JEFRi - JSON Entity Framework Runtime

April 8, 2012

Contents

1 Overview

JEFRi is an entity framework. The idea of an EF is to model software as close to the problem domain as possible. JEFRi accomplishes this by specifying an Entity Context independent of the programming language. This context describes the entities available in the domain model, the properties of those enties, and the attributes of entities and properties. Further, it describes the relationships between entities. These navigation properties allow entities to act naturally and transparently as composite types. This allows intelligent loading of related data, without requiring an application to load an entire object graph.

Entities are analogous to Objects in OOP, and Tables in databases. They are the building blocks of the domain, the things the domain is modeling. These could be receipts in a Point of Sale application, Clients in Contact Management System, or Planets in a physics model. Properties are the pieces of data that describe an entity — the transaction date of the receipt, or the name of the client. Attributes provide meta-information about entities and properties. All entities have a key attribute, that tells which property holds the entity's keys. All properties have a type attribute, describing what data they hold. Beyond some minimal required data, attributes can contain any additional information the application may need. It is the responsibility of the application to use that data — the context will merely hold it.

The word *Object* occurs rather often in this documentation. In general, it does not mean an instance of a class. Rather, *Object* follows the Javascript meaning of a key/value store. *Entity* is used closer to the idea of an instance in classical inheritance.

2 Keys

Every Entity must have a key, a single unique field identifying that entity across all instances of the runtime. To meet this requirement in a distrubuted, heterogeneous environment, keys are version 4 (random) UUIDs. Keys are used exclusively to refer to entities of a specific type. To maintain unique object graphs through flattening into transactions, the key is used as an object identifier, saving the need to handle issues of using memory references or incrementing integers as keys.

Entity keys are immutable.

Keys should be generated by whichever runtime initially creates a new instance of an entity. Thus, it is important that JEFRi instances provide suitably robust random number generators. By having the key assigned by the runtime that first creates an entity, applications eliminate the need for a central repository of "canonical" key information. Without a central key store, applications can be free to handle concurrency with whatever mechanism best fits the domain model. Further, since applications don't need to synchronize keys, significant reductions in bandwidth are seen. The concern of conflicting keys is mitigated by choosing a sufficiently large key domain. Further, keys must only be unique to entity types. Thus, randomly generating Version 4 UUIDs allows 6 million billion keys per entity type before reaching a 50% chance of a single collision amongst keys.

Since a type's full URN within a runtime is technically entity://application/context/type/uuid, it is arguable that UUIDs are "Too Much". Then again, no one needs more than 640K. If a context expects more entities of some type than this, it would be simple to replace the key generator implementation with something more suitably robust for that application.

3 Relationships

Relationships have two ends, the "From" and and the "To" end. These ends change roles depending on which entity is accessing the relationship. The "From" end is always the side of the relationship doing the accessing, and the "To" end is the side being accessed. Ends have a "Type" and a "Property". The "Property" tells which property holds the key for that end, and the "Type" property describes how the "To" end should be accessed. Ends have two types- "has_a" and "has_many". A third type, "is_a", is a specialized "has_a".

Relationships can be navigated by calling the appropriate get_navprop method of the entity. For example, if a Receipt has an associated Client, you could call receipt.get_client() which would return a single Client. Conversely, from the client end, you could call client.get_receipts() to return an array of Receipt entities. Notice how the navigation methods are properly pluralized. This is a result of naming the relationships "user" and "receipts" in the context- it is not the runtime's responsibility to handle pluralities, but the context generator's.

4 Contexts

JEFRi Contexts are descriptions of the entity model. Contexts are described here as JSON objects, though can be any persistable object serialization. Contexts are objects with two properties — meta, describing the overall context options, and entities, an array of entity descriptions.

A context's meta is an object containing arbitrary keyed data for the runtime's consumption. For example, a meta could declare "Persist On Update" to be true. In this case, the runtime could then automatically attempt to persist an entity any time its properties were updated. The list of canonical meta attributes and their meaning will grow organically as JEFRi is fleshed out. Otherwise, the documentation for a particular implementation must specify the Meta attributes it handles, as well as their meanings.

The entities array is the heart of the context. Each entry describes one entity available. The entry is an object with five keys. Those are 'name', the canonical name for the entity type; 'key', the property containing the entity's UUID; properties, an array containing the properties of the entity; relationships, an array containing descriptions of the various navigation properties; and attributes, an object describing various metadata about the entity itself.

