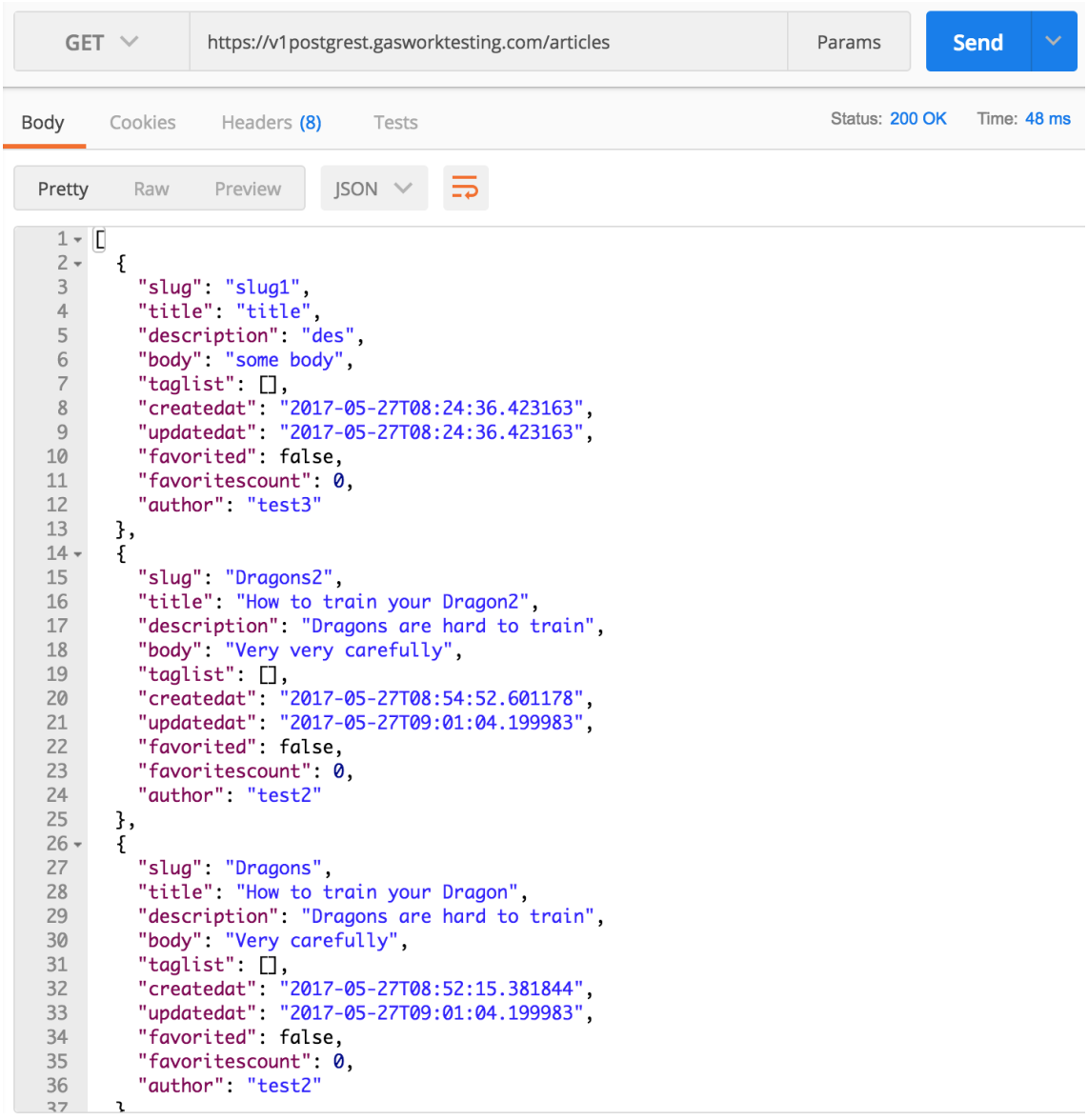

Figure 4.20:

In this example I showed how row level security is used to confine each user to only the row's

they should have access to. The effect is each user can pretend that they are the only ones on the system. Instead of having to try and just modify their own profile they can modify all profiles knowing on theirs will actually change. It is also possible to have the same affect as row level security by adding a where clause to the view on the table, but that method does not allow different row's to be returned when calling SELECT as apposed to calling UPDATE.

4.4.4 Data Integrity

Data integrity is partly the purpose of a PostgreSQL database and as such there are a wide variety of ways that PostgreSQL can be used to keep data from being entered incorrectly. One of the most useful aspects are the "references" keyword, "with check" and triggers. These three help either keep the user from entering invalid data or help keep a variety of tables in sync. Requiring the user of the Web API to remember they need to make multiple requests to keep different sets of data in sync is asking for trouble. Here I demonstrate how this can all be guaranteed by the PostgreSQL database.

I have kept the changes I made in the previous examples, see figure 4.21 for the profiles I added from before.

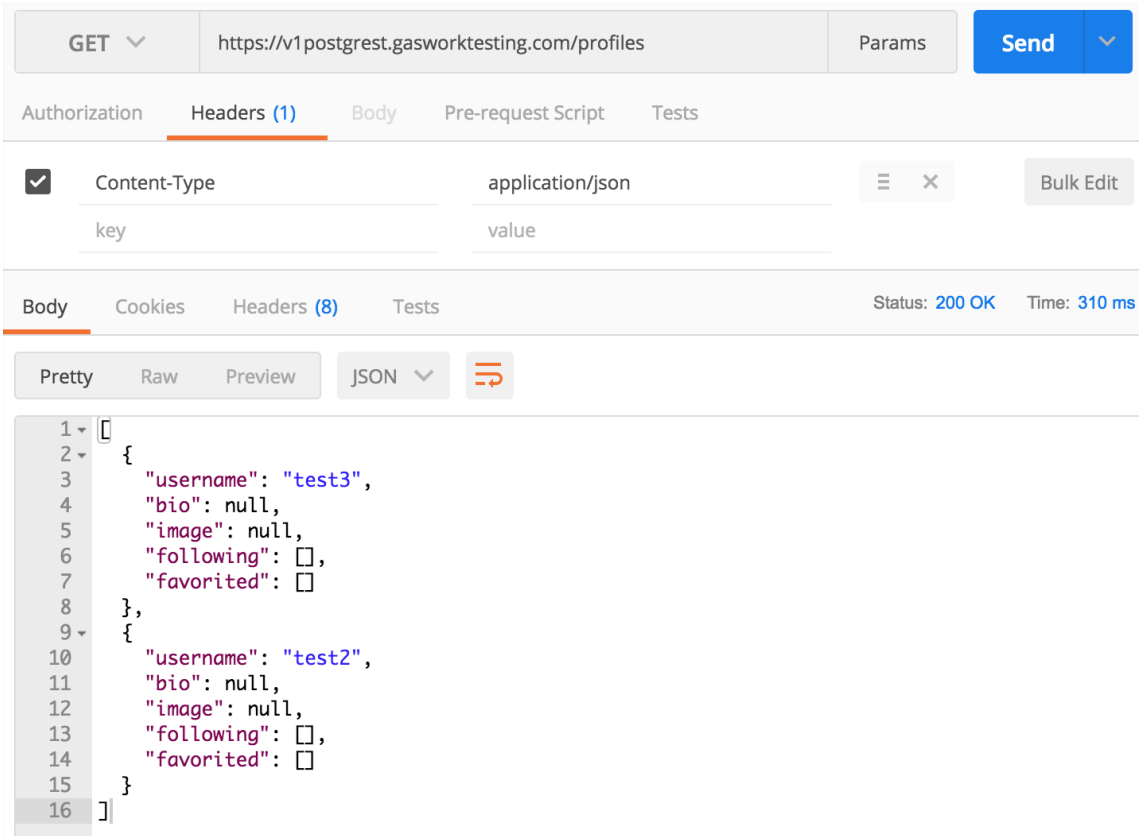

Figure 4.21:

Lets start by trying to follow myself, see figure 4.22. Immediately after I make the query, I get back the error that the `"profiles_check"` has been violated. What is happening is the "with check" clause on the column attribute for "following" is returning false. I wrote a function in SQL that checks each element in the following array before it can be written to the table. So if user tries to follow themselves, a non-existent user or tries to follow the same user multiple times, the "with check" clause will fail. Plus, the function will throw an error if the input is not an array, since the type is jsonb you could try to put any valid json object.

Unfortunately, I have not found a way to send better error messages. I know that the "hint" field is populated from raising an error with the hint filed set and the "description" field is populated from the errors message. However, that only works in places where I can raise an error. In the above function I could have it raise an error when the constraints fail, but that is not always the

case. On fields that have just `referencing`, there is no place for me to throw an error. PostgreSQL does support comments on everything, but PostgREST does not use them for feedback. This is something future versions can support.

Figure 4.22:

The way that following a user works in this version of the Web API is to send a PATCH to the user's profile. Notice again, I do not filter the profiles for my profile, instead relying on row level security. This request is authorized with the JWT I got in the previous example.

In figure 4.23 I set "test2" to follow "test3" which is valid because "test3" is another user. Its possible to add as many followers as there are users on the system.

A defect of this version is that all users can see who everyone else is following. Plus, each time a user wants to change who they are following they need to send a full list of everyone they were following, plus the new user they are following.

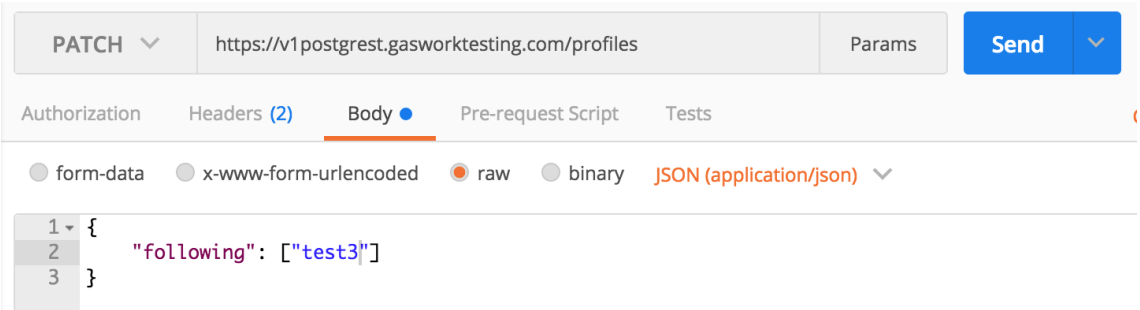

Figure 4.23:

In figure 4.24 you can see the change that "test2" is now following "test3".

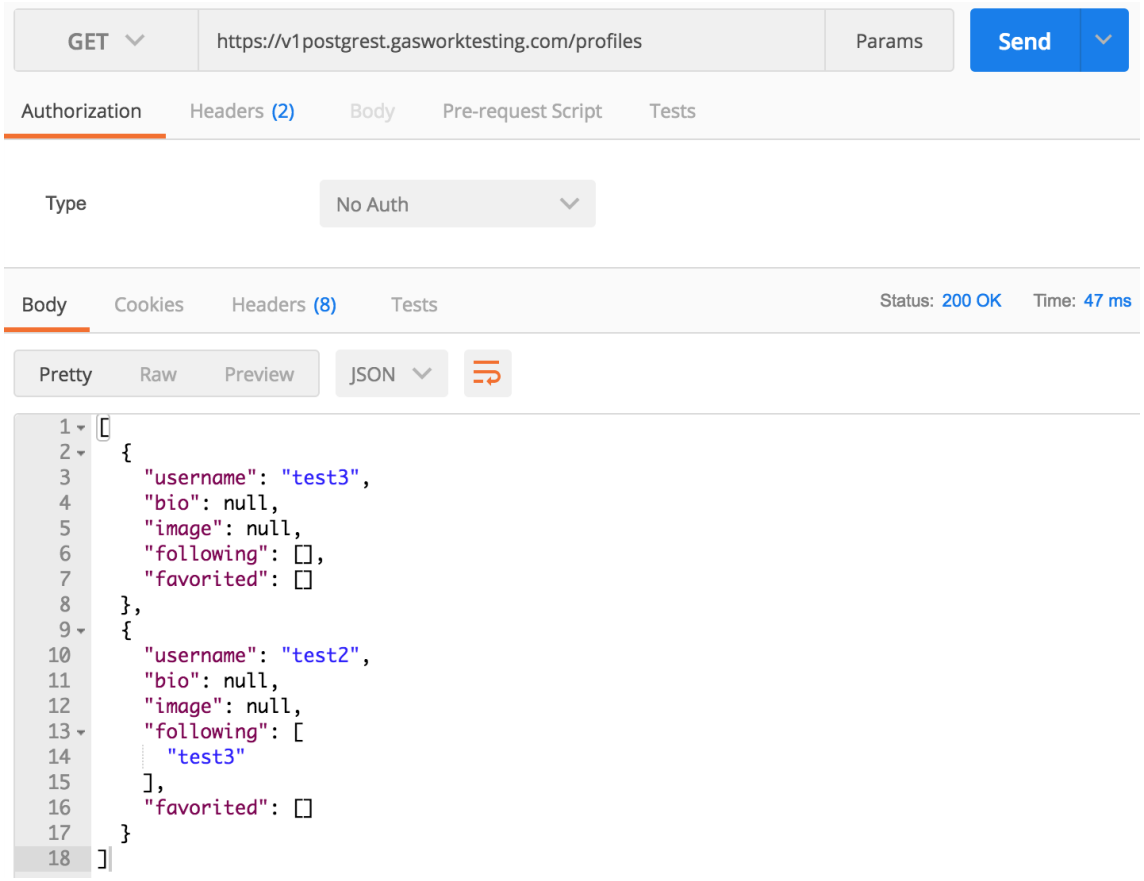

Figure 4.24:

In the previous commands, I showed how the proof of concept kept a single table consistent within itself, but this same method can be applied to more complex situations. In figure 4.25 I am showing you the previous articles that were posted by the past two examples. In those examples I did not explain how "favoritescount" is handled.

