CORONA

instant application server

Todd Bandrowsky

Country Video Games

introduction

Corona is a simple integrated application and database server for Windows designed to make it simple to build effective service stacks. It can serve entirely standalone, as its own database and web server, with a single consistent object oriented and rich querying language.

Corona

- Simplify your development with a object oriented web API that can have classes either in its own databases or connected OBBC databases, all from a single schema file.
- Supports out of the box instant B2C experience.
- Internal user communities, such as a company and federated consultants, can have one set of permissions and workflow options, and end customers can have another, just by specifying the configuration.
    - Teams based security.
    - Teams based workflow.
- Create rich applications with near real time analytics embedded into the application workflow.
    - Join, group by, project and filter across any class in the system.
- Easy to develop from any client.

Corona Schema:

- Supports classes, and derived classes
- Supports its own change management, with versioned changes for classes, objects and imports
- Supports imports of delimited files

command line and configuration

Corona is launched from the command line. It accepts a single parameter indicating the name of its configuration file.

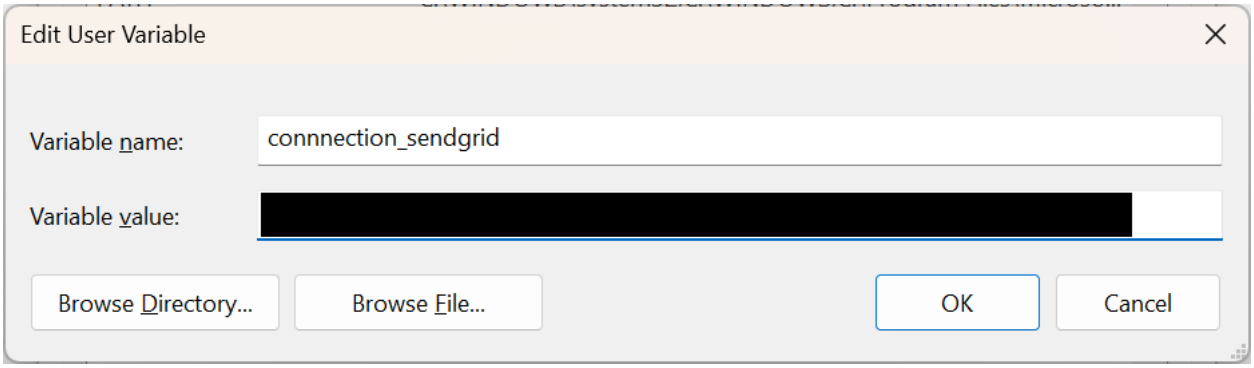
The configuration file looks like this. It contains things like ports, server connections, and importantly, where the schema file lives.

This is from the candidate_config.json that comes with the system as an example.

The system uses the sendgrid api to send out enrollment emails, users getting a confirmation code.

Connections is a map of names followed by an ODBC connection string.

Both the connections and the SendGrid can be overlaid by using environment variables. This is forced to be upper case to match the convention of upper case environment variables. Use CONNECTION_SENDGRID for the send grid string, and CONNECTION_*SOURCENAME* for a sql server connection.



In this example, you would use *CONNECTION_ADVENTUREWORKS2016* to override the contents of the config file.

```
{
  "SendGrid": {
    "ApiKey": "Your key goes here"
  },
  "Connections": {
    "AdventureWorks2016": "Driver={ODBC Driver 17 for SQL Server};Server=BANDROWSKY-
RIPP\\DEV01;Database=AdventureWorks2016;Trusted_Connection=yes;TrustServerCertificat
e=yes"
```

```
  },
  "Server": {
    "listen_point": "http://localhost:5678/corona",
    "application_name": "coronademo",
    "schema_filename": "candidate_schema.json",
    "database_filename": "candidate_database.cdb",
    "sys_user_name": "system",
    "sys_user_password": "systempassword",
    "sys_user_email": "todd.bandrowsky@countryvideogames.com",
    "new_user_default_team": "Guests"
  }
}
```

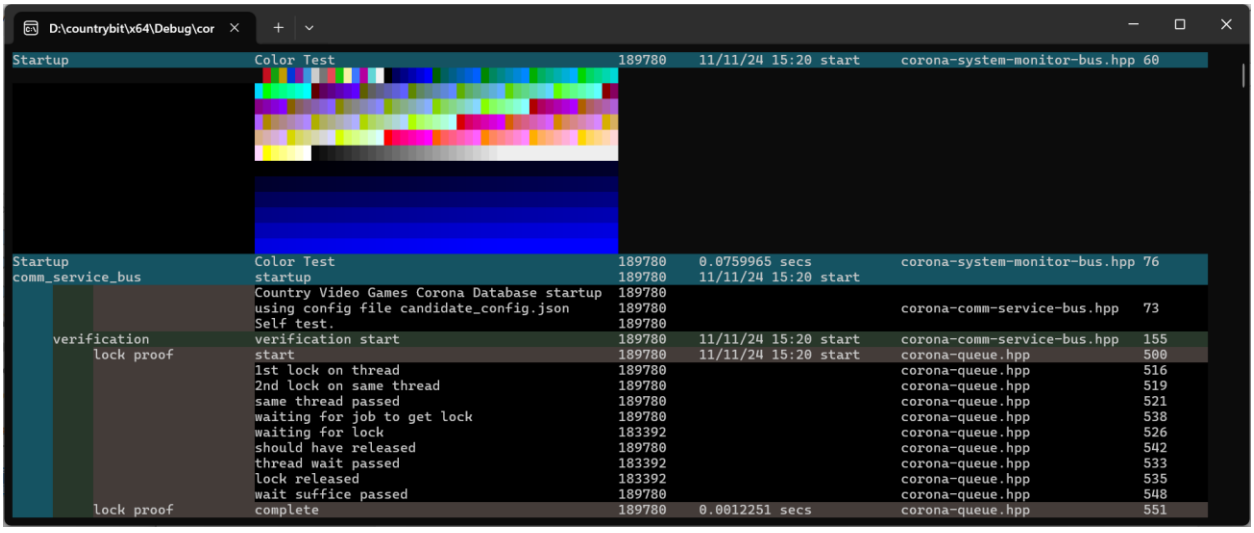
When a database is created, the system must create a super user account that is the root of all things. These are created by the sys_user* settings above. new_user_default_team isn't implemented.