The name is the canonical type name for an entity. It is used extensively to look up property definitions from several points in the application, especially when creating new entities, persisting entities, and navigating entity relationships. Entity names must be unique within a context. Entity names must be valid Javascript object keys- aka, =∼ `/[a-zA-Z_$][a-zA-Z0-9_$]+/`

The key is the name of the property containing the entity's key. See the section on entity keys for more details on how keys are used in JEFRi.

Properties is an array of objects similar in layout to entities, but without keys or relationships. Specifically, properties must define a name, a type, and an array of attributes. The name is the Javascript-safe property identifier, and must be unique among the properties of an entity. The type is the platform-agnostic datatype specifier. Canonically recognized types are int, float, and string. Attributes is an object of arbitrary key/value pairs. Currently recognized keys are nullable, and length for string. Again, it is the runtime implementor's responsibility to document recognized attributes and their behavior.

Relationships is an array describing the various navigation properties available to an entity. A relationship has four keys— name, the identifier for the navigation; type, one of "is_a", "has_a", or "has_many"; and to and from, describing the entities taking part in the relationship. Name must be a locally unique, valid Javascript identifier. Further, it defines which get_ method must be called to navigate the relationship (specifically, entity.get_'name'()). Type determines what will be returned when navigating the relationship. has_a and is_a will return a single entity, while has_many will return a (possibly empty) array of entities. Also, there is some indication that the far side of a has_a or has_many relationship will probably have a has_a or has_many navigation that would lead back to the entity on the from side (however, this is not a requirement). is_a will probably not have a reverse navigation.

Finally, the from and to properties describe where to look up the data needed for the navigation. In all relationships, both the from and to type fields *must* resolve to a valid entity name in the same context (TODO possibly change this so there is a posisbility for cross-context navigation?). The property fields are used slightly differently depending on what the type of the relationship is. In a has_a relationship, the to.property should be the same as the to.type.key, and the navigation property will return the one to.type whose key is equal to the from.property. The opposite is similar for has_many.

4.1 Persistence Stores

Instances of JEFRi contexts need some mechanism for long-term persistence of entity data. Persistence stores provide this by providing a flexible, driver-based approach. Persistence stores must provide a persist and a get method, each taking a JEFRi transaction (see next section) and performing the appropriate actions. Stores should also provide an is_async method, which tells the user whether to expect the store to return immediately, or happen in the background with a callback.

The persist method must function atomically per transaction. It is the responsibility of the persistence store to ensure that transactions happen atomically. In general, it is the persistence store's responsibility to meet the semantics specified in the section on Transactions.

The get method must handle the myriad of filter and join operations available in the transaction specifications. See the section on transaction specifications for the exact semantics of get requests.

5 Transactions

The most important, fundamental concept in computer programing is the word-level atomic write. Since the first computers, it was impossible to read an inconsistent word of memory- if a read and a write occur "simultaneously", the read will either return the value in memory BEFORE the write, or the value in memory AFTER the write. It will NEVER return some of the before and some of the after bits. Of course, at a hardware level, this only works with the smallest addressable batch of memory— a word. Since entities are probably always going to be bigger than a single word (UUIDs alone are 2 full words on a 64-bit machine), it is not possible to achieve atomic entity persistence in hardware alone. Still, it would be really nice to have guaranteed atomic writes.

JEFRi transactions fill in that gap. When a runtime is ready to persist a batch of entity updates, it builds a serialized object representation of just the pieces that need to be updated. By using entity keys extensively, the runtime can flatten the object graph into an object list, with guaranteed unique references between the entities. This transaction detail can then be sent between runtime instances, with some guarantee that that information is consistent (generally, that guarantee will come from TCP).

JEFRi has two transaction types: get and persist. A get transaction is used to request data from a remote JEFRi instance. A persist transaction is used to update data at a remote JEFRi instance. Transactions are a serialized object with two properties: meta, describing the transaction, and entities, describing the entities to deal with.

In a GET transaction, the entities array is a template for which entities should be returned. The return transaction will contain all entities that meet the criteria in the transction. Specifically, the remote instance will look at the first object in the transaction. It will return all entities of that type whose properties match the given properties. It will join specified navigation properties (using just 'name', not get_name). It will do this for each entity in the array. In other words, the individual entities specify AND conditions, while the multiple entities specify OR conditions.