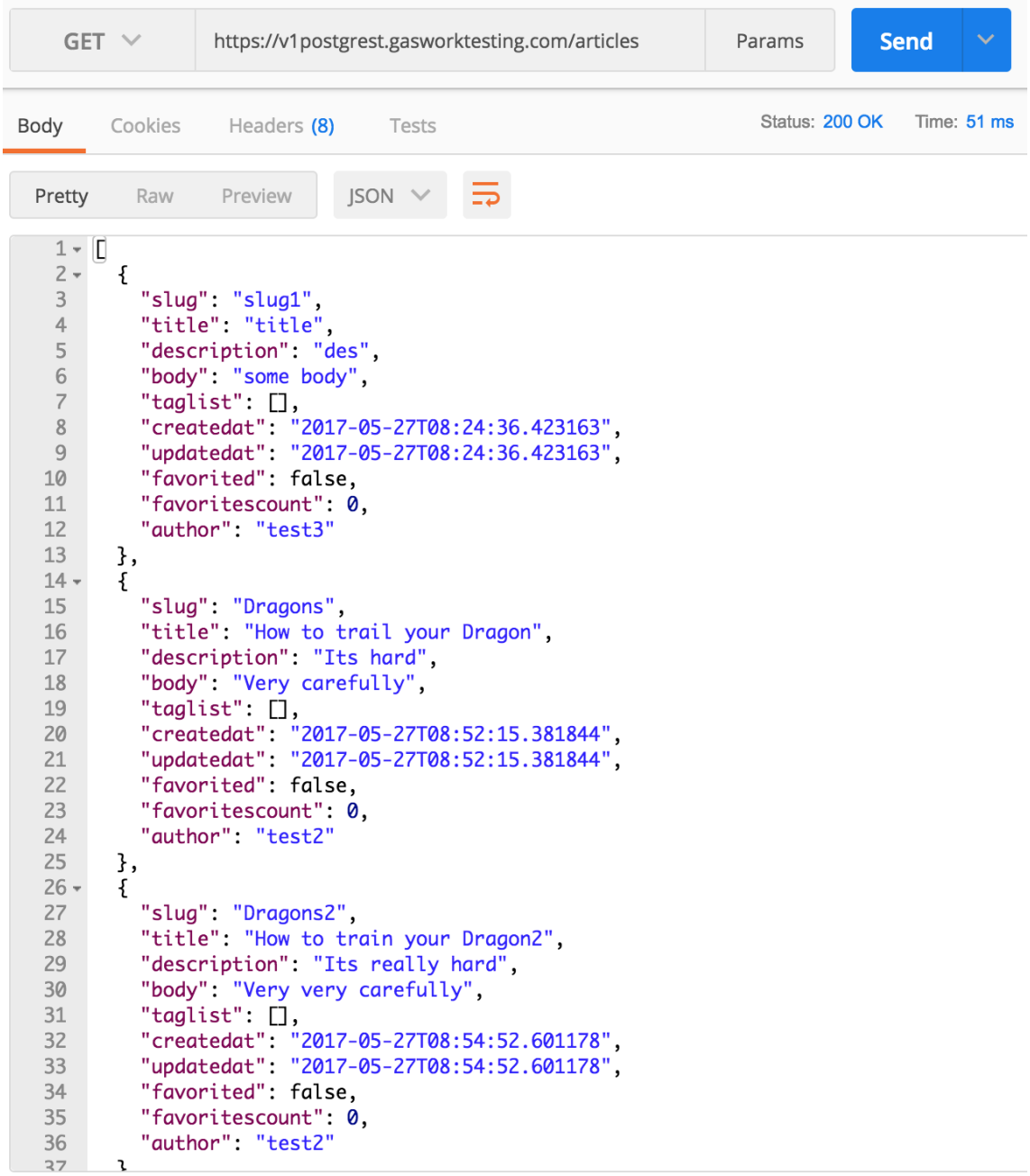

Figure 4.25:

First I need to favorite some of the articles, see figure 4.26. If I try to favorite the same article twice, though, I get a `"profiles_favorited_check"` failure, just like with following other users.

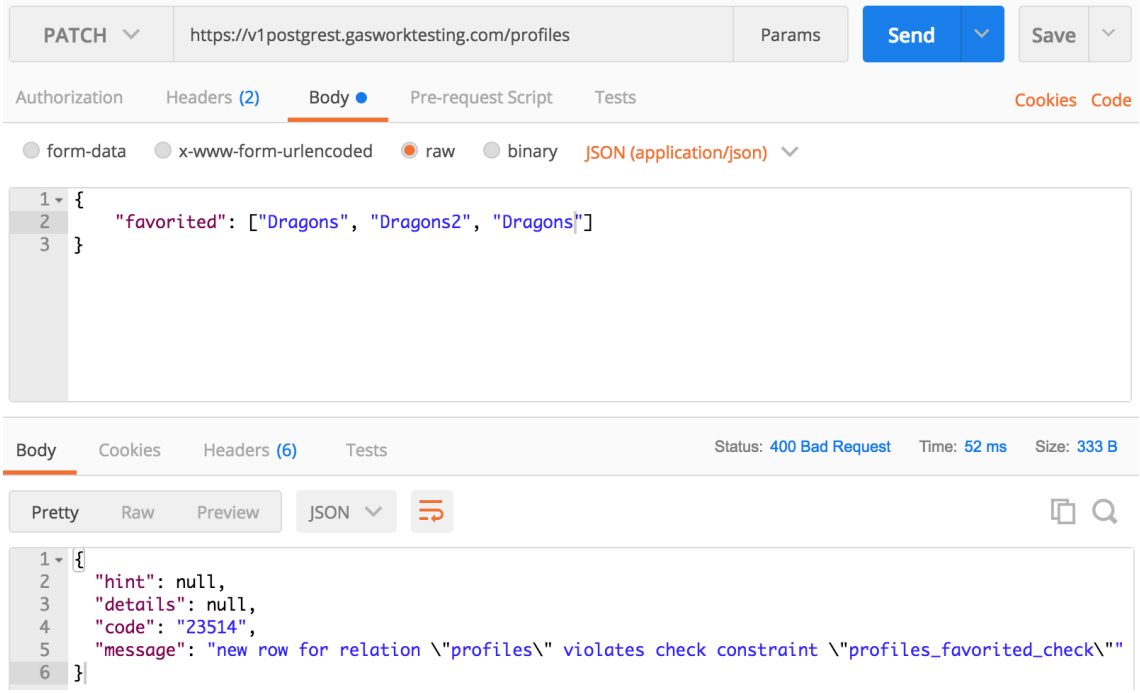

Figure 4.26:

If I try to favorite an article that does not exist, the same check fails, see figure 4.27. Unfortunately, both of these examples show the poor error messages. Data integrity is kept, but it's not clear why the user cannot favorite "Dragons" and "Dragns2". The fact that "Dragns2" is not found is not made clear in this version.

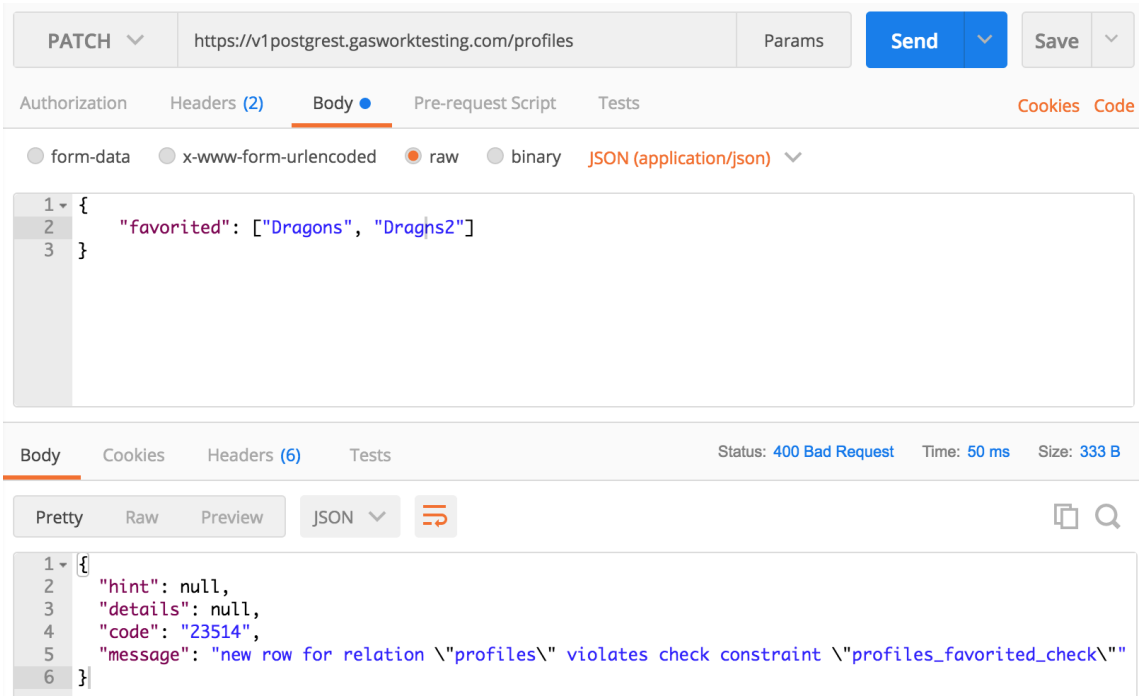

Figure 4.27:

Finally I submit a valid "favorited" array that completes without errors, see figure 4.28. It's worth mentioning that there is no feedback when an operation is successful, instead just a status code of 200 is returned. This is normal practice for a Web API.

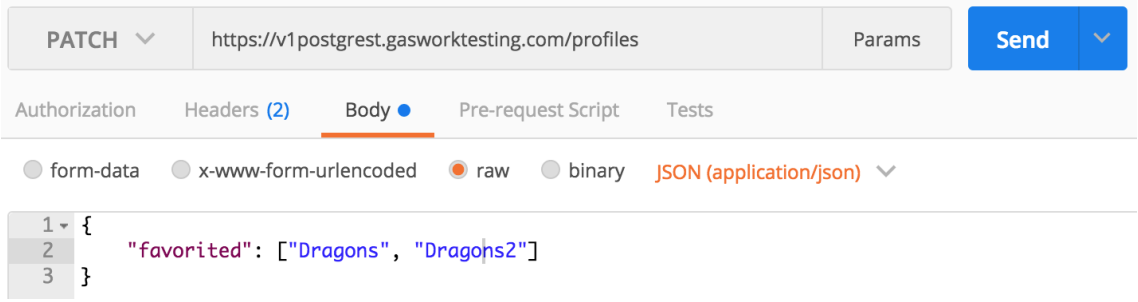

Figure 4.28:

In figure 4.29 I am just showing the result added to "test2"'s profile. Note, I am accessing the profile details without authentication, which is part of the specification I am implementing. However, not part of the spec is showing who is following who and who has favorited what users.

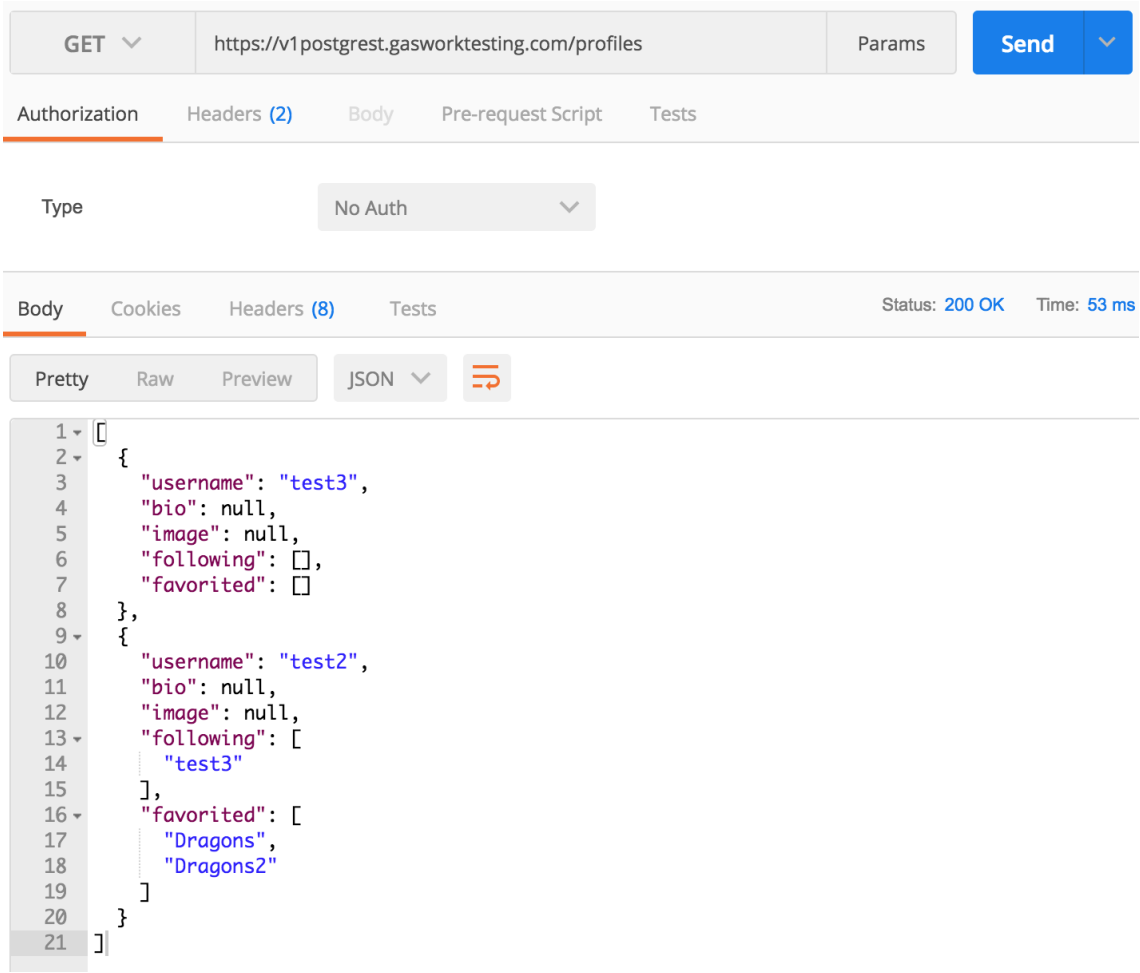

Figure 4.29:

Now in figure 4.30 you can see there is a change in both "Dragons" and "Dragons2"'s "favorite-count". However, the "favorited" flag is set to false. This is because I made this request without authentication.

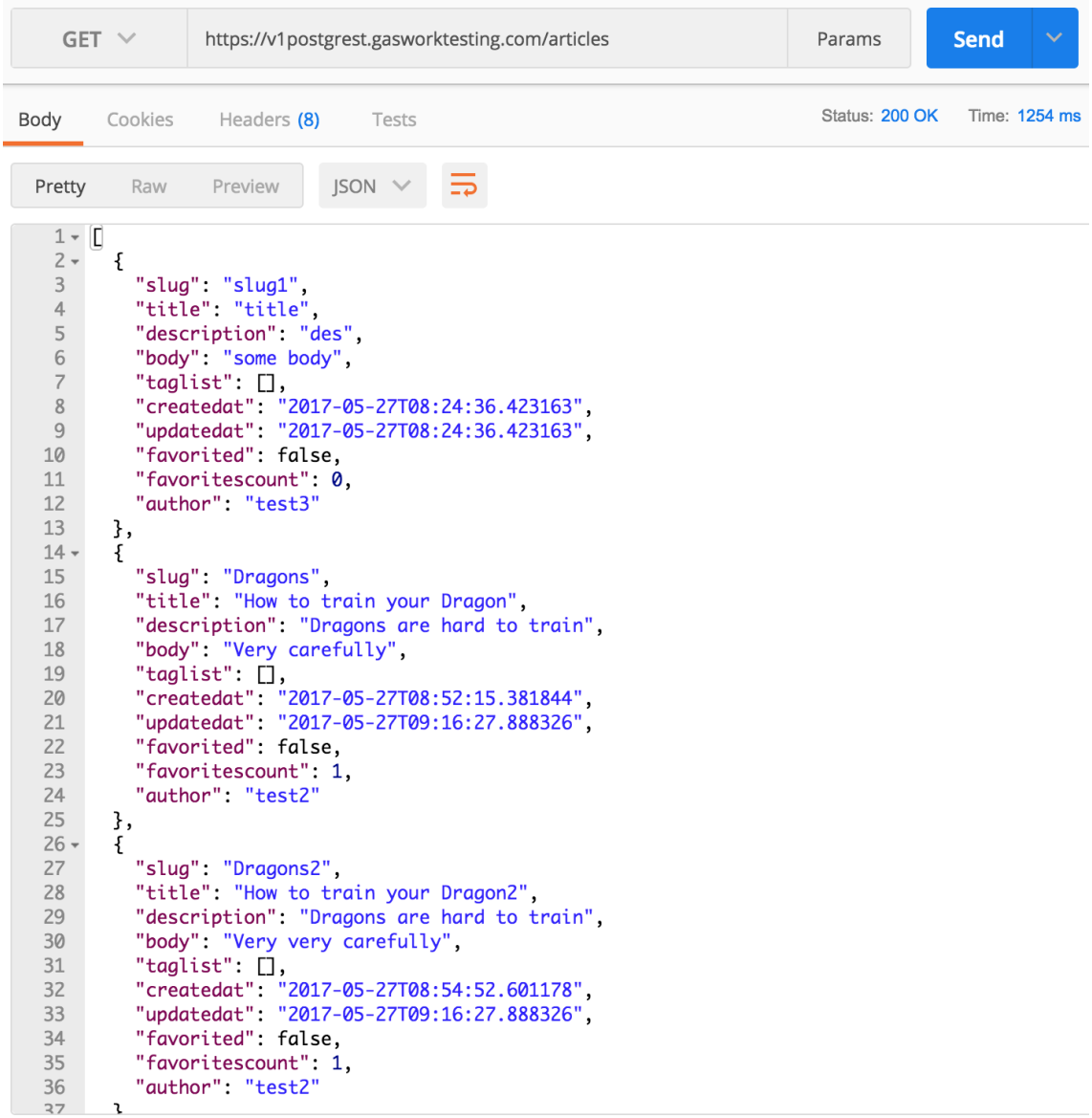

Figure 4.30:

If I make that same GET request as in figure 4.30 with my JWT then I get back that "test2" has favorited both "Dragons" and "Dragons2", see figure 4.31. This feature is implemented by having the article's views call a `isFavorited` function for each article returned in any query. That function uses the username in the provided JWT to lookup the user's profile and check if a given article's slug is in the profile's "favorited" array. This is where SQL makes life easy. All I needed to do was call the function in the article's view with `slug` as its argument. PostgreSQL knows that `slug` is a column in the article's table that is guaranteed to not be null. So I can be sure that this function will always be given a slug for each row that is returned and that PostgreSQL will call this on every article that is returned. Plus, since I verify all `favorited` entries I know the `isFavorited` function will not throw an error because it cannot get something other than an array with slugs of real articles. By programing all requirements into the database, I have less I need to worry about because I know the data is always valid.

