



Creating your first Domain and API

Overview & Purpose

Jargon makes it easy to break down large monolithic models into smaller and more manageable domains - which you will be doing a lot. This training session will cover everything you need to start creating Domains and generating APIs from them.

The Domain we will be creating is a simple Pet Store that has Pets, Orders, Users, etc.

Objectives

1. Create and modify the sample Pet Store Domain
2. Import and use the AS4590 Address
3. Create a Code Table, including from a State Lifecycle
4. Create several API paths
5. Generate an API specification
6. Release the Domain

Prerequisites

All you will need is a Jargon.sh free user account

Length

About an hour

Activity

Create the sample Pet Store domain

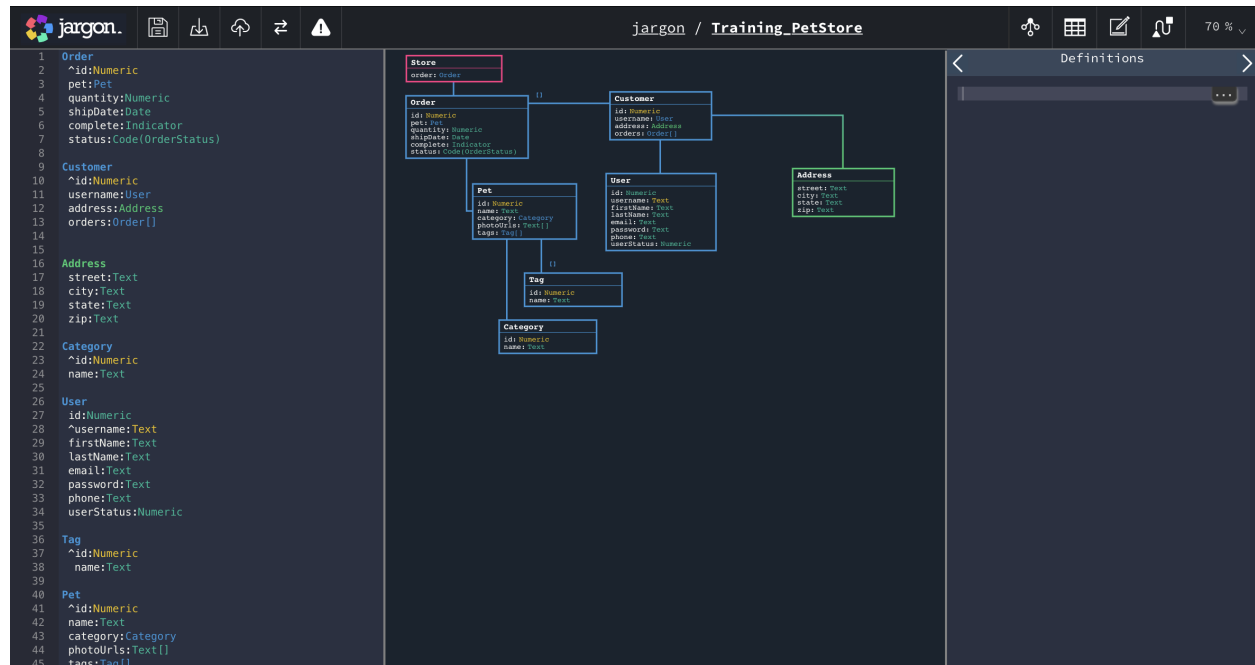
1. Create a new domain, called '**Training_PetStore**'
 - a. Description:
 - i. '**A training domain that demonstrates how to create and modify domains and releases**'
 - b. If you're a member of a team, or a paying user, you can make the domain private
 - c. Click the '**Create domain**' button

The new Domain will be created, and you will be redirected to the details page for the Domain.

2. Click the pink button '**Open editor**' to open the new domain in editor
 - a. As this is a new Domain, Jargon will offer to fill it with an example domain - which we will do, but not just yet.
 - b. Open the '**Getting Started guide**' documentation link in a new tab - everything covered in this lesson is described in more detail that you can refer to
 - c. Click the '**An example would be nice!**' button to create the sample domain that we will be using throughout this lesson.

A new domain will be created that has eight Classes - Store, Order, Pet, Tag, Category, Customer, User and Address.