In a PERSIST transction, the entities array has the literal data that must be persisted. The remote JEFRi runtime is responsible for making the persist atomic and returning a transaction filling in the additional details it has about these entities that was not included in the original transaction (eg, a JEFRi instance may persist an entity to a runtime using DB2 as a backend, and one property has an update trigger setting the timestamp to 'now'— the reply transaction would include the updated value for that property).

5.1 Transaction Specifications

Data for a transaction should be described as close to the domain model as possible. However, requiring only full entities would result in more verbose transactions than necessary. To that end, the specification to a transaction will be an array of objects describing the entities in the transaction. The format should be similar to:

```
[{
    type: 'User',
    surname: 'Franken',
    prop2: value2,
    cards: {}
}]
```

If this were a GET transaction, the reply would have an array of entities of User and Card, with all the User entities having surname = 'Franken', prop2 = value2, and all the Cards correctly associated with the matched Users. Navigation property objects can specify constraints on the nav properties returned, and can themselves declare navigation properties to return. It is not necessary to specity a type in a navprop — any that are will be ignored.

If this were a PERSIST transaction, the key property *must* be included. The persistant store would update (or store) the values of the entity with the properties passed in. Navigation properties should not be included in a transaction (they will be ignored). To save nav properties, each entity should be added individually.

5.1.1 Gory Get Details

To capture the full expressivity developers need in describing data, there are several rules transactions will use to determine the entities to return from a GET transactions.

If the property is the entity key, the value must be a valid UUID. It will be converted to an appropriate format for the store, and the property must match exactly. While the UUID can be transmitted as a hex string, it is recommended to transmit it as a 16-byte integer to conserve space.

If value is an interger, properties must match exactly.

If value is a float, either it should be matched at a 2^{-8} threshold or it can be the first number of an array tuple, whose second parameter specifies the threshold precision.

If the value is a string, it will be treated as an SQL LIKE operation. Returned entities will have properties whose values contain the string (case sensitive).

If the value is an array, the first field in the array may specify an operation, or be a floating point number (see above). Valid operations are any of $<$, $\leq$, $>$, $\geq$, $=$, and `REGEX`. `REGEX` treats the next value as a PCRE regular expression, and any returned entities will have that property matching the regex. Otherwise, the operations are as specified by the store, but generally should be numerical ordering for numbers, and lexicographical ordering for strings.

SUM? AVERAGE? COUNT? MAX? MIN?

GROUP BY?

If the value is an array whose first parameter is not an operator or another array, the returned entity's property will match as an IN clause, with itegers matching exactly and string matching as described above. If the first parameter is an array, then each element of the value's array is treated as an ANDed where clause, with the componentes of sub arrays following the same (value) or (operator, value) rules presented here.

Using a common SQL driver,

```
[{
    _type: 'User',
    surname: 'Franken',
    date_of_birth: [
        ['<=', 1200000000],
        ['>=', 1100000000]
    ],
    cards: {}
}]
```

becomes

```
SELECT
    User.user_id AS user.user_id ,
    User.surname AS user.surname ,
    User.given_name AS user.given_name ,
    User.login AS user.login ,
    User.date_of_birth AS user.date_of_birth ,
    Card.card_id AS card.card_id,
    Card.user_id AS card.user_id,
    Card.name AS card.name,
    Card.email AS card.email,
    Card.phone AS card.phone,
    Card.address AS card.address,
    Card.city AS card.city,
    Card.state AS card.state,
    Card.postal AS card.postal,
    Card.country AS car.country,
FROM USER
LEFT JOIN Card on User.user_id = Card.card_id
WHERE
    User.surname LIKE '%Franken%'
    AND User.date_of_birth <= 1200000000
    AND User.date_of_birth >= 1100000000
```

Since this is a left join and there are no constraints on Card, this query will return a number of rows equal to the total number of matched cards. With more than one contact card per User, some of the User data will be duplicated. The Entity Context the store is running in will intern all the instances of the different Users, so there will be no duplicated entity instances in memory.

5.2 Best Persist Practices

Persist transactions are an application's way to send changed entities to another instance. For a database-backed entity store, these transactions could return quickly. For a store that has to post the transaction to a remote host, the persist might take much longer. To unify the interface, all persists are asynchronous, and must be passed a callback. With this in mind, there are three seperate techniques or approaches to handling a persist transaction.