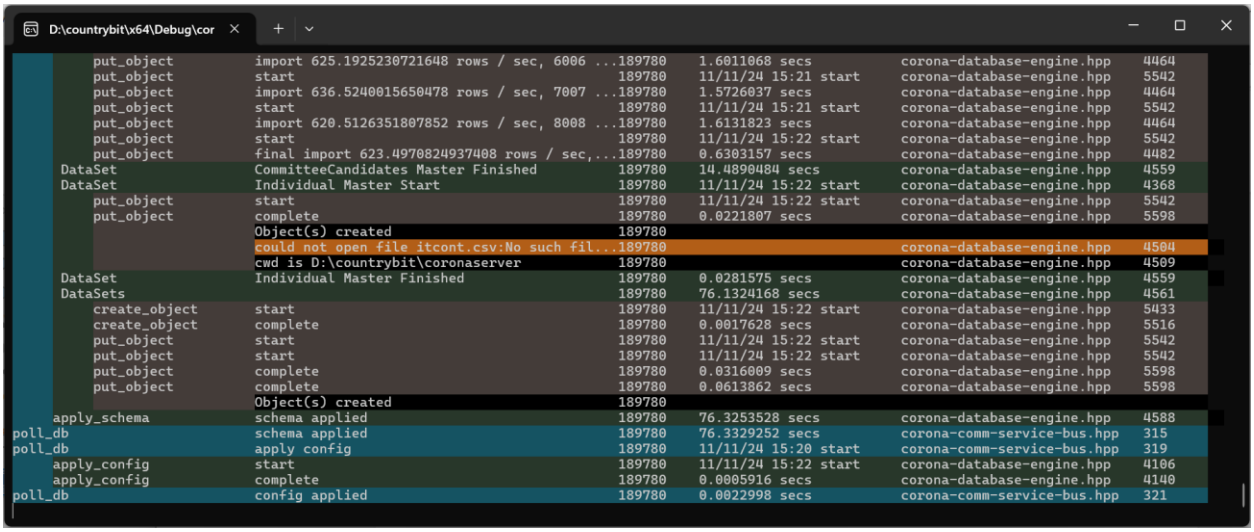
Corona creates a single database file, .cdb, for the database specified by a schema. This lives at the file above.

The schema file is applied at start up and during modification.

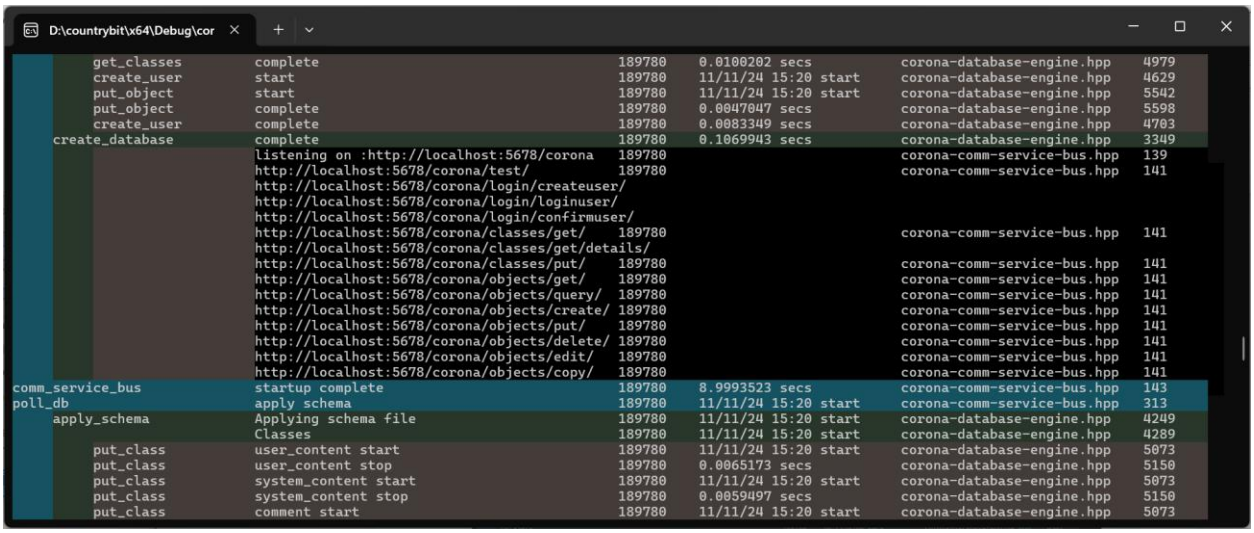
Once Corona starts, it tests the colors on the terminal, and then,

a) Walks through unit tests.
b) Reads and applies the config.
c) Creates the database, if needed.
d) Works through the schema and applies changes.





Corona can handle requests during startup. https requests are logged to the console. Binding points will be displayed during start up like so.



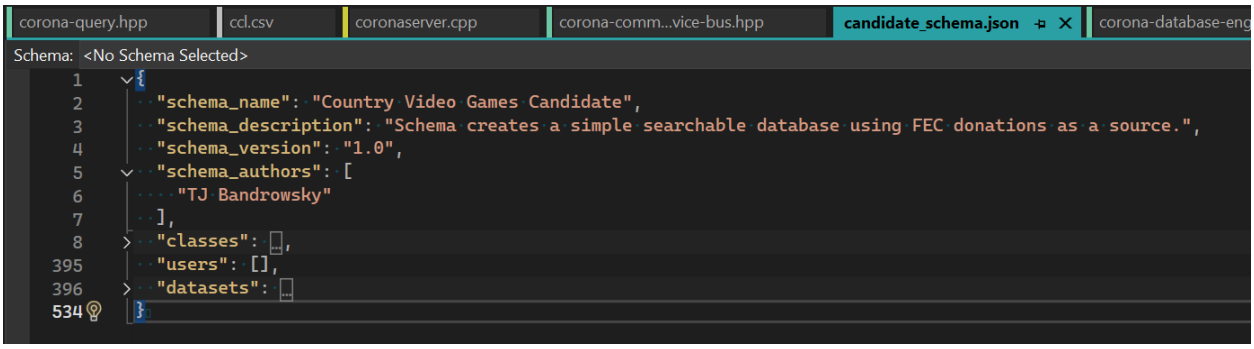
Additionally, there is a postman collection that you can use to experiment with the server.

classes, schema and system objects

The schema file defines a corona database. It is a start-up-time and modify-time script of API commands to create classes, with data loads organized into datasets, each of which only runs once or when its version changes.

The schema file is what defines a Corona application. It describes the classes in a system, and the permissions different teams can have to use them.