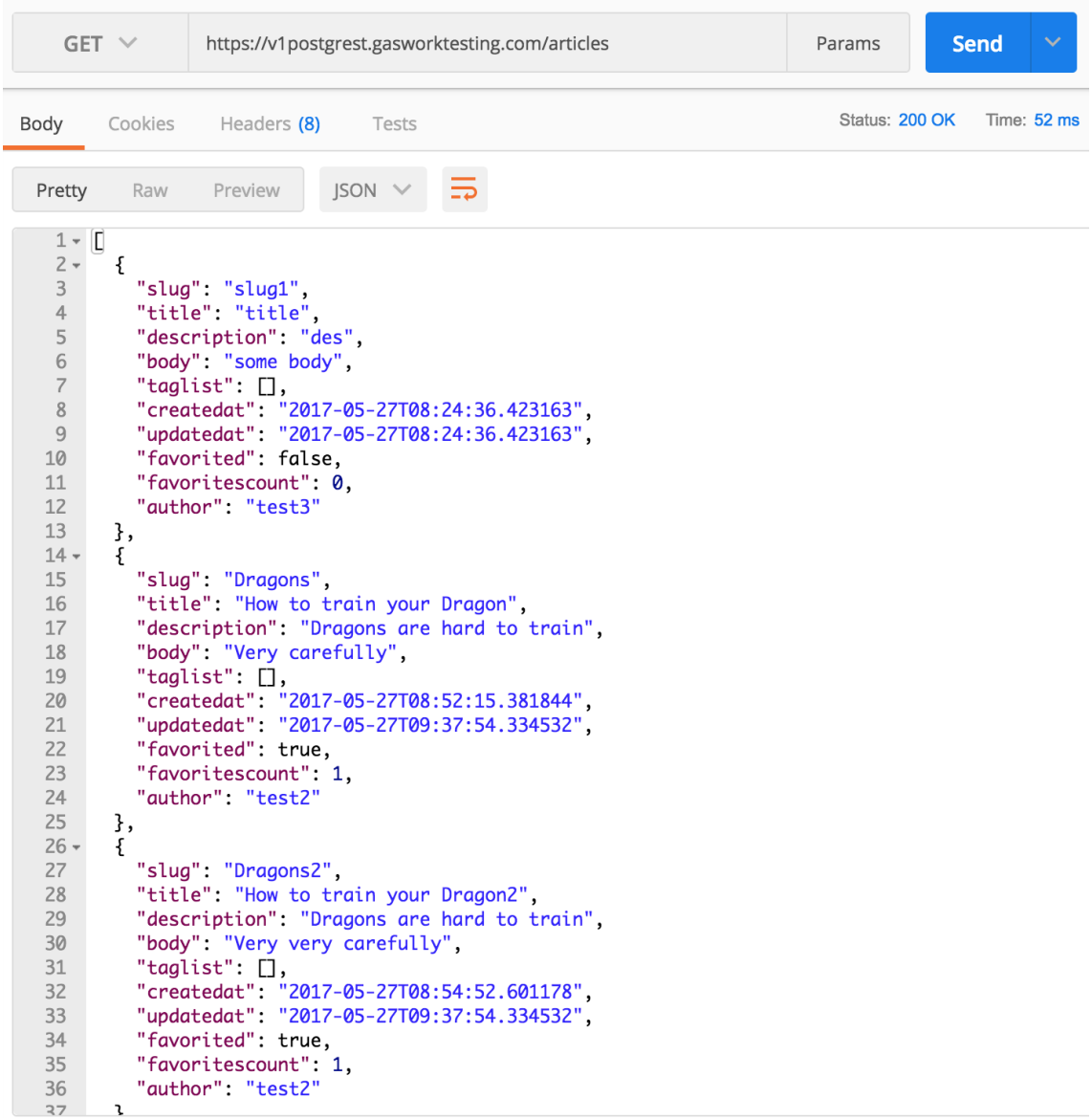

Figure 4.31:

Importantly, keeping the "favoritescount" in sync with the "favorited" in all Profiles is not just a mater of increasing the count. In figure 4.32 I remove "Dragons2" from the array by setting it to only contain "Dragons".

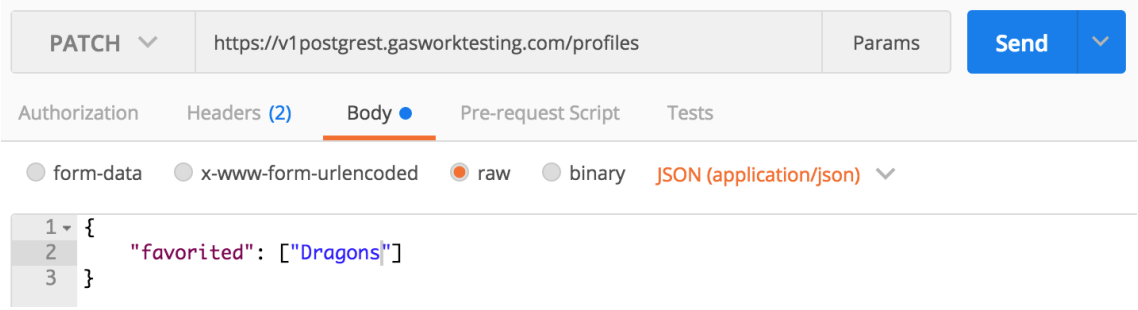

Figure 4.32:

Now you can see in figure 4.33 that the "favoritescount" of "Dragons2" has decreased automatically. All I did with these two requests was update an attribute in the Profiles table and GET all

articles. It was up to the database to keep the data in sync.

However, if you look closely you will notice that the "updatedat" field has changed each time I favorited a post. That filed should change whenever a post's content is changed, but not when just the "favoritescount" changes. This is a bug in this version of the Web API.

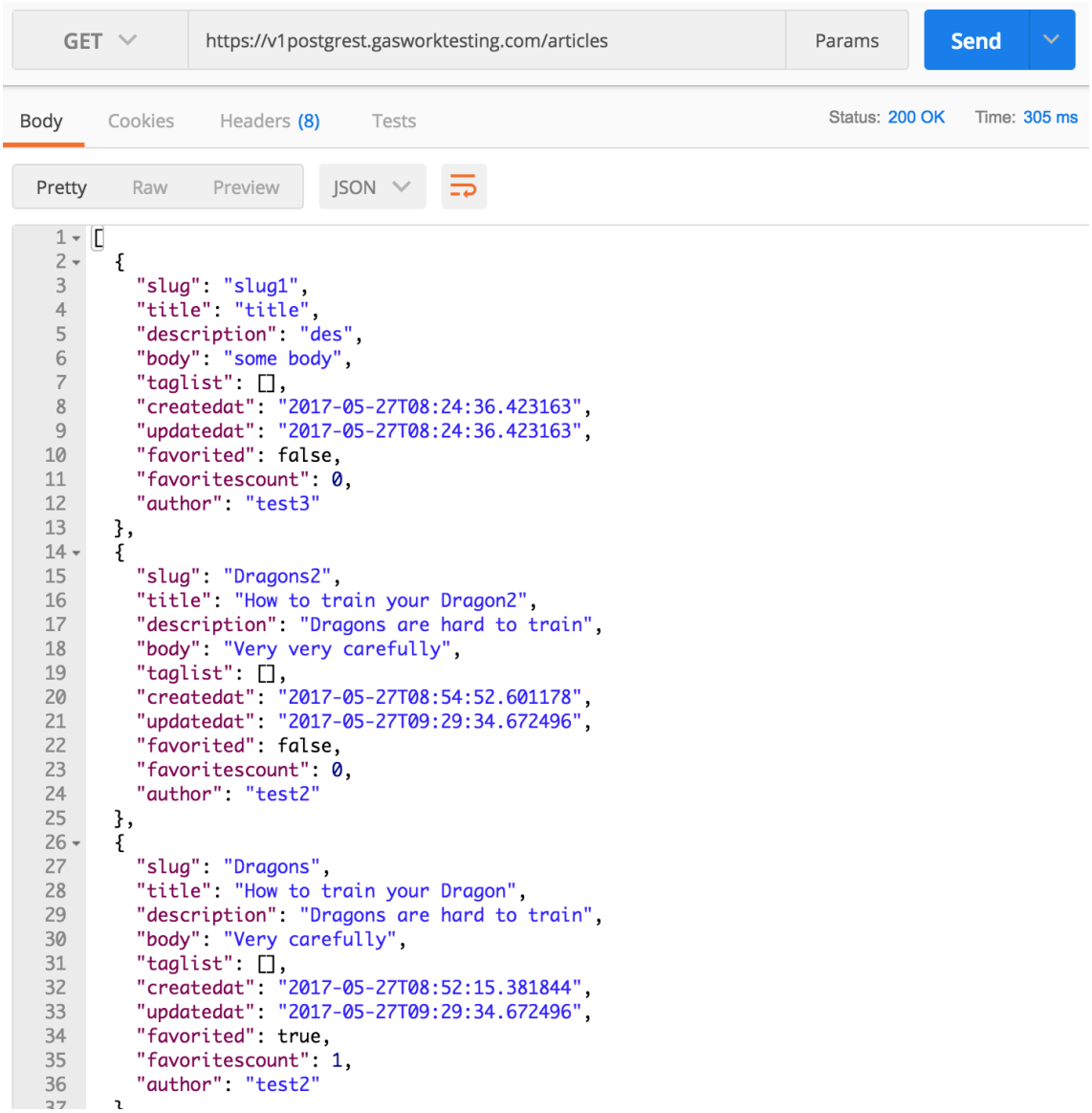

Figure 4.33:

This example illustrates how data integrity can be kept automatically by encoding how data should be kept in sync in the database itself. This prevents authorized access from incorrectly entering information or otherwise compromising other functions in the database. The `isFavorited` function is only guaranteed because of the other constraints in the database, instead of having to handle all cases on its own. Also, all of this logic is encoded at the table level, beyond the reach of any views, so a developer can safely add or modify any existing views without interfering with the integrity of the data. This will be shown to be very crucial in the next step where I show you the next version of this same Web API.

4.4.5 Multiple Versions

There were a number of problems with the first version of the Web API including: every user knew what ever other user had favorited and who every user was following, the error messages were quite cryptic at times and the "updateat" field changed when a post was favorited, not just when there

was a content change.

To make these changes I first duplicate the first version of the Web API's views into a new schema, named v2. Then create a new anonymous role, `anonv2` and new role for logged in users `v2_logged_in`. This whole time I have talked about `anonv1` as this anonymous user for version one of the Web API without explaining why it has a version attached. The reason is so that the Web API can have a separate anonymous user for each version, so there is complete separation between each version of the Web API. There is a slight disadvantage to doing it this way because a database administrator needs to go in and add all existing users to the new `v2_logged_in` group, but the clean separation between two versions is more important.

With this last example, the "createdat" and "updatedat" of the posts have changed since in between writing the previous examples and this one the whole system needed to be re-deployed for unrelated reasons. The same content as posted by the first version of the API has been replicated. To recap, the articles seen in figure 4.33 are same as in figure 4.34 below.

Figure 4.34:

The first addition to the new version is `my_profile` view as seen in figure 4.35. This view can be manipulated just like the profile view from the previous version, but it only provides access to

the user's profile. This means there are two views accessing the underlying profile table: One called `profile` that gives SELECT access to "username", "bio" and "image". The other `my_profile`, which is not accessible to `anonv2`, but gives SELECT to all columns and UPDATE access to all except the "username"; on the row that belongs to the user.

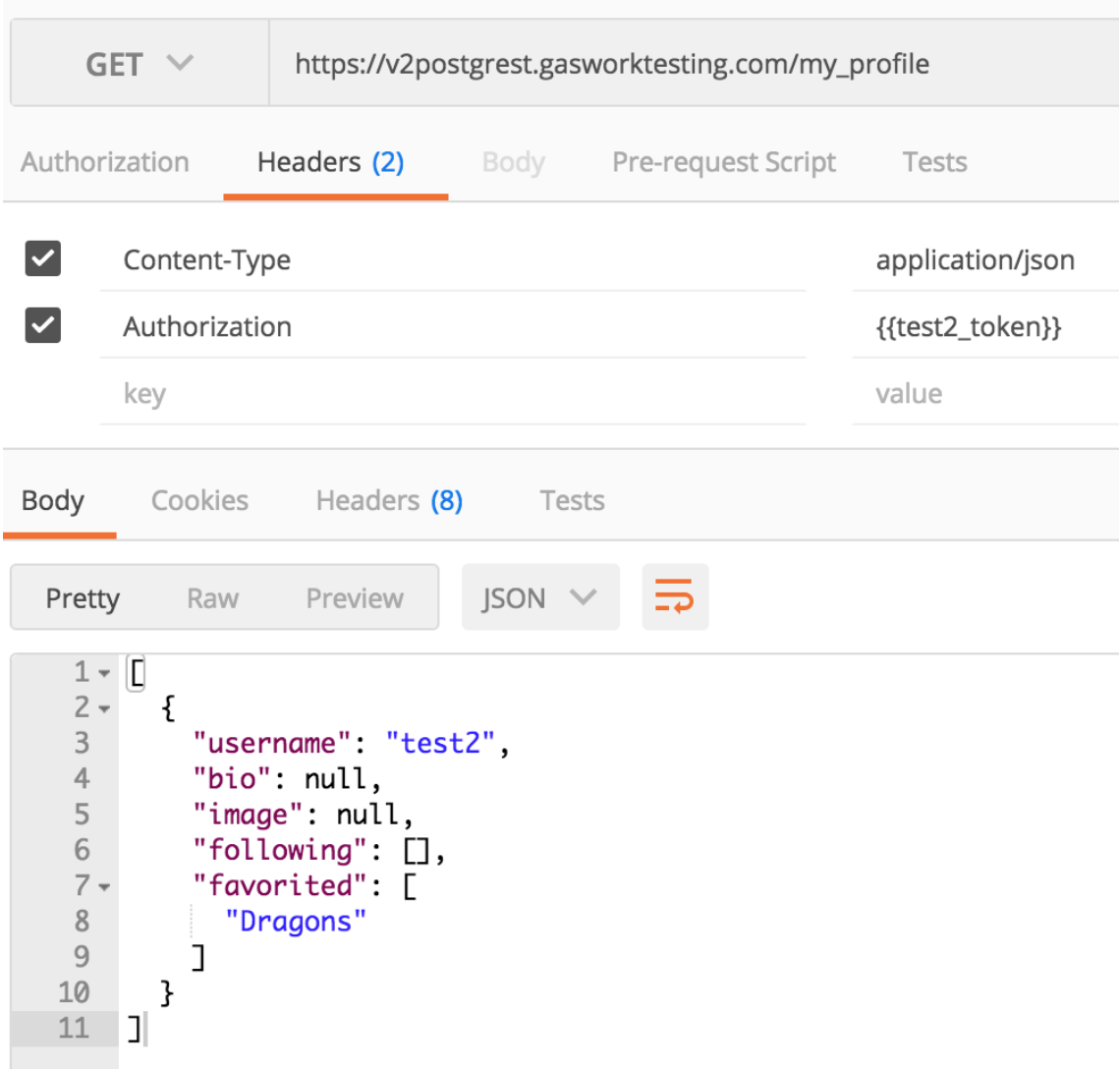

Figure 4.35:

To deal with following, unfollowing, favorite and unfavorite I added functions that add and remove an array of input. Plus I added better error messages for feedback, see figures 4.36, 4.37, 4.38, 4.39. Each of these is run in sequence to show both the different error messages and all the cases that are handled. Although not added to this report, the other three functions have the same cases they handle.