The first approach is managing the transaction directly. First, the application readies a transaction from the EntityContext by calling `ec.transaction()`. The application can then add a number of entities to the transaction via `transaction.add(entity)`. Finally, the application shold call `transaction.persist(callback)` to persist and finalize the request. Persist can be called multiple times on a transaction; however, there is no way to remove an entity from a transaction. The same entity can be added multiple times without being duplicated in the final send.

The second is persisting the changes to a single entity. Every entity has a function `persist(callback)`. This function will call another function, `entity.on_persist(transction)`, passing in the transaction that will handle the persist request. The entity should use this to add related data to the transaction. This technique can be combined effectively with the first approach.

The final technique is a pair of helper methods in the EntityContext itself. The EC maintains an internal record of what entities have been changed, and what entities have been added. The two methods `ec.persist_new(callback)` and `ec.persist_changes(callback)` can be used to easily persist all entities that have been modified, but not persisted.

5.3 Entity Accounting

To facilitate minimizing data transfer, entities and the EntityContext keep track of changes that occur during runtime. The EntityContext maintains an array of new entities and an array of modified entities. These can be used to quickly index what needs to be sent in `persist_changes` or `persist_new`.

Entities maintain accounting information regarding their status. This is stored in two properties and a function.

`__new` Boolean property, set to true if this is a new entity.

`__modified` Object whose keys are property names that have been updated, and those values are the value when the object was last persisted (or the default value, if the entity is new). When an property of an entity is set, the entity should check in its __modified object for the property. If the property is not in __modified, set __modified[property] to the current value of entity[property], update entity[property], and add itself to the EntityContext's _modified list. If the property *is* in __modified, and the value in __modified is the same as the setter's value, then the entity should set its property and remove the property from __modified. If that was the last property in __modified, the entity should remove itself from the EntityContext's modified list.

`_status` Is a function returning one of 'NEW', 'MODIFIED', or 'PERSISTED'.

Every entity has a default `persisted` event handler that clears the modified object, clears the new flag, and removes the entity from the EntityContext's new and modified arrays.

6 API

Context Methods are called on an instance of a JEFRi runtime, and include many data access routines. Entity methods are called on instances of entities, and include primarily navigation properties, as well as entity-specific persistence methods. Transaction methods are called on a class which manages large GET and PERSIST transactions. Data Store methods mirror Transaction methods, but are generally only called from the transaction. However, there is no formal 'user-mode' constraints, so that is a recommendation, not a requirement.

6.1 Context Methods

`EntityContext(URI, protos)` The context CTOR. Creates a new context from the json at the URI given, and extends the entity classes with the protos.

`definition(type)` Returns the context's definition for the specified type.

`extend(type, proto)` Add the methods in proto to type's prototype. Will effect *ALL* instances, both current and future, of type.

`build(type, obj)` Return an non-interned instance of type with properties filled in from obj.

`intern(entity, updateOnIntern)` Returns a canonical instance of the entity from memory. Use this to translate between UUIDs and pointers (or similar machine references). If the boolean `updateOnIntern` is true, and there is a previously stored entity, the old entity will be updated with any non-default properties of the new entity.

`expand(transaction)` Take a return transaction and expand the entities in it. Intern where possible, and call `add_‘name‘` and `set_‘name‘ where` appropriate.

6.2 Entity Methods

`id()` Return the UUID of the object, indepented of whichever property actually stores it.

`_type()` Returns a string of the canonical entity type.

`get_‘name‘()` Follow a navigation property. Returns either a single or (possibly empty) array with the to end of the relationship.

`set_‘name‘(to_entity)` Set the has_a navigation property. Correctly updates everyone’s nav property ids.

`add_‘name‘(to_entity)` Add to the has_many array.

`encode(writer)` Pass the entity, its properties, and its nav properties to the transaction writer.

6.3 Transaction Methods

`Transaction(entities)` Constructor. Entities describes the data in the transaction. See the secton on Transactions for details.

`add(entities)` Add these entities to the transaction.

`meta(attributes)` Specify meta attributes. Currently supported are

`get()` Run the transaction as a get transaction.

`persist()` Run the transaction as a persist transaction.

6.4 Events

6.4.1 EntityContext

`saving/saved` Called before and after the transction is persisted inside `persist_new` and `persist_changes`.