The section about the schema file covers all of the types of things one needs to create a corona object database, and what each means, in detail. Additionally, each topic also covers the APIs that go with it.



schema definition

A schema file has a name, description and version, along with author credits, and then has three main sections, *classes, users* and *datasets.*

users

Corona has users. Users are generally self-sign on, but, you can directly specify users in the schema definition file.

user definition

class_name	The class_name of the user – sys_user
first_name	The first name, like, John.
last_name	The last name, like, Smith.
user_name	A user name. This gets specified at creation time, but, if there is a duplicate, the system will slap a random number on the back of it, and thus you will be named for all eternity, numbered by a number that you never get to pick. 42 might always be you.
mobile	Cell phone number. This will soon be enforced for MFA
street1	Street addresses
street2	Street addresses
City	City
State	State
Zip	Zip – this release is just USA
Password	Internally, this is the users's hashed password. You should never see this.
team_name	The name of the team of which the user is a member. The team is looked up with this name when a user logins or is confirmed.
workflow_objects	A map, keyed by class_name, of the object_id of the each of the activity objects for this user. These objects can be used for home page data, different search gateways to different functions, and so on.
validation_code	This is the validation code generated for the user, internally. This never leaves the database.
confirmed_code	This is the confirmed code entered by the user. This never leaves the database either. In any case, the _code fields are only temporarily lived during the send code / receive code mechanism.

users in schema file

users are specified in an array of the users in the schema file. The idea is that, if you have a schema file, the user goes straight in, as put. *Warning*: this hasn't been fully tested as of this writing. So give it a go and let us know if something goes wrong.

corona users api

There are several apis for working with users in Corona. Taken together, they implement the classic user story of

- Users self-sign and self-serve: new users are onboarded automatically depending on their team domain assignments.
- Team sets for different user groups sets up their permissions, provides a set of their own navigation objects that are set up correctly, and lets them self-serve their own passwords and account details.
- Supports sending a code to email for registration and emergency authentication to change passwords. Formal 2FA is coming once we add Twilio mobile support.
- User objects, such as teams and grants and users, are all just normal corona objects and you can work with them like anything else – if you have permissions.

corona/login/create_user	Request

	{
	"user_name" : "companydrone",
	"email" : "MEAYLN"
	}

	Result

	{
	"data": {
	"token": "",
	"user_name": "companydrone",
	"validation_code": "MEAYLN"
	},
	"message": "Ok",
	"seconds": 6.824127,
	"success": 1.000000,
	"token":"blah"
	}

	Discussion

	This API is to implement the user enter the code form for login confirmation. If the put validation code is correct, then, good, other it will fail.

corona/login/confirmuser	Request

	{
	"user_name" : "companydrone",
	"validation_code" : "MEAYLN"
	}

	Result

	{
	"data": {
	"token": "",
	"user_name": "companydrone",
	"validation_code": "MEAYLN"
	},
	"message": "Ok",
	"seconds": 6.824127,
	"success": 1.000000,
	"token":”blah”
	}

	Discussion
	This API is to implement the user enter the code form for login confirmation. If the put validation code is correct, then, good, other it will fail.

corona/login/loginuser	Request

	{
	"user_name" : "userdrone",
	"password" : "testo12345!"
	}

	Result

	{
	"data": {
	"class_name": "sys_user",
	"created": "2024-11-18T11:37:26Z",
	"created_by": "system",
	"email": "todd.bandrowsky@countryvideogames.com",
	"navigation": {

<table>
<tr><td></td><td><pre>
"candidate_search": {
    "candidates": [],
    "city": "",
    "class_name": "candidate_search",
    "full_name": "",
    "object_id": 37841,
    "political_party": "",
    "state": "",
    "updated": "2024-11-18T11:37:34Z",
    "updated_by": "system"
},
</pre>
etc</td></tr>
<tr><td></td><td>Discussion</td></tr>
<tr><td></td><td>This API allows the system to login a user. If the login is successful, a token is returned. This token should be used as an Authorization : Bearer token on subsequent requests for that user.

Additionally, the API returns the navigation objects specified by the team the user is associated with. A savvy user agent can then use these objects to help construct the user home page or navigation experience.</td></tr>
<tr><td>corona/login/senduser</td><td>Request</td></tr>
<tr><td></td><td><pre>
{
    "data" : {
        "user_name" : "userdrone"
    }
}
</pre></td></tr>
<tr><td></td><td>Response</td></tr>
<tr><td></td><td><pre>
{
    "data": {
        "user_name": "userdrone"
    },
    "message": "Ok",
    "seconds": 10.778928,
    "success": 1.000000
}
</pre></td></tr>
<tr><td></td><td>Discussion</td></tr>
<tr><td></td><td>Forces a resend of a new confirmation code to the email address the user provided</td></tr>
<tr><td>corona/login/passworduser</td><td>Request</td></tr>
<tr><td></td><td><pre>
{
    "data" : {
        "user_name" : "userdrone",
        "validation_code" : "PHQGHU",
</pre></td></tr>
</table>

	"password1" : "superpass123!", "password2" : "superpass123!" } }
	Response

	{
	"data": {
	"password1": "superpass123!",
	"password2": "superpass123!",
	"user_name": "userdrone",
	"validation_code": "PHQGHU"
	},
	"message": "Ok",
	"seconds": 7.530513,
	"success": 1.000000
	}

	Discussion
	Password set lets a logged in user change their own password, or get a confirmation code via email so they can enter that for when they forget what they changed it too.
/corona/objects/query	Request

	{
	"from" : [
	{
	"class_name" : "sys_user",
	"name" : "sys_user",
	"filter" : {}
	}
	],
	"stages" : [
	{
	"class_name" : "filter",
	"name" : "sys_user",
	"input" : "sys_user"
	}
	]

	Response
	"data": [{ "class_name": "sys_user", "confirmed_code": 0.000000, "created": "2024-11-18T19:08:23Z", "created_by": "system",

	"email": "todd.bandrowsky@gmail.com", "object_id": 2, "password": "A59F29E3A63D87BDB620206CC7AD413C7CC64E8E50EA57F7AB6B30E0065D46C9", "user_name": "system", "validation_code": "LFDXFI" }, { "class_name": "sys_user",
	Description
	This is an example of using standard object api to get system user tables – *if you have the permission.* Normal users don't get these permissions, and users won't get any permissions to anything unless they are on a team, and the team only gets what it asks for.

teams

Corona extends the concept of a security role into a team. A team has security permissions, like a role does, but it also defines workflow. A team is a collection of people doing the same kind of job, with the tools for those people that do that job. It has the object permissions, but it also gives you a path from a user login as to what the user is allowed to do.

sys_team class

sys_team is the class of whose objects define the teams in Corona. One creates an object of type sys_team, fills it out and then calls put to save it. The change takes effect immediately. objects of sys_team , like all objects, can be examined via the get_object call, and they can also be queried, being a source in the from_clause.

The class itself looks like this. It has the class_name sys_team, which is derived from sys_object. The class has a nice and pleasant description, and then, finally, it has fields.

team_name: the name of the team.
team_description: a long description of the team.
team_domain: the domain name of users emails who are members of the team.

permissions: an array of at least sys_grants that get auto linked to this via that child object mechanism.