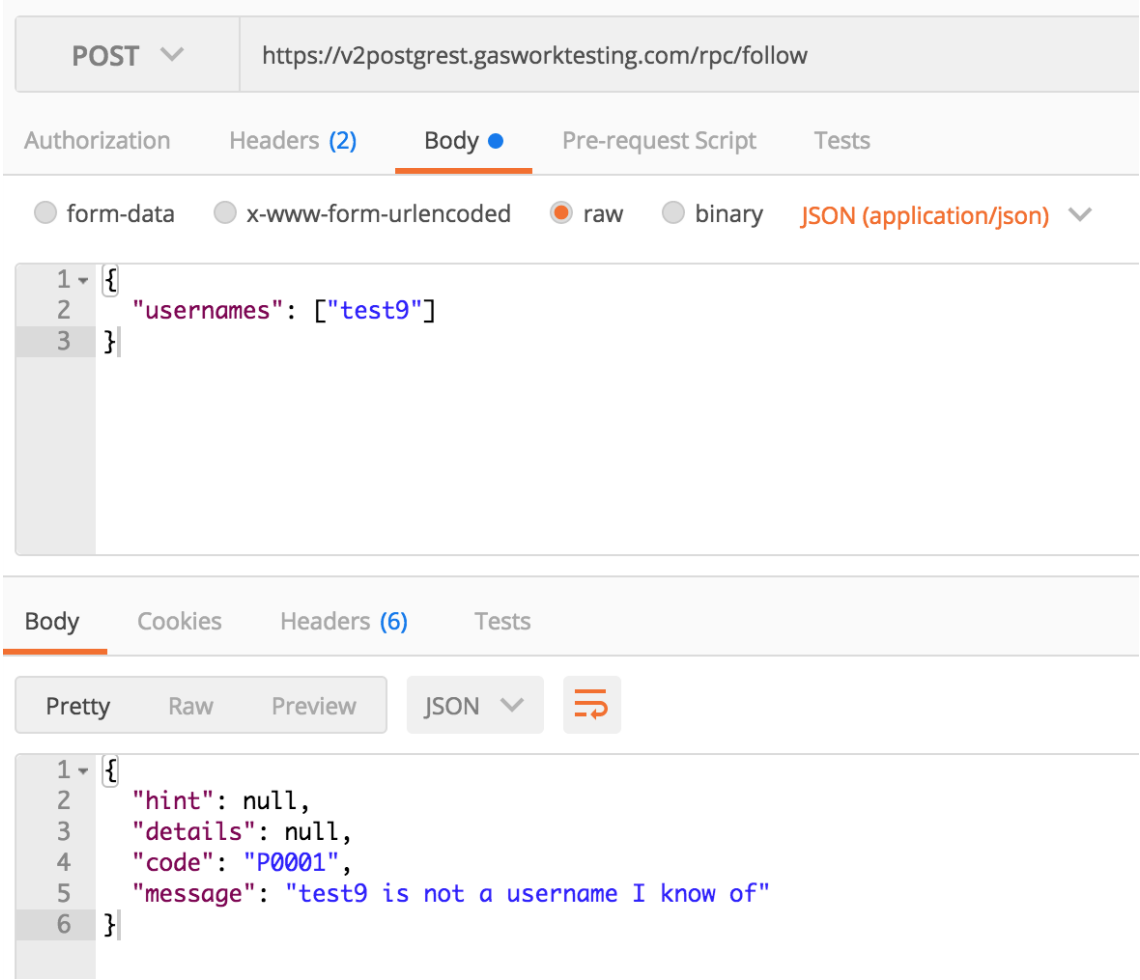
POST
https://v2postgrest.gasworktesting.com/rpc/follow
Authorization
Headers (2)
Body
Pre-request Script
Tests
form-data
x-www-form-urlencoded
raw
binary
JSON (application/json)
{
"usernames": ["test9"]
}
Body
Cookies
Headers (6)
Tests
Pretty
Raw
Preview
JSON
{
"hint": null,
"details": null,
"code": "P0001",
"message": "test9 is not a username I know of"
}

Figure 4.36:

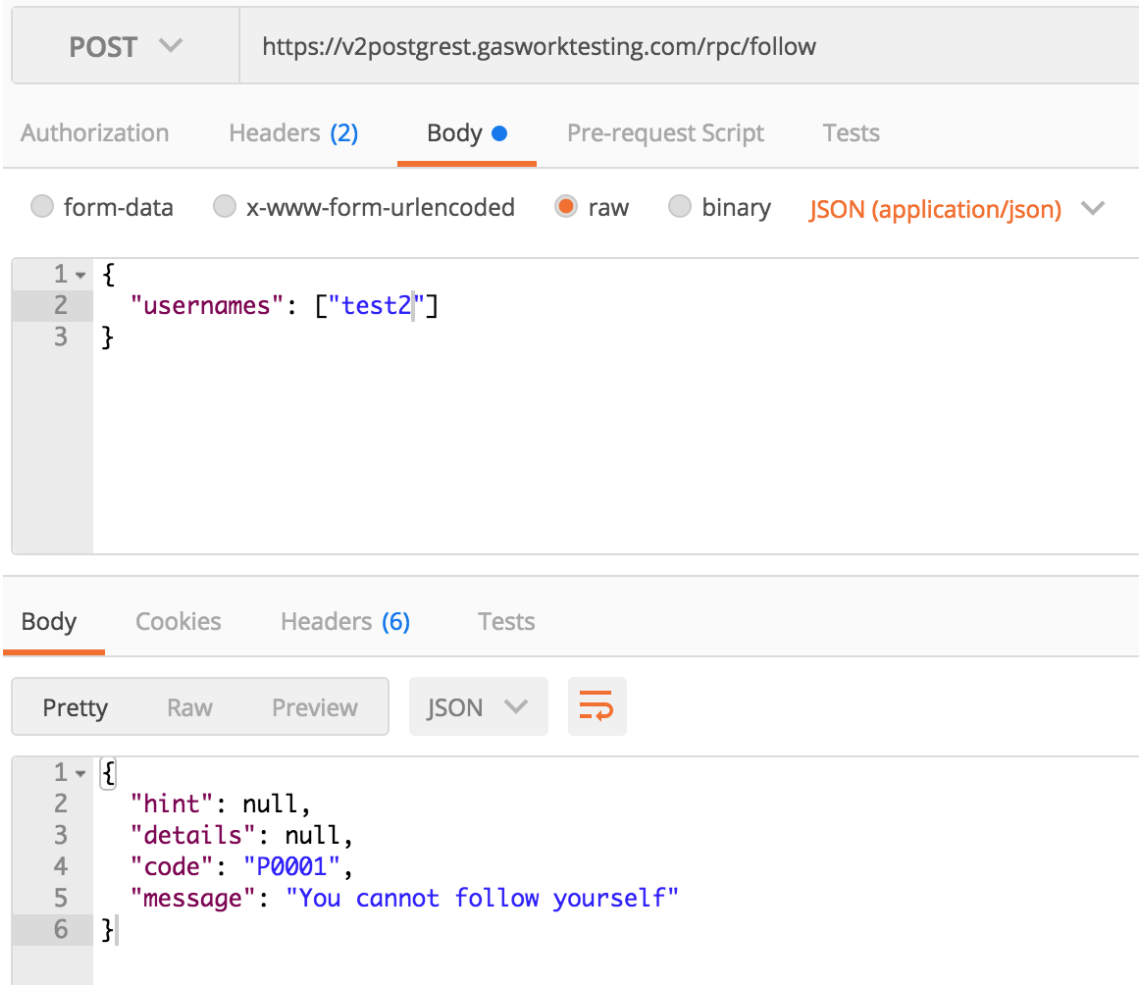

Figure 4.37:

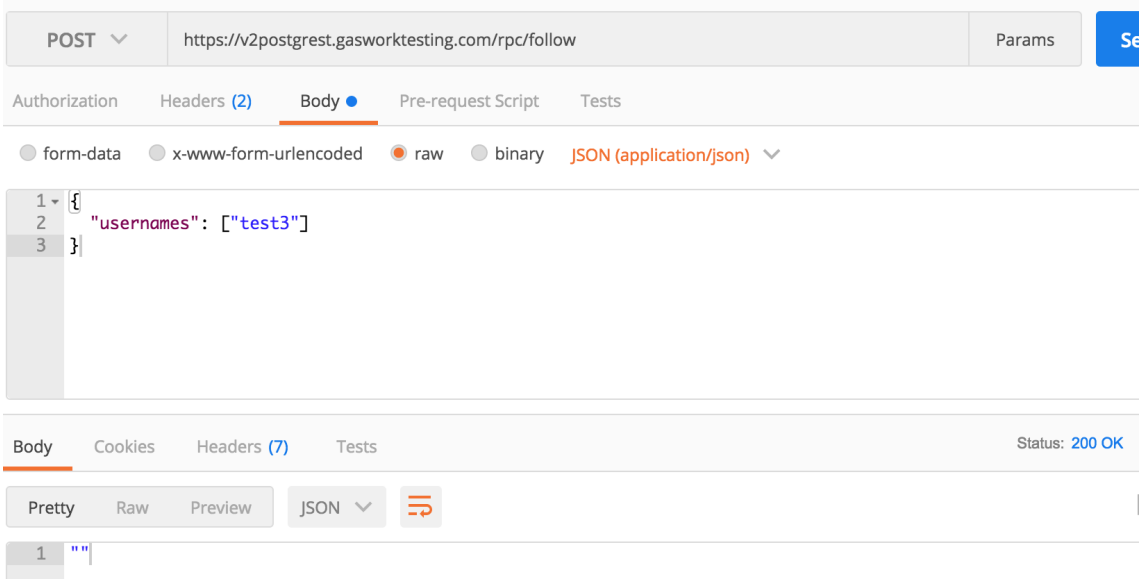

Figure 4.38:

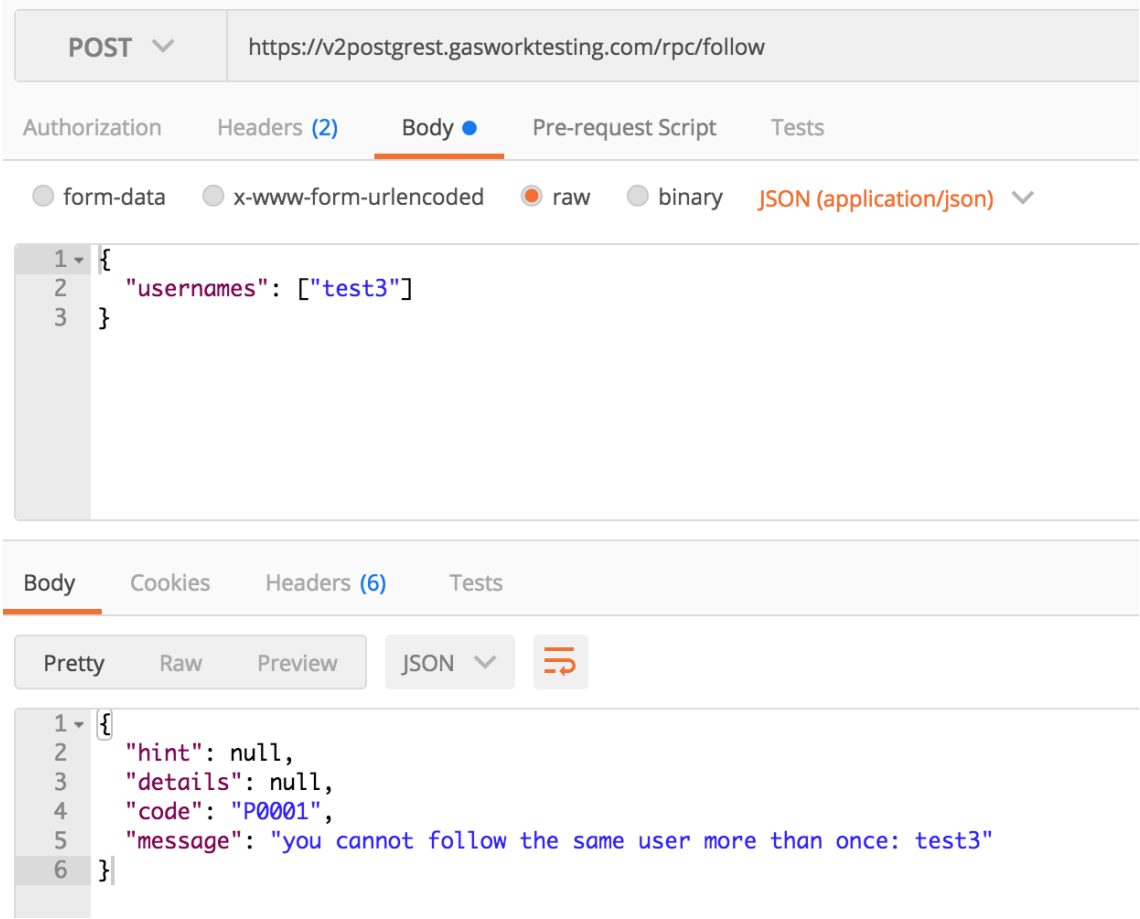

Figure 4.39:

To show both the changes to `profiles` and the GraphQL aspect to the Web API I select all articles as well as the profiles of the author of those articles, see figure 4.40. The syntax after the "?" says, "select=*", which means get all columns from `articles` then the "," specifies a traversal of any "references" in the table followed by what is selected from that table. In this case `profiles{*}` is selecting all columns from the `profiles` table. The result is effectively an SQL join that returns the relevant row as a key within the original object as seen in figure 4.40. However, this traversal is not accessing the `profiles` table directly, rather it is selecting from the `profiles` view. This is despite the fact that the "reference" is an attribute of the underlying table. This is very important to give the correct behavior, since one of the issues in the previous version was exposing too much information from the `profiles` table. There is also the fact that PostgREST does not have access to the `profiles` table in the first place.

There was also some minor tweaks to the names like "createdat", which became `created_at` and so on. Originally I did not realize that PostgREST removed capitalization from names.