6.4.2 Entity

`on_persist(transaction)` called during single entity persistence.

`persisting()`

`persisted()`

6.4.3 Tranasaction

`getting()`

`gotten()`

`persisting()`

`persisted(data)`

`sending()`

`sent()`

`continuation()`

6.4.4 PersistStore

`getting`

`gotten`

`persisting`

`persisted()`

6.4.5 Order

This is the order of events fired during a PersistStore GET request.

```
transaction.getting()
store.getting()
store.sending()
$.ajax(fn(){
    ec.expand(data)
    store.gotten();
    transaction.gotten()
    callback();
});
store.sent();
```

7 Reference

7.1 Sample Context

```
{"meta":{},
"entities":[
    {   "name": "User",
         "key": "user_id",
         "properties": [
            {   "name": "user_id",
                "type": "int",
                "attributes": {
                    "primary": "true"}},
            {   "name": "surname",
                "type": "string",
                "attributes": {
                    "length": "45"}},
            {   "name": "given_name",
                "type": "string",
                "attributes": {
                    "nullable": "true"}},
            {   "name": "login",
                "type": "string",
                "attributes": {
                    "length": "255",
                    "unique": "true"}},
            {   "name": "date_of_birth",
                "type": "float",
                "attributes": {
                    "nullable": "true"}}],
        "relationships": [
            {   "name": "cards",
                "type": "has_many",
                "to": {
                    "type": "Card",
                    "property": "user_id",
                    "vname": "user"},
                "from": {
                    "type": "User",
                    "property": "user_id",
                    "vname": "user"}},
            {   "name": "authinfo",
                "type": "has_a",
                "to": {
                    "type": "Authinfo",
                    "property": "user_id",
                    "vname": "user"},
                "from": {
                    "type": "User",
                    "property": "user_id",
                    "vname": "user"}}],
        "attributes": {
              "vname": "users",
              "svname": "user"}},

    {   "name": "Manager",
        "key": "user_id",
        "properties": [
            {   "name": "user_id",
                "type": "int",
                "attributes": {
                    "primary": "true"}}],
        "relationships": [
            {   "name": "user",
                "type": "is_a",
                "to": {
                    "type": "User",
                    "property": "user_id",
                    "vname": "user"},
                "from": {
                    "type": "Manager",
                    "property": "user_id",
                    "vname": "user"}}}],
        "attributes": {
              "vname": "managers",
              "svname": "manager"}},

    {   "name": "Executive",
        "key": "user_id",
        "properties": [
            {   "name": "user_id",
                "type": "int",
                "attributes": {
                    "primary": "true"}}],
        "relationships": [
            {   "name": "user",
                "type": "is_a",
                "to": {
                    "type": "User",
                    "property": "user_id",
                    "vname": "user"},
                "from": {
                    "type": "Executive",
                    "property": "user_id",
                    "vname": "user"}}],
          "attributes": {
               "vname": "executives",
               "svname": "executive"}},

     {    "name": "Card",
          "key": "card_id",
          "properties": [
              {    "name": "card_id",
                   "type": "int",
                   "attributes": {
                       "primary": "true"}},
              {    "name": "user_id",
                   "type": "int",
                   "attributes": {}},
              {    "name": "name",
                   "type": "string",
                   "attributes": {
                       "nullable": "true"}},
              {    "name": "email",
                   "type": "string",
                   "attributes": {
                       "nullable": "true"}},
              {    "name": "phone",
                   "type": "string",
                   "attributes": {
                       "nullable": "true"}},
              {    "name": "address",
                   "type": "string",
                   "attributes": {
                       "nullable": "true"}},
              {    "name": "city",
                   "type": "string",
                   "attributes": {
                       "nullable": "true"}},
              {    "name": "state",
                   "type": "string",
                   "attributes": {
                       "nullable": "true",
                       "length": "5"}},
              {    "name": "postal",
                   "type": "string",
                   "attributes": {
                       "nullable": "true",
                       "length": "10"}},
              {    "name": "country",
                   "type": "string",
                   "attributes": {
                       "nullable": "true",
                       "length": "2"}}],
          "relationships": [
              {    "name": "user",
                   "type": "has_a",
                   "to": {
                       "type": "User",
                       "property": "user_id",
                       "vname": "user"},
                   "from": {
                       "type": "Card",
                       "property": "user_id",
                       "vname": "user"}}],
          "attributes": {
               "vname": "cards",
                "svname": "card"}},

    {   "name": "Authinfo",
        "key": "authinfo_id",
        "properties": [
            {   "name": "authinfo_id",
                "type": "int",
                "attributes": {
                    "primary": "true"}},
            {   "name": "user_id",
                "type": "int",
                "attributes": {}},
            {   "name": "username",
                "type": "string",
                "attributes": {
                    "length": "45"}},
            {   "name": "password",
                "type": "string",
                "attributes": {
                    "length": "45"}},
            {   "name": "activated",
                "type": "string",
                "attributes": {
                    "nullable": "true",
                    "length": "45"}},
            {   "name": "banned",
                "type": "string",
                "attributes": {
                    "nullable": "true",
                    "length": "45"}}],
        "relationships": [
            {   "name": "user",
                "type": "has_a",
                "to": {
                    "type": "User",
                    "property": "user_id",
                    "vname": "user"},
                "from": {
                    "type": "Authinfo",
                    "property": "user_id",
                    "vname": "user"}}],
        "attributes": {
                "vname": "authinfo",
                "svname": "authinfo"}}
]}
```