In the child object mechanism of permissions, we can observe that while sys_grants is the only member of child_objects, there could be more. sys_grant is another class, and this construct means only sys_grant or sys_grant derived objects belong in permissions. And, when a sys_grant is created or edited by this object, then, it will have its values initialized from the two handlers – copy_values, and construct_values. This is just a simple map of look from to get to.

```
{
	"class_name" : "sys_team",
	"base_class_name" : "sys_object",
	"class_description" : "Teams a user can belong to",
	"fields" : {
			"team_name" : "string",
			"team_description" : "string",
			"team_domain" : "string",
			"permissions" : {
				"field_type" : "array",
				"field_name" : "permissions",
				"child_objects" : {
					"sys_grant" : {
						"child_class_name" : "sys_grant",
						"copy_values" : {
							"object_id" : "team_id"
						},
						"construct_values" : {
							"object_id" : "team_id"
						}
					}
				}
			},
			"workflow_classes" : "array"
	},
	"indexes" : {
	  "sys_team_name": {
	    "index_keys": [ "team_name" ]
	  },
	  "sys_team_email": {
	    "index_keys": [ "team_domain" ]
	  }
	}
}
```

Then, two indexes are created, one for the name of the team, and one for the domain. The latter is what is used to resolve request emails.

An object instance of a sys_team might look more like this. Note that, the permissions are not exploded out. As we'll see later on, you can use the flag include_children on a get object call or a query to have the system bring you back everything:

```
  {
        "class_name": "sys_team",
        "created": "2024-11-18T19:08:25Z",
        "created_by": "system",
        "object_id": 5,
        "permissions": [],
        "team_description": "company admins",
        "team_domain": "countryvideogames.com",
        "team_name": "candidate_admins",
        "workflow_classes": [
            "committee_search",
            "candidate_search",
            "people_search"
        ]
    },
```


Now let's look at the permissions as they came back. There's an object for each class, with grant_class being the class that has been granted, the fields alter, delete, get, put describe what can be done with each of those, along with the object id and class of the grant object itself.


```
  {
        "class_name": "sys_team",
        "created": "2024-11-18T19:08:25Z",
        "created_by": "system",
        "object_id": 5,
        "permissions": [
            {
                "alter": "none",
                "class_name": "sys_grant",
                "created": "2024-11-18T19:08:25Z",
                "created_by": "system",
                "delete": "own",
                "get": "any",
                "grant_class": "user_content",
                "object_id": 6,
                "put": "own",
                "team_id": 5
            },
            {
                "alter": "any",
                "class_name": "sys_grant",
                "created": "2024-11-18T19:08:25Z",
                "created_by": "system",
                "delete": "any",
                "get": "any",
                "grant_class": "system_content",
                "object_id": 7,
```

```
                "put": "any",
                "team_id": 5
            }
        ],
        "team_description": "company admins",
        "team_domain": "countryvideogames.com",
        "team_name": "candidate_admins",
        "workflow_classes": [
            "committee_search",
            "candidate_search",
            "people_search"
        ]
    },
```

That's a bit, so let's turn this into some more real world examples.

Company Vendor User Team Example

Consider this set up for a company. I have a single corona based application shoved in there like superglue to stitch together company C, it's vendor V, and some set of user U. I want to have the C company people with their stuff to do, the vendor V in the system with their stuff, and finally U consuming what they consume.

In Corona, we break these audiences into teams. So we have a company_team, a vendor_team, and a user_team, a team for each group of people, bound by a similar set of functions. In Corona terms, those are navigation objects and permissions.

Let's assume the company, giganto, has a domain likewise named, and that's the one employees use for their email : giganto.com.

"team_name": "giganto_team"
"team_description": "our awesome company cashing us up."
"team_domain": "giganto.com"

So, if someone tries to create a user with giganto domain, they are in, once they confirm.

Now, while giganto.com is pretty big, there do happen to be a few people out there that use yahoo, google, pretty much anyone that you don't know. Those are all the customers.

"team_name": "customer_team"
"team_description": "our genius customers we love you."

```
“team_domain”: “*”
```

No worries about our vendor though.

```
“team_name”: “supervendor_team”
“team_description”: “our genius super vendor.”
“team_domain”: “supervendor.com”
```

Now, with that much, we have three teams, but, we still have to say more about what they will do. First thing up, is permissions. We’ll just apply permissions based on this imaginary story. A company owns a web store, which received products from a vendor, and sells them to customers. Armed with our vivid imagination, we can conjur up classes such as company_class, vendor_class, and customer_class. We might then decorate our teams like this:

```
“team_name”: “giganto_team”,

“permissions” : [

{ “grant_class”:“company_class”, ”get”:”any”, ”put”:”own”, “delete”:”none”,
“alter”:”none” },

{ “grant_class”:“vendor_class”, ”get”:”any”, ”put”:”none”, “delete”:”none”,
“alter”:”none” },

{ “grant_class”:“customer_class”, ”get”:”any”, ”put”:”none”, “delete”:”none”,
“alter”:”none” }]
```

This says, giganto users can do what they want to with themselves, but, can’t modify other team’s stuff.

```
“team_name”: “customer_team”,

“permissions” : [

{ “grant_class”:“customer_class”, ”get”:”own”, ”put”:”own”, “delete”:”none”,
“alter”:”none” }

]
```

This says, customers are in their own shared world.

```
“team_name”: “customer_team”,

“permissions” : [
```

{ “grant_class”:“customer_class”, ”get”:”own”, ”put”:”own”, “delete”:”none”, “alter”:”none” }

]

This says, vendors can work on their own vendor_class objects but can use company_class objects as a reference perhaps.

“team_name”: “vendor_team”,

“permissions” : [

{ “grant_class”:“company_class”, ”get”:”any”, ”put”:”none”, “delete”:”none”, “alter”:”none” },

{ “grant_class”:“vendor_class”, ”get”:”own”, ”put”:”own”, “delete”:”none”, “alter”:”none” }

]

Additionally, we’re going to add putting a filter condition on the class, so that, you can have teams dividing up a class by one or more key columns.

Team workflow classes

Finally, if anything, for navigation and dashboard purposes, a team can specify what classes of objects should be created and attached to the user’s account. Objects can contain queries so you can construct a dashboard and also use them to handle a search and list dynamic. The user agent may call edit_object followed by run_object, repeatedly, as users search. Following the graph of creatable objects from these graphs is a workflow.

Workflows may be specified on teams as follows. They are just string login names of classes. When a user is confirmed on a team though, their account gets their own copy of each of these objects. The user has objects created automatically to keep track of and securely present their results of queries to various items they are assigned to manage them in the system. How that is presented is up to the user agent.

```
"workflow_classes": [
        "committee_search",
        "candidate_search",
        "people_search"
    ]
```

sys_team api

The sys_team class and its objects can be accessed via the normal corona object and class api.

API	Purpose
corona/objects/query	query and perform analytics on corona objects. To get started, try putting a sys_team in a from clause.
corona/objects/get	Get an object by its class_name and object_id. To get started, take on of those ids and classes and give this a go
corona/objects/create	Creates an object using a class fields, and creating an id for it.
corona/objects/put	Puts an object, saving or creating it internally – if it passes validation.
corona/objects/erase	Deletes an object
corona/objects/edit	Brings back a single object and the class used to define it, in such a way as to support edit on the fly.
corona/objects/run	Save a modified object, run its query fields, and then return the result.

classes

Classes are conceptually a place to put data when organizing it. Classes in Corona are like tables or collections.