GET https://v2postgrest.gasworktesting.com/articles?select=*,profiles{*}

Body Cookies Headers (8) Tests

Pretty Raw Preview JSON


```
[
  {
    "slug": "Dragons2",
    "title": "How to train your Dragon2",
    "description": "Dragons are hard to train",
    "body": "Very very carefully",
    "tag_list": [],
    "created_at": "2017-05-31T06:07:18.09583",
    "updated_at": "2017-05-31T06:07:18.09583",
    "favorited": false,
    "favorites_count": 0,
    "author": "test2",
    "profiles": {
      "username": "test2",
      "bio": null,
      "image": null,
      "following": false
    }
  },
  {
    "slug": "slug1",
    "title": "title",
    "description": "des",
    "body": "some body",
    "tag_list": [],
    "created_at": "2017-05-31T06:08:10.730187",
    "updated_at": "2017-05-31T06:08:10.730187",
    "favorited": false,
    "favorites_count": 0,
    "author": "test3",
    "profiles": {
      "username": "test3",
      "bio": null,
      "image": null,
      "following": true
    }
  },
  {
    "slug": "Dragons",
    "title": "How to train your Dragon",
```


Figure 4.40:

Another missing column was the article Slug in the `comments` table, see figure 4.41 to see the addition. However, this change can break the previous version. If the new column is defined as not null without a default, then the first version of the `comments` view will not be able to insert into the table. There are two solutions, either patch the old view by dropping it and re-creating it or

make the added column as "nullable". There cannot be a default article that comments made on, so "nullable" is the only option that still meets the "references" constraint. In this case `comments` without an article can never be displayed or even make sense for the front-end to create, so it is better to change the previous version's view. This illustrates that, although keeping previous versions as immutable is very useful, if some of the functionality is completely broken it does make sense to fix them. The purpose of being backwards compatible is so that the developer is not forced to make a choice between making a better interface that will break everyone else's code and never modifying anything.

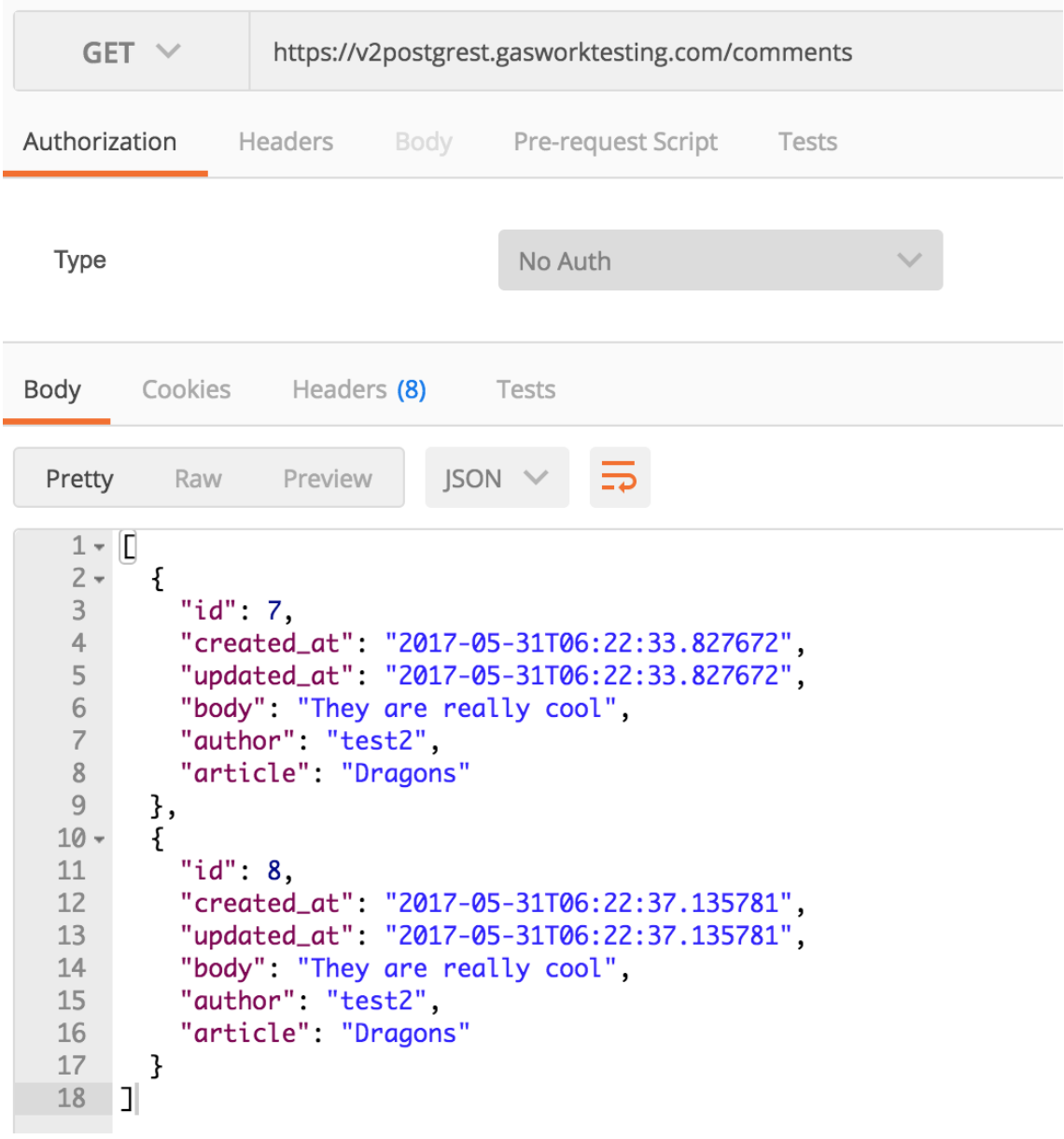

Figure 4.41:

The last feature is tags which is another view built on top of the `articles` table. In figures 4.42 and 4.43 I am adding tags to the two existing articles.

Figure 4.42:

Figure 4.43:

Finally in figure 4.44, without authentication I am making a GET request to the tags view which grabs all tags from each article and puts them in a single array for me.

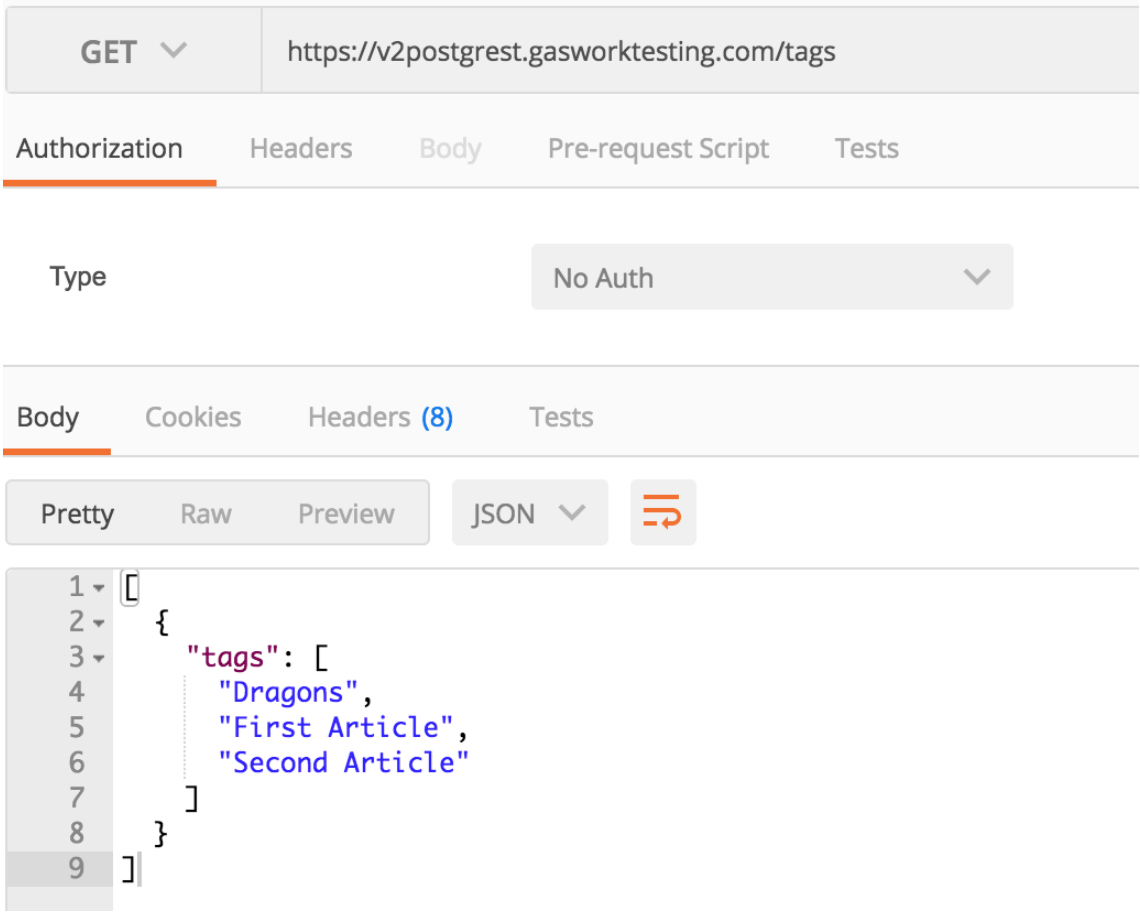

Figure 4.44:

This example shows how to make numerous changes to an existing Web API while still running off the same data as a previous version of the same Web API. In this case I had to change part of the previous version because the `comments` table was missing a critical feature, but no other change required any modification of the previous Web API. Importantly, that change should not have affected anything working off of the previous version since the `comments` view was effectively useless without being related to an article.

Chapter 5

Evaluation of Proof of Concept

5.1 Interface

The generic Web API provides rich query syntax with fine grained filtering that puts bespoke Web APIs to shame. The proof of concept is not only as expressive as the specification 5.3, but also allows for far more complex quires that the specification. Although, PostgREST itself has some minor limitations, the underlying idea of generating a Web API from the data in a PostgreSQL database is only limited by what SQL itself is able to query. Plus the entire interface comes for free; there is no need to write any bespoke code. All things considered, one of the best reasons to use a generic Web API is because of the expressive RESTful interface.

5.2 Data Security

The PostgREST instance really is just a glorified way of translating a URL into a SQL query. It does validate that the given URL and body are syntacticly valid, plus checks the JWT for validity, see section 4.3. However, beyond that it is entirely up to the PostgreSQL instance to decide on what to do with the query. This means attacks like SQL injection are handled very differently than in traditional systems, see section 4.3. However, the resulting Web API is far more powerful than any traditional bespoke Web API with much less effort. In the database, by just defining a single view the PostgREST instance provides access to, select, insert, update and delete with similar powers of filtering as SQL itself. If PostgREST supported GraphQL in its entirely, then the Web API would be equality as powerful as running SQL directly in the PostgreSQL database. So expressibility in this system is not about what can be expressed in the URI, but how to encode requirements of the data so that this vast variety of queries can be constrained to the problem space.

Since SQL is Turing complete, as implemented is PostgreSQL, any constraint can be expressed. However, using SQL in practice can be painful to express some kinds of constraints. SQL is good at expressing constraints on how data should be stored and the relationships between tables but when it comes to expressing who should have access to what in fine grained detail, minor changes can have widespread, undesired effects. In essence SQL was designed for querying data and validating data before storing it, while some types restricting access came later. This is evident when critical security features such as `security_barrier` are being added in PostgreSQL at version 9.2 while the current version is 9.5.

For example, I can express all articles must have an author with a profile by adding "text not null references profiles(username)", where profiles and articles are separate tables. I can even extend this requirement to say that all profiles must have user credentials by adding "text primary key references users(username)" to the profiles table. The effect is, I am guaranteed that you cannot create an article that does not have an author with both a profile and user credentials. At first having seemly complex requirements are easy enough to add. However, if I want to restrict updates to the article to only the author, there needs to be role for each author, that role must be added to a role with the correct privileges, there needs to be a way for each user to authenticate, a way to validate claims for each request and either row level security or restricted views to the underlying tables. Naturally this process is painful and error prone. If any of the steps above have an error, the user could access data they should not be able to.

To some degree the above security problem is not just a problem with SQL, but of authentication in general. That topic in and of itself is beyond the scope of this project. Despite the difficulty, as shown in the proof of concept, it is feasible to handle all security requirements with just SQL in a PostgreSQL database. As such, the problem is not expressibility, but usability meaning a plausible solution to this problem is to build a system built on top of PostgreSQL that generates SQL that is guaranteed to handle both data integrity and data security according to a specification.

5.3 Comparison to the RealWorld API Spec

This section goes through a direct comparison to the RealWorld API Spec in the Appendix. I am only comparing the second version I created with their spec. Also, for the most part, none of the differences between the Specification and implementation result in a loss of features or information. The exceptions are the bug with queries within JSON objects and the missing "Feed Articles". Also, the naming of all keys use underscores instead of the Spec's camel case.

1. Authentication Header
   (a) They use "Token jwt.token.here" I use "Bearer jwt.token.here"
2. Objects
   (a) Users (for authentication)
      i. I return only the token:

```
[ {"token" : "jwt.token.here"} ]
```

      The other information they send back can be gotten from my_profile
   (b) Profile
      i. Our Profile objects are identical, just PostgREST returns the object inside of an array.
   (c) Single Article
      i. PostgREST wraps the response in an array, even if there is just one result.
   (d) Multiple Articles
      i. The "articles" key is removed, instead returning the array as the top level object. Plus the "articlesCount" key is not present, instead PostgREST puts that in the header with the key, "Content-Range".
   (e) Single Comment
      i. Instead of "author" the key is "profile".
   (f) Multiple Comments
      i. Same differences as the Single Comment, but also does not have the "comments" key, instead the results are just in an array.
   (g) List of Tags
      i. Identical, just also wraps the whole result in an array. This is an issue with PostgREST, although minor.
   (h) Errors and Status Codes
      i. This is handled quite differently, if the request makes it to the database then this is returned:

```
{
"hint": "some hint",
   "details": "details",
   "code": "P0001",
   "message": "message"
}
```

      Otherwise only the "message" key is filled in if PostgREST rejects the request.

3. Other Status Codes

   (a) 401 and 404 are the same as the spec, but 403 never happens. The issue is, when a user tries to modify something they don't have access to the database runs their update as normal, just with no content. So as far as PostgREST knows, everything went well since PostgreSQL does not report an error. This is harder to fix and depending on the use case may be desirable. The fact that the database returns nothing, means a user cannot probe for objects to find out what exist.

4. Endpoints

   (a) Authentication

      i. The route becomes:

         `/rpc/login`

      and the "user" key is not there, instead only a single object with the keys "email" and "password" are provided. Plus, only the jwt is returned.

   (b) Registration

      i. the route becomes:

         `/rpc/new_user`

      and the "user" key is not there as well.

   (c) Get Current User

      i. the route becomes:

         `/my_profile`

      returns a Users object without the jwt.

   (d) Update User

      i. the route becomes:

         `/my_profile`

      The "user" key is removed and PostgREST allows partial updates.

   (e) Get Profile

      i. the route becomes:

         `/my_profile`

      otherwise identical.

   (f) Follow user

      i. the route becomes:

         `/rpc/follow`

      The user to follow goes into the body as an object after the key "usernames" as an array, which means I can follow multiple users with one request.

   (g) Unfollow user

      i. the route becomes:

         `/rpc/unfollow`

      The user to follow goes into the body as an object after the key "usernames" as an array, which means I can unfollow multiple users with one request. Uses the HTTP verb POST.

   (h) List Articles

      i. the route becomes:

```
/articles?select=*,profiles{*}
```

A. Filter by tag becomes:

```
/articles?tag_list=@>.{AngularJS}&select=*,profiles{*}
```

Unfortunately, this returns an error that has something to do with how PostgREST structures the json. Consider this a bug with the implementation, rather than an actual lack of expressibility.

B. Filter by author becomes:

```
/articles?author=eq.jake&select=*,profiles{*}
```

C. Filter by user becomes:

```
/articles?favorited=eq.true&select=*,profiles{*}
```

D. Limit number and offset/skip of articles can be done the same way as in the Spec or can be in the header

(i) Feed Articles

i. This did not get implemented, not hard to add, just forgot to add it.