7.2 Sample GET

```
{
    _type: "Datapoint",
    datatype: {
        campaign: {
            campaign_id: "aac068f1-4fea-4922-8a32-7d8cf8f2923a"
        },
        options: "[boolean]",
    },
    product: {
        brand: {}
    },
    task: {
        location: {
            country: "AU"
        }
    }
}
```

7.2.1 Translated SQL

```
SELECT
    datapoints.datapoint_id AS 'datapoints.datapoint_id',
    datapoints.task_id AS 'datapoints.task_id',
    datapoints.datatype_id AS 'datapoints.datatype_id',
    datapoints.value AS 'datapoints.value',
    datapoints.product_id AS 'datapoints.product_id',
    tasks.task_id AS 'tasks.task_id',
    tasks.parent_id AS 'tasks.parent_id',
    tasks.user_id AS 'tasks.user_id',
    tasks.campaign_id AS 'tasks.campaign_id',
    tasks.location_id AS 'tasks.location_id',
    tasks.title AS 'tasks.title',
    tasks.description AS 'tasks.description',
    tasks.earliest_start AS 'tasks.earliest_start',
    tasks.deadline AS 'tasks.deadline',
    tasks.estimate AS 'tasks.estimate',
    locations.location_id AS 'locations.location_id',
    locations.latitude AS 'locations.latitude',
    locations.longitude AS 'locations.longitude',
    locations.name AS 'locations.name',
    locations.address AS 'locations.address',
    locations.city AS 'locations.city',
    locations.state AS 'locations.state',
    locations.postal AS 'locations.postal',
    locations.phone AS 'locations.phone',
    datatypes.datatype_id AS 'datatypes.datatype_id',
    datatypes.campaign_id AS 'datatypes.campaign_id',
    datatypes.name AS 'datatypes.name',
    datatypes.description AS 'datatypes.description',
    datatypes.options AS 'datatypes.options',
    campaigns.campaign_id AS 'campaigns.campaign_id',
    campaigns.client_id AS 'campaigns.client_id',
    campaigns.name AS 'campaigns.name',
    campaigns.description AS 'campaigns.description',
    products.product_id AS 'products.product_id',
    products.brand_id AS 'products.brand_id',
    products.name AS 'products.name',
    products.description AS 'products.description',
    products.size AS 'products.size',
    brands.brand_id AS 'brands.brand_id',
    brands.client_id AS 'brands.client_id',
    brands.name AS 'brands.name',
    brands.description AS 'brands.description'
FROM "datapoints"
    JOIN "tasks" ON (datapoints.task_id = tasks.task_id)
    JOIN "locations" ON (tasks.location_id = locations.location_id)
    JOIN "datatypes" ON (datapoints.datatype_id = datatypes.datatype_id)
    JOIN "campaigns" ON (datatypes.campaign_id = campaigns.campaign_id)
    JOIN "products" ON (datapoints.product_id = products.product_id)
    JOIN "brands" ON (products.brand_id = brands.brand_id)
WHERE
    campaigns.campaign_id LIKE 'aac068f1-4fea-4922-8a32-7d8cf8f2923a'
    AND
    datatypes.options LIKE '%[boolean]%'
```