Classes have

- Name and Description – class_name and class_description.
- Base Class – base_class_name gives the name of a base class. You must have a base class that is sys_object, or your own class. This is because sys_object contains the object_id, which is global to the database, and tracks the user names and dates of object modifications.
- Fields – a map of fields, consisting of field names to simple description. Fields may have validation rules, and there can be query rules.
- Indexes – names of indexes
- Sql – specifies mappings to an existing SQL source.

core class

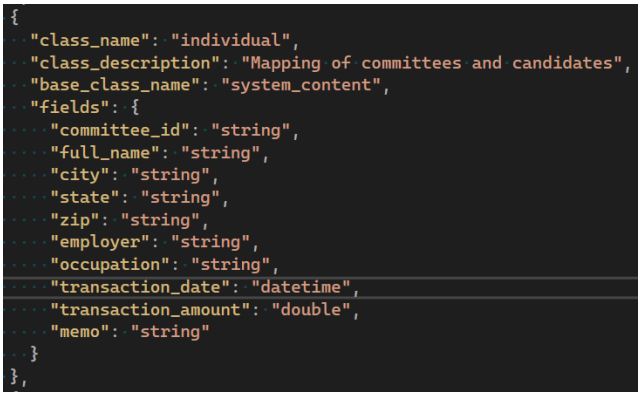
```
{
  "class_name": "comment",
  "class_description": "Comments made by users",
  "base_class_name": "user_content",
  "fields": {
    "comment_text": "string"
  }
},
{
  "class_name": "link",
  "class_description": "Links made by users",
  "base_class_name": "user_content",
  "fields": {
    "link_url": "string"
  }
},
```

The core of the class is its own name, and relationship to other classes in the system. Corona classes have a name, a description, and a single base class, from which it may inherit everything.

class_name	Unique name of class
class_description	Human description
base_class_name	Either use sys_*object* as a base class, or your own. Single inheritance only.

fields

A Corona class can have fields. There's a long way and a short way to specify a field. The quick way is a field name mapped to one of several types:



```
{
  "class_name": "individual",
  "class_description": "Mapping of committees and candidates",
  "base_class_name": "system_content",
  "fields": {
    "committee_id": "string",
    "full_name": "string",
    "city": "string",
    "state": "string",
    "zip": "string",
    "employer": "string",
    "occupation": "string",
    "transaction_date": "datetime",
    "transaction_amount": "double",
    "memo": "string"
  }
},
```

The long way is to specify an extended specification. Only the long way gives you extra validation by letting you set up constraints.

Corona Field Types

string	A string field
int64	A 64 bit integer, useful for ids
double	A number
datetime	A datetime
array	An array, in the JSON sense
object	An object, in the JSON sense
query	Query based on using members of the object as parameters.
drop_down	A scalar with a class for a source. For lookups

extended field specifications

Instead of a simple field type string, one supplies an object.

"my_field" : "string",

One might do

"my_field": {

```
“field_type”:”string”
}
```

These extended options are useful for validation. Corona automatically validates when an object is put.

Extended options for strings.

min_length	An integer indicating the minimum length of a string
max_length	An integer indicating the maximum length of a string
match_pattern	A regular expression that the string must match
enum	A list of strings, that the string must be one of. Case insensitive.

Extended options for int64, double and datetime

min_value	Minimum value of field
max_value	Maximum value of field

Extended options for arrays and objects

child_objects	Specifies child classes that may be an object member, or an object element of an array field.

arrays, objects and child objects

By default, Corona lets you put anything into an object and an array. But Arrays and objects can be constrained by class, and if so, Corona will break out the objects into parent and child tables and keep track of the mappings for you. Conceptually, to do this, you need to have a constructor for a new child, and an assignment for an existing child. In this way, if you have an inbound json with children, Corona classes will know where to put them.

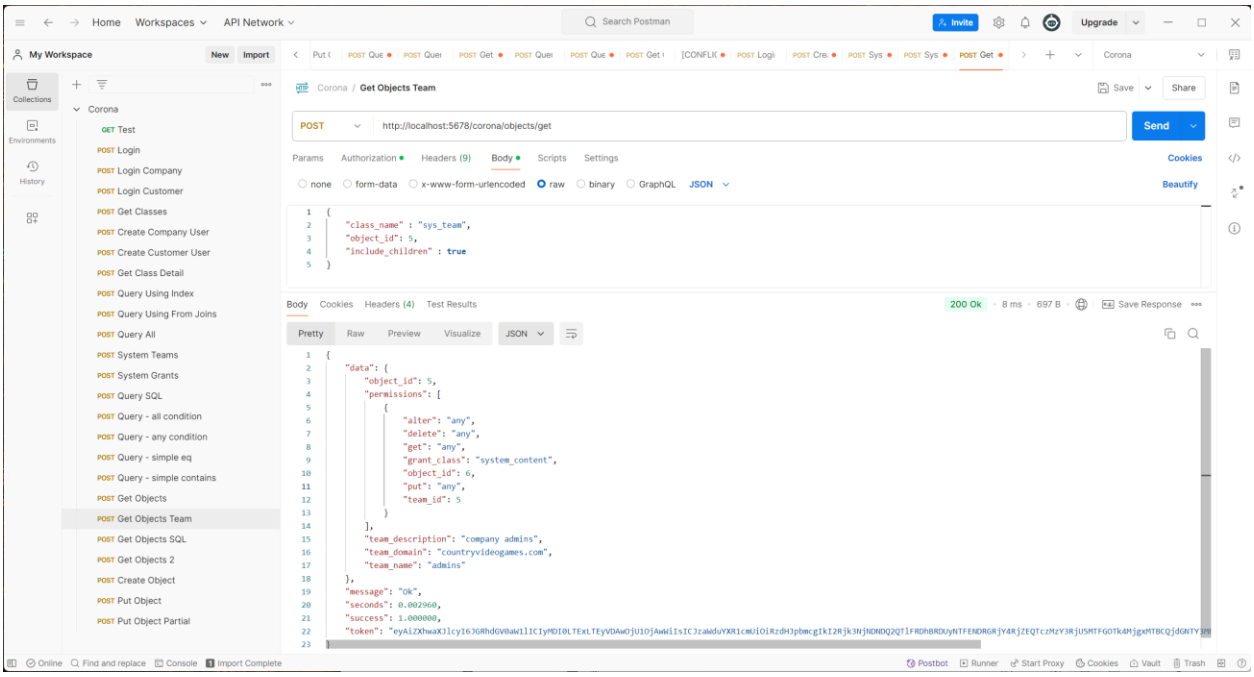
The structure you need to create to get this effect is “child_objects”.