(j) Get Article

i. This route becomes:

```
/articles?slug=eq.dragon&select=*,profiles{*}
```

(k) Create, Update and delete Article

i. The body for create and Update does not have the top level "article" key and the route name becomes:

```
/articles
```

Otherwise identical for all three routes.

(l) Create, Update and delete comments

i. The body for create and Update does not have the top level "comment" key and the route name becomes:

```
/comments?select=*,profiles{*}
```

Otherwise identical for all three routes.

(m) Get comments from an article

i. The route just becomes:

```
/comments?article=eq.Dragons
```

Otherwise identical.

(n) Delete Comment

i. The route just becomes:

```
/comments
```

Otherwise identical.

(o) Favorite Article

i. The route just becomes:

```
/rpc/favorite
```

The slug goes into the body of the request in an array under the key "slugs", this means I can favorite multiple articles at the same time.

(p) Unfavorite Article

i. The route just becomes:

```
/rpc/unfavorite
```

The slug goes into the body of the request in an array under the key "slugs", this means I can delete multiple favorites at the same time. Plus the HTTP verb is POST.

(q) Get Tags

i. The route just becomes:

```
/tags
```

Otherwise identical.

5.4 Summary

Overall, the proof of concept was able to satisfy the requirements of a non-trivial Web API by only defining the relationships between the data and attributes of the data. The result is secure, although there is no formal verification, and is designed to easy support multiple versions working off of the same data at the same time. The result is a more flexible interface and internal system than in traditional bespoke approaches. Not to mention the fact that this whole system has been tested by deploying it in a production environment.

Chapter 6

Reflection

I started this project with little understanding of what research was and to some degree designed my question from the answer I already "knew". At the time I was frustrated with the fact that building an Web API involved so many components and had too many places where the whole system breaks. This is something I found time and again across multiple blogs and personal accounts on how broken traditional web development is [Yor17]. Not to mention my own experience playing around in this space. I had this nagging feeling that there had to be a better way. At first I "knew" the answer was some way of propagating the types from the back end to everything else, just like Haskell does for each function within an entire project. For example, the Persistent library lets a developer design the database structure in a QuasiQuoter, which is just a fancy way of saying an embedded DSL, [Sno17]. That DSL handles both the database migrations and generates Haskell types so the developer can use the type system to guarantee their program is consistent.

I "knew" the solution must be to extend this methodology and it turns out there is an attempt at this with Swagger. The idea is your Web API generates a Swagger Specification then that is used by the Swagger UI for documentation and the Swagger Codegen to generate parts of your front-end code. Nice idea, but sloppy in practice. In part this is because the Swagger Specification itself does not allow as many constraints as you can express in the Haskell type system and the Swagger Codegen itself does not always respect the constraints it does.

After dabbling with Swagger in practice and starting my adventure into k8 and the ecosystem around it, I was starting to realize that the idea of a compiler or tool that would check a system for consistency, although useful, did not exist and would be very hard to build. Sure you could create a model of your system and check the model for consistency, but how do you know that the model matches what is implemented? In the end it's really the need for a specification that can be automatically turned into the implementation. Naturally this led me to executable UML.

I think this paper said it best [Rum14], the idea might be good, but the tooling is not there yet. However, as a result I looked into the idea of generating your Web API and found PostgREST which seemed like the light I had been looking for. By this point I realized my quest was more about finding an answer to my question than finding evidence for the answer I thought was right. The problem was: what exactly was my question? To some degree I wanted to know if I could propagate the types from my back end to my front ends, but that really is just an implementation for how to keep your system consistent, or as Haskell programmers put it "at least make sense". At the same time, PostgREST is trying to solve the consistency problem by removing a component, in effect the Web API and PostgreSQL database become one. So my question really becomes: "does merging you database layer and Web API layer result in a secure, expressive and scalable alternative to traditional approaches? At the same time does the whole system end up with a stronger guarantee of consistency?"

The idea that your data could give rise to your Web API makes a lot of sense, since in general the way you store your data is how it should be exposed. Fortunately for me PostgREST handles this part reasonably well, but an aspect that is not inherent of the data is the security of that data. What soon became apparent, was not the feasibility of this approach, but the security implications. Since all security is being handled by PostgreSQL instead of the PostgREST, essentially this system opens the database to the world. Sure not all SQL queries can be generated from PostgREST, but the approach feels like a dangerous idea. In general the whole point of a deployment is to make sure nothing can get to the database and the purpose of most attacks are just to gain access to

the database. For example all SQL injection attacks are literally about running queries directly on the database.

However, by putting all trust on the database and thus properly testing that no SQL query can result in unauthorized access, SQL injection like attacks become useless. If done properly, yes this method would be a sound way to secure data. To guarantee that a deployment is secure would be a matter of using either property based testing or verifying the design using a tool that would generate the needed SQL.

Chapter 7

Conclusion

The Web API developed using this method allows for multiple versions, data security and a very expressive query from the client. The downside is the difficulty of using SQL, but otherwise provides a secure and restricted method of accessing data with any web capable client. The fact that the Web API layer is completely merged with the database layer eliminates any problems that would arise because of the discrepancies between the code serving up the Web API and the underlying database. Since the proof of concept is deployed online in a production like environment with well thought out data security, I can say with certainty that generating a Web API from the underlying data structure is a viable approach to building a back end.

Chapter 8

Appendix

This section has three artifacts: the details of the deployment, the RealWorld API specification for reference and the full README on how the cluster was deployed. The details of the deployment, although complex are not relevant to the purpose of this paper.

8.1 Details of k8 Deployment

The deployment is best understood with what each part was used for rather than a flat list of each component. The request path walks though how a connect gets from the outside world into the cluster, the container deployment is how the Docker containers get on AWS so k8 can deploy them and the deployment with kube is a quick overview of the parts that are required to deploy a k8 cluster. These sections are just to show how each aspect works in practice, see the README for details on how they were deployed.

8.1.1 Request Path

A request starts when some external device makes a request to one of the sub-domains in Route 53, see figure 8.3 at H. For example in Route 53 if I had v1api.example.com then the external device would be making some HTTP or HTTPS request to that domain, see figure 8.3J. Route 53 will have a canonical name (CNAME) record for v1api.example.com has the ELB as its value, see figure 8.3 at C.1. Once the request is sent through the ELB the SSL is striped and the request enters the cluster, see figure 8.3 at D. The reason the SSL is striped at the ELB is so AWS can handle the SSL certificates for me. Since the ELB sits inside the VPC the unencrypted traffic is safe. Finally the request is handled by some application inside the k8 cluster, in my case it passes through an NGINX and Warp instance before being processed by a PostgREST instance that makes the actual requests to the PostgreSQL database.

8.1.2 Container Deployment Path

The container deployment path is how code is deployed on the k8. First the code is put into a git repository on GitLab and setup with a .gitlab-ci.yaml file that tells the GitLab Runner how to build the application and deploy it. See figure 8.3 at I for all the git repositories that I used. Each repository has a Makefile and a few files for k8: a deployment.yaml, a service.yaml and sometimes an ingress.yaml file. These files instruct k8 how to setup, scale and possibly give external access to the application.

By default the GitLab Runner will deploy to the EC2 Container Service on each commit, see the connection between F and I in figure 3.2. The .gitlab-ci.yaml file is the same in each GitLab repository and simply references the /glsmakefile for how to build each Docker container.

To deploy a container to k8 the developer needs to use kubectl on each of the yaml files with the corresponding Docker container commit number. Then the code will be running on one or more of the nodes in the k8 cluster.

It is important to note that I used the GitLab runner to build the PostgREST image because I cannot build Haskell code inside a docker image on OSX, due to a bug in Stack. The default image in the PostgREST repository has too large an attack surface, as with most Docker containers that

have way more than they actually need to run the given application. So early in the project I put PostgREST in an Alpine Linux Docker container. It's not perfect, docker makes that nearly impossible, but close to only having what is needed to run PostgREST.

8.1.3 Deploying with Kube

The deployment requirements are an S3 bucket, see 8.3 at G and a domain name for internal routing at Route53, see 8.3. I also require SSL certificates for the deployment of the ELB, but that can be done after the k8 cluster is deployed. Kube creates its own VPC and kubectl will create the ELB for you when you deploy the ingress.

RealWorld API Spec

RealWorld API Spec

This spec was taken from realworld project.

Authentication Header:

`Authorization: Token jwt.token.here`

JSON Objects returned by API:

Users (for authentication)

```
{
  "user": {
    "email": "jake@jake.jake",
    "token": "jwt.token.here",
    "username": "jake",
    "bio": "I work at statefarm",
    "image": null
  }
}
```

Profile

```
{
  "profile": {
    "username": "jake",
    "bio": "I work at statefarm",
    "image": "https://static.productionready.io/images/smiley-cyrus.jpg",
    "following": false
  }
}
```

Single Article

```
{
  "article": {
    "slug": "how-to-train-your-dragon",
    "title": "How to train your dragon",
    "description": "Ever wonder how?",
```

```
    "body": "It takes a Jacobian",
    "tagList": ["dragons", "training"],
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:48:35.824Z",
    "favorited": false,
    "favoritesCount": 0,
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }
}
```


Multiple Articles


```
{
  "articles":[{
    "slug": "how-to-train-your-dragon",
    "title": "How to train your dragon",
    "description": "Ever wonder how?",
    "body": "It takes a Jacobian",
    "tagList": ["dragons", "training"],
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:48:35.824Z",
    "favorited": false,
    "favoritesCount": 0,
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }, {

    "slug": "how-to-train-your-dragon-2",
    "title": "How to train your dragon 2",
    "description": "So toothless",
    "body": "It a dragon",
    "tagList": ["dragons", "training"],
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:48:35.824Z",
    "favorited": false,
    "favoritesCount": 0,
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }],
  "articlesCount": 2
```

```
}
```

Single Comment

```
{
  "comment": {
    "id": 1,
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:22:56.637Z",
    "body": "It takes a Jacobian",
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }
}
```

Multiple Comments

```
{
  "comments": [{
    "id": 1,
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:22:56.637Z",
    "body": "It takes a Jacobian",
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }]
}
```

List of Tags

```
{
  "tags": [
    "reactjs",
    "angularjs"
  ]
}
```

Errors and Status Codes

If a request fails any validations, expect a 422 and errors in the following format:

```
{
  "errors":{
    "body": [
```

```
      "can't be empty"
    ]
  }
}
```

Other status codes:

401 for Unauthorized requests, when a request requires authentication but it isn't provided

403 for Forbidden requests, when a request may be valid but the user doesn't have permissions to perform the action

404 for Not found requests, when a resource can't be found to fulfill the request

Endpoints:

Authentication:

`POST /api/users/login`

Example request body:

```
{
  "user":{
    "email": "jake@jake.jake",
    "password": "jakejake"
  }
}
```

No authentication required, returns a User

Required fields: `email`, `password`

Registration:

`POST /api/users`

Example request body:

```
{
  "user":{
    "username": "Jacob",
    "email": "jake@jake.jake",
    "password": "jakejake"
  }
}
```

No authentication required, returns a User

Required fields: `email`, `username`, `password`

Get Current User

`GET /api/user`

Authentication required, returns a User that's the current user

Update User

`PUT /api/user`

Example request body:

```
{
  "user":{
    "email": "jake@jake.jake",
    "bio": "I like to skateboard",
    "image": "https://i.stack.imgur.com/xHWG8.jpg"
  }
}
```

Authentication required, returns the User

Accepted fields: `email`, `username`, `password`, `image`, `bio`

Get Profile

`GET /api/profiles/:username`

Authentication optional, returns a Profile

Follow user

`POST /api/profiles/:username/follow`

Authentication required, returns a Profile

No additional parameters required

Unfollow user

`DELETE /api/profiles/:username/follow`

Authentication required, returns a Profile

No additional parameters required

List Articles

`GET /api/articles`

Returns most recent articles globally be default, provide `tag`, `author` or `favorited` query parameter to filter results

Query Parameters:

Filter by tag:

`?tag=AngularJS`

Filter by author:

`?author=jake`

Favorited by user:

`?favorited=jake`

Limit number of articles (default is 20):

`?limit=20`

Offset/skip number of articles (default is 0):

`?offset=0`

Authentication optional, will return multiple articles, ordered by most recent first

Feed Articles

`GET /api/articles/feed`

Can also take `limit` and `offset` query parameters like List Articles

Authentication required, will return multiple articles created by followed users, ordered by most recent first.

Get Article

`GET /api/articles/:slug`

No authentication required, will return single article

Create Article

`POST /api/articles`

Example request body:

```
{
  "article": {
    "title": "How to train your dragon",
    "description": "Ever wonder how?",
    "body": "You have to believe",
    "tagList": ["reactjs", "angularjs", "dragons"]
  }
}
```

Authentication required, will return an Article

Required fields: `title`, `description`, `body`

Optional fields: `tagList` as an array of Strings

Update Article

`PUT /api/articles/:slug`

Example request body:

```
{
  "article": {
    "title": "Did you train your dragon?"
  }
}
```

Authentication required, returns the updated Article

Optional fields: `title`, `description`, `body`

The `slug` also gets updated when the `title` is changed

Delete Article

`DELETE /api/articles/:slug`

Authentication required

Add Comments to an Article

`POST /api/articles/:slug/comments`

Example request body:

```
{
  "comment": {
    "body": "His name was my name too."
  }
}
```

Authentication required, returns the created Comment

Required fields: `body`

Get Comments from an Article

`GET /api/articles/:slug/comments`

Authentication optional, returns multiple comments

Delete Comment

`DELETE /api/articles/:slug/comments/:id`

Authentication required

Favorite Article

`POST /api/articles/:slug/favorite`

Authentication required, returns the Article

No additional parameters required

Unfavorite Article

`DELETE /api/articles/:slug/favorite`

Authentication required, returns the Article

No additional parameters required

Get Tags

`GET /api/tags`

No authentication required, returns a List of Tags

Instructions on how to deploy entire setup

Pfalzgraf

General Structure

I first walk you through creating the kops cluster and then setting up the prerequisites the API. Then I go through putting the demo API online and end with my notes on how to setup GitLab (if you didn't make the PostgREST image on your own computer).

I pulled the monitoring from the project, it just never really gave me back reliable numbers (because of some config issues I had). I would really need another week to get that working properly, so I cut it out.

Notes to future me: helm for deploying heapster (for auto scaling) kept having strange issues. Try deploying it manually next time.

Also, you need to change the secretes in the env file, this whole project is being saved online.

Global Environment Variables

The environment variables are assumed to be present when any of the commands in this document are run.

I wrote an script called env that describes all of the environment variables and should be sourced before running any of the commands below.

Deployment of the K8 cluster on AWS

When you run this, you have no idea if it will work. No this is not because I set this up wrong or did not test it well enough. Simply put, I can run the exact same steps and get different outcomes. I'm not sure what magic is going on at AWS or even with the tools I use. If it does not work, just try it again and it might work Which is why I do not provide a script or Makefile to deploy a cluster.

This section goes through how to deploy a k8 cluster on AWS. There are three sections. The Pre-requisites goes through everything you need to do before calling `kops`, which sets up the actual cluster on AWS. Deployment with Kops is how to invoke `kops`. Validation goes though the steps involved with checking that the deployment is ready for production use.

Pre-requisites

Keep in mind that without an account, none of the other steps can be done. Although you can install the tools beforehand.

Create AWS account

You can get free credit for an AWS account by getting access to the github student pack bundle. I think they will let academics also use their uni email to get some free credit.

After account creation, you can begin the next steps

Create IAM role on AWS

After you create your account you have credentials to what is considered the master of the account, but in general you don't need that kind of power. So instead AWS has IAM that lets you create roles with restricted privileges. Back when I first created my AWS account IAM took you through a wizard to create your administrator account which you should always use instead of the master.

Once you have done that, go ahead and create access keys for the admin account from the IAM console. You will need them to setup the aws tool.