A child objects looks like this. Here we have a “permissions” field, which is an array. On that array, there is a child_objects which allows the contents of the array to be one of the types specified, or a derived class.

```
"permissions" : {
    "field_type" : "array",
    "field_name" : "permissions",
    "child_objects" : {
        "sys_grant" : {
            "child_class_name" : "sys_grant",
            "copy_values" : {
                "object_id" : "team_id"
            },
            "construct_values" : {
                "object_id" : "team_id"
            }
        }
    }
},
```

child_class_name	The base class name of the child object.
copy_values	A map of from -> to values. When a child object is placed onto the collection, corona copies the elements in the copy_values from the parent to the child.
construct_values	A map of from -> to values. When a child object is created onto the collection, corona copies the elements in the copy_values from the parent to the child.

By default, Corona does *not* return child objects. But if you use the get method, you can include_children to get child objects.



Also, you can always query a child class directly, similar to how you might do it in a relational database.

```
    ],
    "stages" : [
        {
            "class_name" : "filter",
            "name" : "people_result",
            "input" : "sys_grant"
        }
    ]
}
```

y Cookies Headers (4) Test Results

retty Raw Preview Visualize JSON

```
{
    "data": [
        {
            "alter": "any",
            "delete": "any",
            "get": "any",
            "grant_class": "system_content",
            "object_id": 6,
            "put": "any",
            "team_id": 5
        },
```

Showing a bit of the query engine is a nice flow into the next thing.

Query fields

The query engine for the Corona API is the same as the one as the engine used to query the entire database. The one main difference between the two is that a class object will inject itself as **“`this`”** into one of the sources. **`this`** may be referenced downstream to provide for filtering.

To create a query field, you need a corona query. A corona query is structured like this

from	An array of from items. Each item is a corona class. If a base class is provided, all objects of any derived class will also be included.
stages	An array of stages. Stages are steps in a query processing pipeline. They are filter, join, project, and we'll probably have a few more on the way. Each stage can refer to the outputs of other stages as inputs and the final stage's output is the delivered result.

Thus, the query has an array for from, and an array for stages.

```
{
    "from" : [
        {
            "class_name" : "candidate",
            "name" : "candidate",
            "filter" : {
                "candidate_id" : "H0AK00105"
            }
        }
    ],
    "stages" : [
        {
            "class_name" : "filter",
            "input" : "candidate"
        }
    ]
}
```

from object

from object has a class_name, for which is its source of objects in corona, and it has a name, with which later filters and stages may refer to it.

The filter on a from object is simplistic. It can contain only a list of exact keys that match a given value. It's job is to really just allow the use of indexes in certain important cases. In the previous, we can see matching the candidate_id on an exact value. However, we can see how both the implicit **`this`** in the from section coupled with references to it throughout the rest of the query. In this example, the user agent would call /objects/run on this object as the

user made changes to his or her query form of some kind. The query result is just a method on the object, and it looks like this:

```
"class_name": "candidate_search",
"class_description": "search object for candidates",
"base_class_name": "sys_object",
"fields": {
  "full_name": "string",
  "state": "string",
  "city": "string",
  "political_party": "string",
  "candidates": {
    "field_type": "query",
    "query": {
      "from": [
        {
          "class_name": "candidate",
          "name": "candidate"
        }
      ],
      "stages": [
        {
          "class_name": "filter",
          "stage_input_name": "candidate",
          "condition": {
            "class_name": "any",
            "conditions": [
              {
                "class_name": "contains",
                "valuepath": "full_name",
                "value": "$this.full_name"
              },
              {
                "class_name": "eq",
                "valuepath": "state",
                "value": "$this.state"
              },
              {
                "class_name": "contains",
```

Stages are specified as objects of one of these class names: “filter”, “join”, or “project”

filtering stages are used to select rows. A filtering stage is class_name “filter”, takes a single stage input name, and has a condition.

conditions can be one of the following: “contains”, “eq”, “lt”, “gt”, “lte”, “gte”, “any”, “all”.

These conditions all the have same basic template:

“contains”, “eq”, “lt”, “gt”, “lte”, “gte”,

All look like

```
{
                    "class_name": "eq",
                    "valuepath": "state",
                    "value": "$this.state"
                },
```

Example query. Basic format of a from level join.

Joins on from clauses can be way faster.

```
{
    "from" : [
        {
            "class_name" : "candidate",
            "name" : "candidate",
            "filter" : {
                "candidate_id" : "H0NV01219"
            }
        },
        {
            "class_name" : "committee_candidate",
            "name" : "committee_candidate",
            "filter" : {
                "candidate_id" : "$candidate.candidate_id"
            }
        },
        {
            "class_name" : "committee",
            "name" : "committee",
            "filter" : {
                "committee_id" : "$committee_candidate.committee_id"
            }
        }
    ],
    "stages" : [
        {
            "class_name" : "filter",
            "input" : "candidate"
```

```
        }
    ]
}
```

indexes

corona classes may be queried by any field, and so, to speed up some of those queries, indexes may be defined. Indexes are simply specified. First have an index map. Each key is the name of the index, and the value side is an array that lists the fields in order. All indexes in Corona are trailed by an implicit object id. So you can lose some worry about creating an index without adequate keys coverages.

In the below example, for the class user_content, and index is created named object_content. Object_content has the index keys content_object_id. which is a field on the class. Indexes can theoretically have any type of key, but strings, dates, and numbers, particularly ints, seem to the most effective.

```
"class_name": "user_content",

"class_description": "Base of user owned objects",
"base_class_name": "sys_object",
"content_object_id": "int64",
"indexes": {
  "object_content": {
    "index_keys": [
      "content_object_id"
    ]
  }
}
```

Indexes are internally sorted by strict weak ordering. And, the decision to use in an index is scored based on the query. The index whose keys match the query the most picks which index will be used. For SQL backed classes, indexes to facilitate the mapping between Corona's object id and the associated SQL Primary key are created.

An index can be dropped by re-putting the same class without the index.

example classes

user comment class. This is from the candidates example schema, and you can see, it derives from user_content, which is derived from sys_object.

```
{
  "class_name": "comment",
  "class_description": "Comments made by users",
  "base_class_name": "user_content",
  "fields": {
    "comment_text": "string"
  }
},
```

A quick check to see the class via the api shows us this:


```
{
    "data": {
        "definition": {
            "ancestors": [
                "sys_object",
                "user_content"
            ],
            "base_class_name": "user_content",
            "class_description": "Comments made by users",
            "class_name": "comment",
            "descendants": [
                "comment"
            ],
            "fields": {
                "class_name": {
                    "field_name": "class_name",
                    "field_type": "string"
                },
                "comment_text": {
                    "field_name": "comment_text",
                    "field_type": "string"
                },
                "content_object_id": {
                    "field_name": "content_object_id",
                    "field_type": "int64"
                },
                "created": {
                    "field_name": "created",
                    "field_type": "datetime"
                },
                "created_by": {
                    "field_name": "created_by",
                    "field_type": "string"
                },
                "object_id": {
                    "field_name": "object_id",
                    "field_type": "int64"
```

```
            },
            "updated": {
                "field_name": "updated",
                "field_type": "datetime"
            },
            "updated_by": {
                "field_name": "updated_by",
                "field_type": "string"
            }
        },
        "table_fields": [
            "class_name",
            "comment_text",
            "content_object_id",
            "created",
            "created_by",
            "object_id",
            "updated",
            "updated_by"
        ],
        "table_location": 70336
    },
    "info": {
        "comment": {
            "block": "branch",
            "block_content": "leaf",
            "block_count": 0,
            "block_key_start": "",
            "block_location": 68208,
            "children": []
        }
    }
},
"message": "Ok",
"seconds": 0.001040,
"success": 1.000000,
```


class api and modifications

corona's class api allows you to do make modifications at run time, just like how a database server is capable of altering tables. You can create classes, add or remove fields and indexes, and derive new classes from existing classes.

For lot of people, modify classes at run time is not the best practice, and for them, it's good

to know that by default it is switched off for teams, although the super user could do it. But, for some customer intensive applications, the ability to create subclasses in the system at run time allows for quite a bit of self-service that would be impossible in normal closed systems.

Some general rules

Creating a new class just creates a class

Putting a changed class modifies the class you put to, *and* the derived classes.

corona/classes/get	Request

	{
	
	}

	Response

	{
	"data": [
	{
	"ancestors": [
	"sys_object",
	"system_content"
	],
	"base_class_name": "system_content",
	"class_description": "Candidates running for public office",
	"class_name": "candidate",
	"descendants": [
	"candidate"
	],
	"fields": {

	Description
	Returns a decent list of all the classes, and for the moment, a fair few of the details with. The biggest thing that is lacking is some internal layout information that comes with the details, along with whatever metrics we might throw in there in the future.
corona/classes/get/details/	Request

	{
	"class_name" : "comment"
	}

	Response

	{
	"data": {
	"definition": {

<table>
<tr><td></td><td><pre>
"ancestors": [
    "sys_object",
    "user_content"
],
"base_class_name": "user_content",
"class_description": "Comments made by users",
"class_name": "comment",
"descendants": [
    "comment"
],
"fields": {
    "class_name": {
        "field_name": "class_name",
        "field_type": "string"
    },
</pre></td></tr>
<tr><td></td><td>Description</td></tr>
<tr><td></td><td>Returns extended information about a class.</td></tr>
<tr><td>corona/classes/put</td><td>Request</td></tr>
<tr><td></td><td><pre>
{
    "data":{
        "base_class_name": "sys_object",
        "class_description": "Base of user owned objects",
        "class_name": "user_content",
        "fields": {
            "class_name": {
                "field_name": "class_name",
                "field_type": "string"
            },
            "content_object_id": {
                "field_name": "content_object_id",
                "field_type": "int64"
            },
            "user_content_tale_of_woe": {
                "field_name": "user_content_tale_of_woe",
                "field_type": "string"
            },

            "created": {
                "field_name": "created",
                "field_type": "datetime"
            },
            "created_by": {
                "field_name": "created_by",
                "field_type": "string"
            },
            "object_id": {
</pre></td></tr>
</table>

	"field_name": "object_id",
	"field_type": "int64"
	},
	"updated": {
	"field_name": "updated",
	"field_type": "datetime"
	},
	"updated_by": {
	"field_name": "updated_by",
	"field_type": "string"
	}
	},
	"indexes": {
	"object_content": {
	"index_keys": [
	"content_object_id",
	"object_id"
	],
	"index_name": "object_content",
	"table_location": 55336
	}
	}
	}
	}

	Response
	{ "data": [], "message": "Ok", "seconds": 3.767289, "success": 1.000000,
	Description
	Given a class definition, attempts to update it and its descendant classes with new fields and indexes.

There's actually no way to delete a class, once you create it. Also need namespaces, and querying not just off of class relationships, but also, any list of classes.

Datasets

Corona schema files support the concept of datasets for load out. These are versioned in a fashion similar to Liquibase, with the developer intent that reapplying a schema won't cause a dataset to be reapplied if the version is the same.

A DataSet can be one of two things. It can have a list of objects which are put immediately, and it can have a CSV file for import.

objects

Objects are specified in a data set. The objects in a dataset is just an array of objects, with some header information to identify it.

```
{
  "dataset_name": "Candidate Teams",
  "dataset_description": "Load Teams",
  "dataset_version": "1.0",
  "dataset_author": "TJ Bandrowsky",
  "dataset_source": "inline",
  "objects": [
    ...,
  ]
},
```

The objects themselves, are just objects in json, just like they would be put. For example, look at our teams again.

```
    {
      "class_name": "sys_team",
      "team_name": "admins",
      "team_description": "company admins",
      "team_domain": "countryvideogames.com",
      "permissions": [
        {
          "class_name": "sys_grant",
          "grant_class": "system_content",
          "put": "any",
          "get": "any",
          "alter": "any",
          "delete": "any"
        }
      ],
      "workflow_classes": [ "committee_search", "candidate_search", "people_search" ]
    },
```

imports

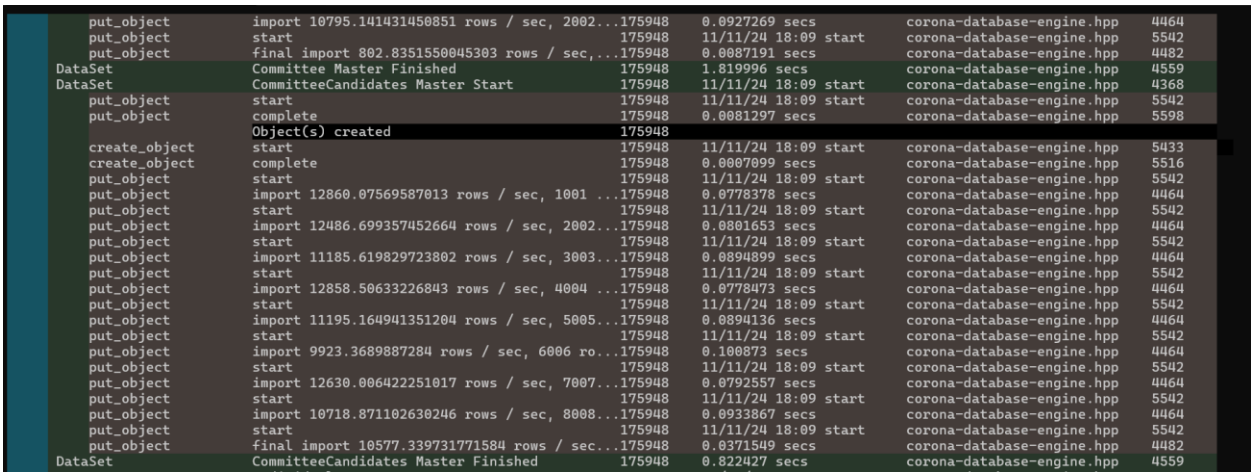
An import data set looks like this. Presently only csv is supported, which is a bare delimited file. The delimiter may be specified. The filename given is the source filename for the import,