Local Tools

Note: You need to have the same version of kubectl on both client and server or things don't work nicely. You can download the version I used here while the download page is here.

After you deploy you can run `kubectl version` to find out what version `kops` has deployed. Not sure how to ask `kops` before deployment.

aws tool

This install commands assume you have `brew` installed. If not go ahead and go to their website for install instructions.

To install aws you can use `pip` if you are not on a mac, otherwise just use `brew`.

```
brew install awscli
```

The version this project tested with is: `aws-cli/1.11.89 Python/2.7.10 Darwin/16.5.0 botocore/1.5.52`

To configure it with your credentials run:

```
aws configure
```

kubectl tool

Please see the online docs: kubectl install instructinos.

The version that this project tested with is:

```
Client Version: version.Info
{
  Major:"1",
  Minor:"6",
  GitVersion:"v1.6.4",
  GitCommit:"d6f433224538d4f9ca2f7ae19b252e6fcb66a3ae",
  GitTreeState:"clean",
  BuildDate:"2017-05-19T20:41:24Z",
  GoVersion:"go1.8.1",
  Compiler:"gc",
  Platform:"darwin/amd64"
```

```
}
```

kops tool

Your AWS Acces Key ID and AWS Secret Access Key come from the access key you got from the previous step. The default region is not important, but you can set to ap-northeast-2. However, it is important to set the default output format to json. Some of the scripts rely on that aspect.

Next you need to install `kops`, either by downloading a release or running

```
brew install kops
```

The version this project was tested with is: `Version 1.6.0`

jp

I used `jp` version 0.1.1

It is a way to filter json objects that I think is superior to `jq`, but I did not know about it at the start of the project. So unfortunately I use both...

```
brew tap jmespath/jmespath
brew install jp
```

jq

I used `jq` version-1.5

```
brew install jq
```

If I get time I would replace this with jp.

Domain on AWS

You need to have a domain registered on Route53 in AWS. This domain is used for both ingress into the k8 cluster and internal communication in the cluster, so you need to have one before you can deploy the cluster. In the route53 interface they give you an option to both buy a domain and have it automatically managed by them.

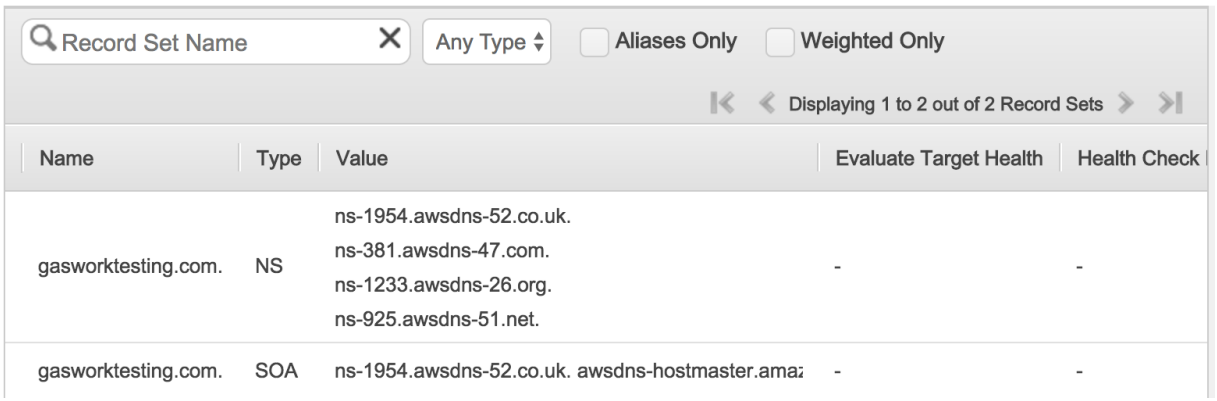

Figure 1:

The next setup is to create certificates for the domain. Us the AWS Certificate Manager to create a wild card certificate for the domain. This is important since we need to create a variable number of sub domains,

all of which need a valid certificate. You can use something like lets encrypt, but that would mean you need to generate a certificate for each subdomain.

SSH keypair

Next you need to upload your rsa public key to AWS so `kops` can put in on the actual virtual machines it will create. To create a new key:

```
ssh-keygen -t rsa -b 4096 -f k8-$REGION.pem
```

Note: I like to call the key the name of the region, example: k8s-ap-southeast-2

You can upload your public key in the EC2 service under Key Pairs.

S3 Bucket

`kops` needs an S3 bucket to store the `terraform` code and k8 config files.

```
aws s3 mb $BUCKET --region $REGION
```

Deployment with Kops

You will need to source the environment variables in the same shell you run this command in:

Note: This command will replace your ~/.kube/config file. So if you have previously setup minicube, you will lose your credentials.

```
kops create cluster \
            --yes \
            --zones $ZONES \
            --ssh-public-key $PUBLIC_KEY_PATH \
            $NAME
```

You might get this kind of error:

```
error determining default DNS zone: No matching hosted zones found for "<your url>";
please create one (e.g. "<your url>") first
```

I have no idea why this happens, but if you deploy the cluster again, a few times, it just works.

To delete the cluster, say if something goes wrong like above, run this:

```
kops delete cluster --yes $NAME
```

Note: sometimes you need to delete the cluster even if nothing actually deployed. Not really sure what state is kept when `kops` fails, but it can prevent `kops` from working correctly.

Validation

Validation is done by running:

```
kops validate cluster $NAME
```

That command outputs a few others that you can try. However, you need to wait quite awhile before they will tell you the cluster works. For me it took about 20min before the cluster was ready to respond. So if it does not work, just wait. For example if you see:

```
cannot get nodes for "<your url>":
Get https://api.<your url>/api/v1/nodes: dial tcp <some ip>:443: i/o timeout
```

Then may mean your cluster is not yet setup and you just need to wait or everything is broken and you will have no idea why.

Docker repository

You need to have some repository for k8 to pull off of. I set one up on AWS in ap-southeast-2, since my cluster is there. Once you have set the `DOCKER_REPO` environment variable to the base URI.

The Makefiles assumes that there is a swagger-ui and postgrest repository under the base URI.

Deploying services on k8

These tend to be a lot more predictable. Either they work flawlessly or throw sometimes useful errors. As such I have put them in Makefiles to speed up deployment and reversals. I will go into detail about how each one works, but at the start of each section I have a summary of just the commands you need to run to deploy that part.

I created a make file to illustrate the parts, but the problem is some of the commands take time after they have completed to finish the operation on the k8 cluster. Makefiles are really poor man's automation tool. So I have used the Makefile where appropriate, but you don't get to just run `make`. Out of scope of this project, but I need to find a better build tool.

Each of these sections should be done in order. You need to deploy the ingress (Default-backend service and Nginx/Warp) first, then comes postgresql, then sqitch and finally the PostgREST service.

K8 dashboard

To check if it is installed run:

```
kubectl get pods --all-namespaces | grep dashboard
```

If not run:

```
kubectl create -f https://git.io/kube-dashboard
```

Note: in the version of `kpos` I use you have to create the dashboard manually.

Default-backend service and Nginx/Warp

This is the service that handles load-balancing and internal routing. It creates the Elastic Load Balancer which all traffic goes through. This is important since that load balancer handles the certificates. Plus this also has the default backend which is part of the internal routing.

The service is split access two different deployments and services, one is the Nginx/warp instance and the other is a image from google. This means the `nginx-ingress-controller-deployment.yaml' is actually made up of two images, the Nginx and the Wrap image. The`nginx-ingress-controller-service.yaml`creates the load balancer with the certs on AWS. While the 'default-backend-deployment.yaml` and `default-backend-service.yaml` is responsible for internal routing.

Note: Services in k8 are how other services "discover" each other. It provides an abstracted view of what a deployment does.

Note: The Nginx might need some changes to the config so swagger-ui works correctly. As of now I can only get swagger-ui to accept the swagger spec from PostgREST if it is run locally.

Note: The Warp instance's purpose is to force ssl, it redirects all HTTP traffic to HTTPS.

Note: find out if when you send a request with a password over HTTP, does the request get rejected before the password is sent?

Note: The traffic between the ELB and the rest of the cluster should be in HTTP, I think I have taken the necessary precautions, but I should ask about it when I get the chance.

Note: The reason why there is a Dockerfile in that repo is so that I could modify the nginx to add the required CORS headers, but I was not successful. There is a program on the nginx-ingress-controller image that creates the nginx.conf when an ingress rule is created in k8. I would need to figure out how that works to actually add in the needed headers. The nginx.conf file in the repo has the headers I think swagger-ui wants.

Setup

You have to lookup the arn in the AWS Certificate Manager online:

Once you get to the place in the image above click on your cert and in the drop down you will see the ARN. Put that into your `env` file as `SSL_CERT_ARN`.

Summary of deployment commands

To Deploy, in the k8/ingress directory run:

```
make
```

To remove, in the same directory run:

```
make clean
```

Postgresql

This uses the `helm` tool to deploy a postgresql cluster on the k8 cluster. So you need to install helm from here.

I am on version:

```
Client: &version.Version
{
  SemVer:"v2.4.2",
  GitCommit:"82d8e9498d96535cc6787a6a9194a76161d29b4c",
  GitTreeState:"clean"
}
```

Essentially `helm` is taking in the values.yaml in the k8/postgresql repo to make the postgresql cluster. The password is generated for us and the other make files use `kubectl` to get it for you.

Summary of deployment commands

In the k8/postgresql folder run, after you have sourced your env file:

```
make setup
```

followed by

```
make
```

Sqitch

This is the demo structure I created for PostgREST to serve up. You will need to run it before PostgREST can get access to the database (since it sets the access passwords and roles).

Just run:

```
make setup
make
```

The setup will forward a local port to the postgresql instance so Sqitch can access the database. I assume that you have done nothing more the instructions I have given. The Makefiles know how to both get the credentials and figure out the names of the pods.

PostgREST

This one has a few pain points. The reason I started using environment variables in the first place was so I could deploy different versions of PostgREST, but the result is quite clumsy. If you want to deploy say `v1` you set the `DBSCHEMA` environment variable to 'v1'. When you run `make` in the k8/PostgREST folder it will create a new deployment, service, ingress and url to access a PostgREST instance that access the schema v1. If you want to remove that version you run `make clean` and its all removed for you, but if you want to remove say version `v2` you have to switch the environment variable to `v2` before you can remove it.

This is a problem with this since `make clean` is only going to remove the version with the corresponding `DBSCHEMA`. In reality I need to have a configuration file for each version that is handled separately. I think `helm` may be able to fill that role, but that is beyond the scope of this project.

There is another part to keep in mind: all instances use the same `JWT_SECRET`, `DBURI` connection and the anonymous role is hard coded to be anon concatenated with `DBSCHEMA`. So you have to make sure in the sqitch files that the authenticator role can switch to each anonymous role, that each role gets access to the group types for each version (in fact when creating a new version you need to grant all existing roles with any new group types that they should be a part of) and make sure the anonymous role follows the pattern anonDBSCHEMA (example anonv1). Of course the only parts that the PostgREST instance can see are whatever is in the schema `DBSCHEMA`.

Note: the `AWS_HOST_ID` env uses `DOMAINNAME` to figure out which of the `hostedZones` is the right one. Just make sure the `DOMAINNAME` is only the domain. For example: gasworktesting **NOT** gasworktesting.com

Note: The `ELB` env variable is filled with the first elastic load balancer it finds. I might be able to do something where I ask k8 what vpc the cluster is in and use that to filter the results, but for now just keep this is mind.

Pre-requisites

The k8/PostgREST repo just has the deployment scrips, the PostgREST image itself you need to have on a docker registry where k8 can access it. I used the AWS's docker hub registry to host the image and GitLab-runner to build the image / upload the image. I have a separate repository that is a fork of PostgREST

with my preliminary work for deploying the service, deployment images, build images and GitLab-runner's configuration. There is a Makefile that shows you how to upload the image from your local machine, but you have to build the docker image on linux for that to work. If you want to setup the GitLab-runner as I have, please see that section.

Then you will need to update the `GIT_REVISION_POSTGREST` environment variable.

In the PostgREST repo I have my fork that I built the PostgREST image with. Please see that repo for details on how to build the image.

Summary of deployment commands

To deploy version 1, set DBSCHEMA to `v1` in your env script and run:

```
make
```

To remove run:

```
make clean
```

To access go to here and you will get the swagger spec.

Swagger-UI

Very similar to the way PostgREST is setup. The version is also handled with DBSCHEMA env variable. So to deploy the version 1 of swagger-ui you just need to have set DBSCHEMA to v1. Then:

```
make
```

And to remove (while having the DBSCHEMA set to v1)

```
make clean
```

Then to access go here. If you also created version 2 then the swagger-ui would be here.

GitLab-runner

Here are my notes are setting up the GitLab-runner and other instructions I needed.

- Launch a new machine on AWS w/ 4GB RAM
- [ ] It should have the same zone as your repositories
- [ ] It should be an Ubuntu 16.04 LTS AMI
- Update & Upgrade and use 4.8 Kernel (GHC problems otherwise)

```
apt-get update
apt-get -y upgrade
apt-get -y dist-upgrade
apt-get -y install \
        linux-signed-image-4.8.0-46-generic \
        linux-image-extra-4.8.0-46-generic \
        linux-headers-4.8.0-46-generic \
        linux-tools-4.8.0-46-generic
reboot
```

- Install stack

```
curl -sSL https://get.haskellstack.org/ | sh
```

- Install make (or just build-essential)

```
apt-get -y install build-essential
```

- Install the AWS CLI

```
apt-get -y install python-pip
pip install --system awscli
```

- Install gitlab-runner https://docs.gitlab.com/runner/install/linux-repository.html
- [ ] Install Docker

  ```
  curl -sSL https://get.docker.com/ | sh
  ```

- [ ] Install gitlab runner

  ```
  curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-ci-multi-runner/script.deb.s
  cat > /etc/apt/preferences.d/pin-gitlab-runner.pref <<EOF
  Explanation: Prefer GitLab provided packages over the Debian native ones
  Package: gitlab-ci-multi-runner
  Pin: origin packages.gitlab.com
  Pin-Priority: 1001
  EOF
  apt-get update
  apt-get -y install gitlab-ci-multi-runner
  usermod -a -G docker gitlab-runner
  ```

- Configure gitlab-runner

You'll need the url & token from the gitlab website (under Settings -> CI/CD Pipelines). Choose the shell runner. `gitlab-ci-multi-runner register` `systemctl restart gitlab-runner`

- Steps to secure the server

In the AWS security group set Http and Https inbound to only accept connections from gitlab (172.31.20.160/32). Then turn on ufw logging to get the ip address

I also restricted ssh to my ip adress

Note: there are enviroment variables that GitLab can set, lookinto later.