and the target_class is where it goes. A column_map maps the field on the right to the column id on the left.

```
    "import": {
        "type": "csv",
        "delimiter": "|",
        "filename": "cn.csv",
        "target_class": "candidate",
        "column_map": {
            "0": "candidate_id",
            "1": "full_name",
            "2": "political_party",
            "3": "election_year",
            "10": "street1",
            "11": "street2",
            "12": "city",
            "13": "state",
            "14": "zip"
        }
    }
```

Corona processes the schema import csvs in sets of 1000.

Issues with malformed objects are reported.



objects

Objects are the data that one puts into Corona. An object is an instance of data specified by its class. Internally, it's a table definition with some added frills.

To create an object, use the create_object api. That gives you an object whose members are not filled out, and all you have to do is drop some data in, and save it back. Or, you can just create an object without an id and Corona will give you one when you put it.

One can examine a single object using get object or edit object. Edit object is like get object, plus it contains details about the class of object itself, so that you can build an editor around it. The other method you can use, is run, so you can make some changes to an object, put, have its onboard queries evaluated, and get an answer back. In this way you can use an object as a functor, whose mission is to feed itself as parameters to its onboard queries. So you edit the object, and then have user modifications, followed by run. Or, you can just call put_object.

Objects can also be copied and deleted. When an object is copied, it may be copied to an instance of an entirely different class, in which case, only matching fields are copied. Deleting an object in Corona is permanent.

Objects can be in queries, or even just listed. Classes are queried for objects. If you try to query a base class, that automatically includes all the descendants.

corona/objects/get	Request

	{
	"class_name" : "candidate",
	"object_id": 32,
	"include_children" : true
	}

	Response

	{
	"data": {
	"candidate_id": "H0AR03030",
	"city": "FAYETTEVILLE",
	"class_name": "candidate",
	"comments": [],
	"committees": [],
	"created": "2024-11-27T15:51:28Z",

<table>
<tr><td></td><td><pre>
        "created_by": "system",
        "election_year": 2010,
        "full_name": "WHITAKER, DAVID JEFFREY",
        "object_id": 32,
        "political_party": "DEM",
        "state": "AR",
        "street1": "PO BOX 957",
        "street2": "",
        "zip": "727020957"
    },
    "message": "Ok",
    "seconds": 0.017062,
    "success": 1.000000,
</pre></td></tr>
<tr><td></td><td>Description</td></tr>
<tr><td></td><td>Returns an object given a class name and object id. If include_children is true, runs all the queries on the class with this object.</td></tr>
<tr><td>corona/objects/put</td><td>Request</td></tr>
<tr><td></td><td><pre>
{
  "data":
  {
   "class_name": "comment",
   "comment_text": "this is my sample comment text.",
    "content_object_id": 15,
    "user_content_tale_of_woe": "another comment text"
  }
}
</pre></td></tr>
<tr><td></td><td>Response</td></tr>
<tr><td></td><td><pre>
{
    "data": {
        "comment": [
            {
                "data": {
                    "class_name": "comment",
                    "comment_text": "this is my sample comment
text.",
                    "content_object_id": 15,
                    "created": "2024-11-27T21:33:30Z",
                    "created_by": "system",
                    "object_id": 37841,
                    "user_content_tale_of_woe": "another comment
text"
                },
                "message": "Ok",
</pre></td></tr>
</table>

<table>
<tr><td></td><td><pre>
                "success": 1.000000
            }
        ]
    },
    "message": "Object(s) created",
    "seconds": 0.035820,
    "success": 1.000000
}
</pre></td></tr>
<tr><td></td><td>Description</td></tr>
<tr><td></td><td>Returns a structure indicating the disposition of the object. You can send an array to put multiple objects.</td></tr>
<tr><td>corona/objects/query</td><td>Request</td></tr>
<tr><td></td><td><pre>
{
    "from" : [
        {
            "class_name" : "user_content",
            "name" : "user_stuff",
            "filter" : {}
        }
    ],
    "stages" : [
        {
            "class_name" : "filter",
            "name" : "user_stuff",
            "input" : "user_stuff"
        }
    ]
}
</pre></td></tr>
<tr><td></td><td>Response</td></tr>
<tr><td></td><td><pre>
{
    "data": [
        {
            "class_name": "link",
            "content_object_id": 0,
            "link_url": "http://www.microsoft.coo",
            "object_id": 37840,
            "updated": "2024-11-28T10:58:37Z",
            "updated_by": "system"
        },
        {
            "class_name": "comment",
            "comment_text": "test comment",
            "content_object_id": 0,
            "object_id": 37841,
            "updated": "2024-11-28T10:58:29Z",
</pre></td></tr>
</table>

	"updated_by": "system" }], "message": "completed", "seconds": 0.002195, "success": 1.000000, "token":
	Description
	Queries one or more classes for objects. If a base class is queried, itself and its descendant classes are queried.

Postman Walkthrough

Corona comes with a postman collection that is used to test these queries, develop Corona, and allow people to play with it.

Postman Name	Description
Test	Returns an echo to see if the server is up
Login	Logins in as the system user
User-Create Company User User-Confirm Company User User-Create Customer User User-Send Customer Confirm User-Change Customer Password User – Login Company User – Login Customer	The requests walk through the sign in system. Company users and customer users can both self-serve, and each is assigned to teams based on the sign-ons verified email domain.
Get Classes Get Class Detail Put Class	These do some checks on the class api
Query Using Index	Used to test if queries are hitting indexes
Query using Joins	Walks through joins in the query from
System Teams	Take a look at system teams
System Grants	Take a look at system grants
Query SQL	Query mechanism using a Corona class with a SQL Server implementation.
Query – All condition	Query to check the all condition, which lets you say all of these conditions are true
Query – Any condition	Query any condition, which lets you say if any single condition is true
Query – eq Condition	Query eq condition, and others
Query – contains condition	Query contains, search for text contains
Get Objects	Fetch objects from Corona
Get Objects Team	Get a team
Get Objects SQL	Does it work with SQL
Get Objects 2	Another look at getting objects
Create Object Sibling1 Put Object Sibling1 Create Object Sibling2 Put Object Sibling2	These are to test if two objects of a related base will be queryable from the base.

put Object put Object partial	Used to ensure that a partial put doesn't blow away all the other fields.