- Fixing: "Your Authorization Token has expired"
- run `aws configure` and give it the GitLab_Docker_Repo credentials

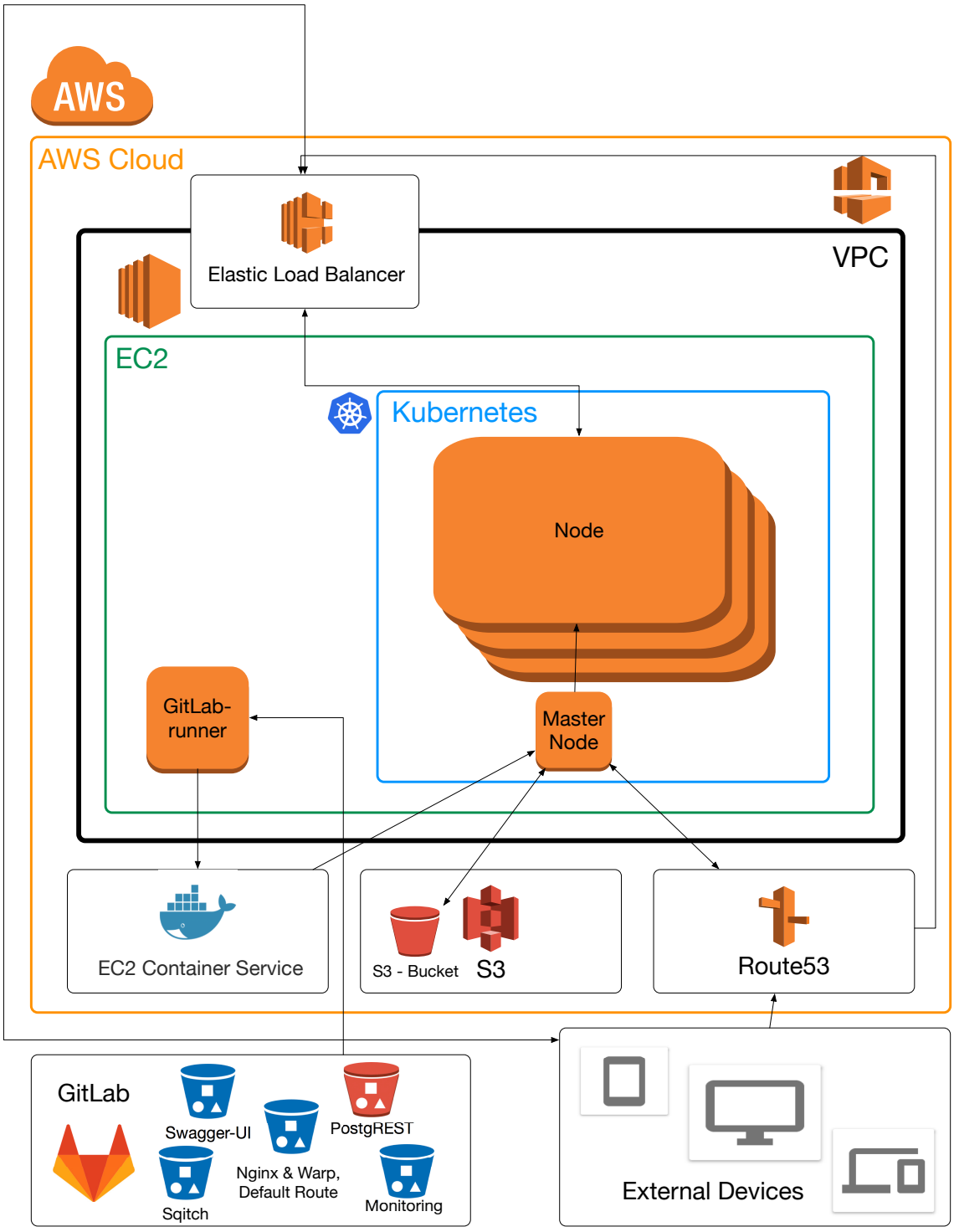

Figure 8.1:
The deployment I created to illustrate PostgREST in production. To know more about any of the elements see: GitLab, GitLab Runner, Elastic Compute Cloud (EC2), AWS, Elastic Load Balancer (ELB), Virtual Private Cloud (VPC), PostgreSQL, EC2 Container Service, Simple Storage Service (S3), Route 53, Relational Database Service (RDS), Kubernetes, sqitch, NGINX, Swagger UI, Warp, PostgREST. Monitoring is made up of Prometheus and Grafana.

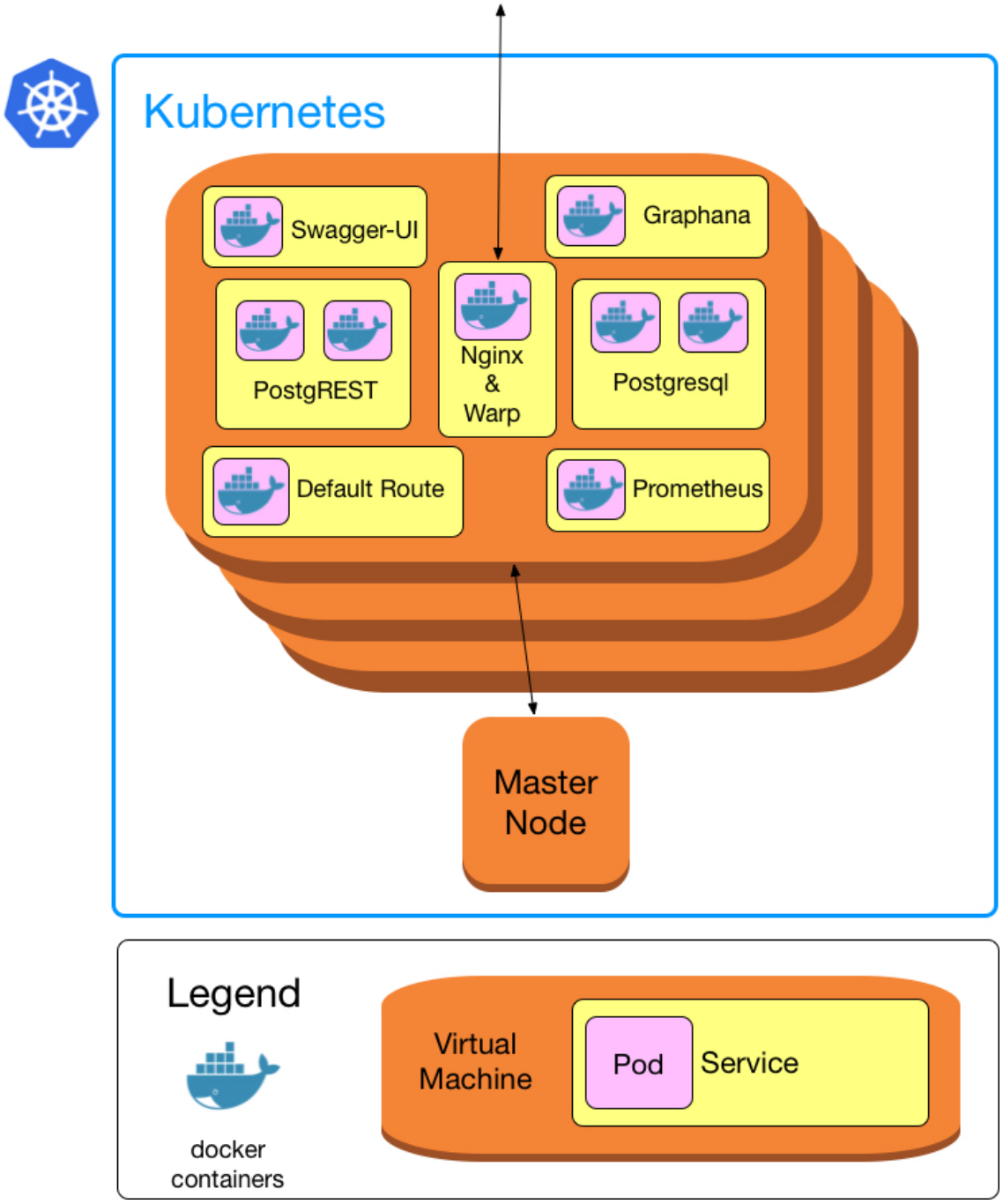

Figure 8.2:
This figure illustrates that there are any number of physical nodes (virtual machines) where pods are running anywhere on those nodes. In my deployment I have two nodes, but it is trivial to add more nodes. To know more about the services I have deployed on k8 please see: Swagger UI, Grafana, PostgREST, PostgreSQL, NGINX, Warp, Prometheus. The Default Route is just a black hole for connections to go when a route does not exist.

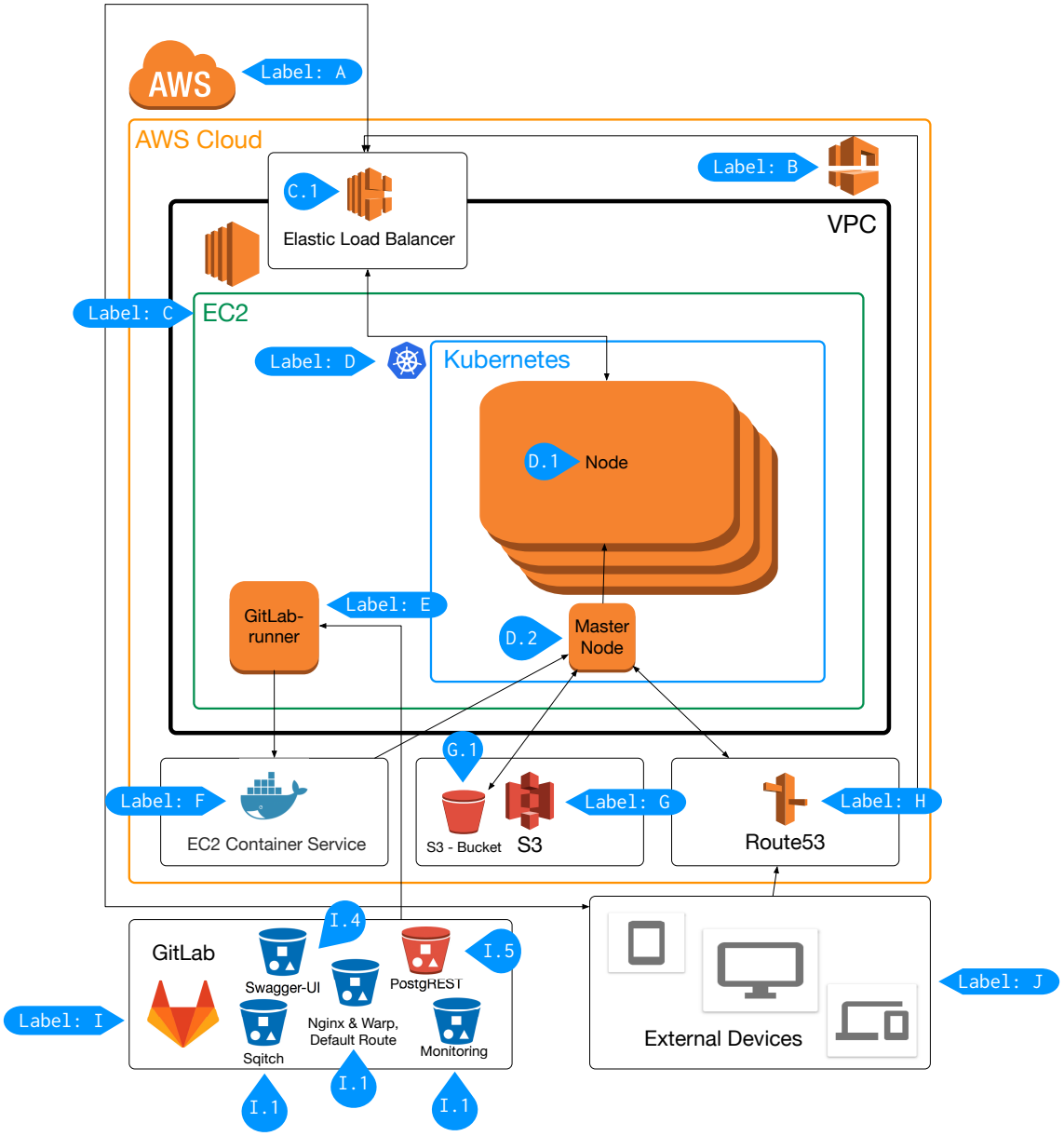

Figure 8.3:
It is important to note that there may by any number of virtual machines on EC2 that act as the nodes in k8, which is why there is a stack of Nodes at number 5.

Bibliography

[aws17] What is aws? - amazon web services. https://aws.amazon.com/what-is-aws/, 2017. (Accessed on 05/11/2017).

[doc] What is docker? https://www.docker.com/what-docker. (Accessed on 05/04/2017).

[Dre] Dreamfactory | api automation. https://www.dreamfactory.com/. (Accessed on 06/01/2017).

[Fie00] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000. (Accessed on 05/04/2017).

[git] Code, test, and deploy together with gitlab open source git repo management software | gitlab. https://about.gitlab.com/. (Accessed on 05/04/2017).

[Gra] Graphcms - graphql based headless cms. https://graphcms.com/. (Accessed on 06/01/2017).

[gra17a] Grafana - the open platform for analytics and monitoring. https://grafana.com/, 2017. (Accessed on 05/04/2017).

[gra17b] Graphql | the data layer for modern apps. http://www.graphql.com/, 2017. (Accessed on 05/11/2017).

[jsoa] Json. http://json.org/. (Accessed on 05/04/2017).

[JSOb] Json web tokens - jwt.io. https://jwt.io/. (Accessed on 05/04/2017).

[Mas11] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, 2011.

[ngi] nginxinc/kubernetes-ingress: Nginx and nginx plus ingress controllers for kubernetes. https://github.com/nginxinc/kubernetes-ingress. (Accessed on 05/04/2017).

[OAu] Oauth 2.0 — oauth. https://oauth.net/2/. (Accessed on 05/04/2017).

[Posa] Motivation — postgrest 0.4.0.0 documentation. https://postgrest.com/en/v0.4/intro.html#declarative-programming. (Accessed on 05/04/2017).

[Posb] Overview of role system — postgrest 0.4.0.0 documentation. https://postgrest.com/en/v0.4/auth.html#ssl. (Accessed on 03/30/2017).

[posc] postgraphql/postgraphql: A graphql api created by reflection over a postgresql schema. https://github.com/postgraphql/postgraphql. (Accessed on 06/01/2017).

[Posd] Postgresql: About. https://www.postgresql.org/about/. (Accessed on 05/05/2017).

[Pose] Postgresql: Documentation: 9.5: Database roles. https://www.postgresql.org/docs/9.5/static/user-manag.html. (Accessed on 03/23/2017).

[Posf] Postgresql: Documentation: 9.5: Schemas. https://www.postgresql.org/docs/9.5/static/ddl-schemas.html. (Accessed on 03/22/2017).

[Pos17a] Postgresql: Documentation: 9.6: Concepts. https://www.postgresql.org/docs/9.6/static/tutorial-concepts.html, 2017. (Accessed on 05/07/2017).

[Pos17b] Postgresql: Documentation: 9.6: Create view. https://www.postgresql.org/docs/current/static/sql-createview.html, 2017. (Accessed on 05/07/2017).

[Pro] Prometheus - monitoring system & time series database. https://prometheus.io/. (Accessed on 05/04/2017).

[RES11] Restful api design - second edition - youtube. https://www.youtube.com/watch?v=QpAhXa12xvU, November 2011. (Accessed on 05/08/2017).

[Rum14] B. Rumpe. Executable Modeling with UML. A Vision or a Nightmare? *ArXiv e-prints*, September 2014.

[Sno11] Michael Snoyman. Warp: A haskell web server. http://steve.vinoski.net/pdf/IC-Warp_a_Haskell_Web_Server.pdf, June 2011. (Accessed on 05/04/2017).

[Sno17] Snoyberg. Persistent :: Yesod web framework book- version 1.4. http://www.yesodweb.com/book/persistent, April 2017. (Accessed on 05/30/2017).

[sqi] Sqitch by theory. http://sqitch.org/. (Accessed on 05/08/2017).

[swa] Swagger specification. http://swagger.io/specification/. (Accessed on 05/04/2017).

[swa17] swagger-api/swagger-codegen: swagger-codegen contains a template-driven engine to generate documentation, api clients and server stubs in different languages by parsing your openapi / swagger definition. https://github.com/swagger-api/swagger-codegen, 2017. (Accessed on 05/07/2017).

[Wha] What is kubernetes? | kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (Accessed on 05/04/2017).

[yam] The official yaml web site. http://yaml.org/. (Accessed on 05/04/2017).

[Yor17] Yorhel. An opinionated survey of functional web development. https://dev.yorhel.nl/doc/funcweb, May 2017. (Accessed on 06/01/2017).