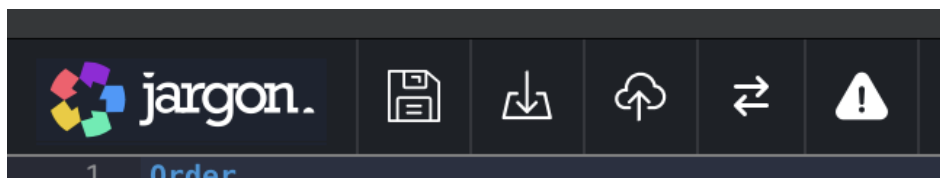
It should look like this:



Import the AS4590 Domain

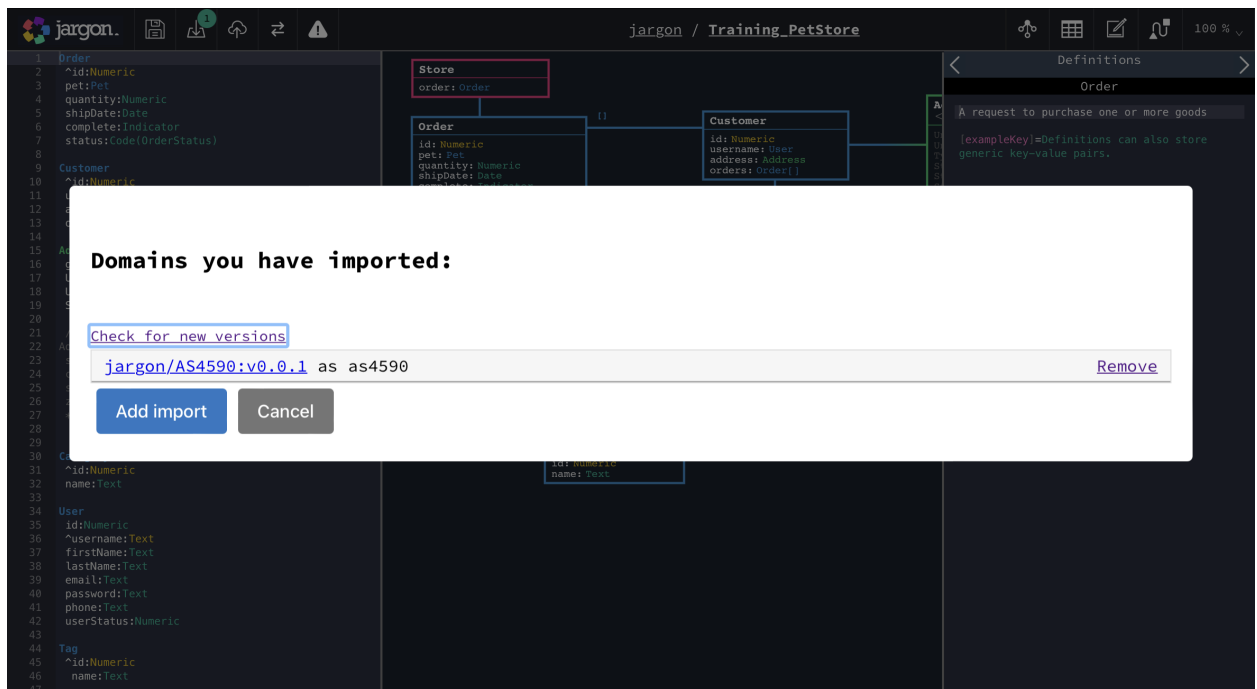
Instead of locally declaring an Address in this domain - which isn't really right as this Domain talks about Pets, not Addresses. We're going to be reusing an Address from another domain.

1. First we'll go over what the menu buttons are.



2. From left to right they are: Save, Import (the one we're about to use), Upload, Export, and Errors.
3. Clicking the Import button will bring up a search box where we can search for a Domain. Searching looks in a Domains description, release notes, and the names of its Classes. The results are ordered based on their relevance, and the automatic quality scores that Jargon assigns to domains.

4. If you're not using the public jargon.sh instance, you'll also need to tick the 'search public jargon' checkbox.
5. Type '**Address**' into the search box, and you will see several results - each one is a domain that you could potentially import and use. We'll be using the jargon/AS4590 result.
6. You will need to give this domain an alias - like a nickname to use throughout your domain - short and unique aliases are best, so let's use '**as4590**'
7. Clicking on the Import button will now show you the domains you have imported, and their aliases - it should look like this:



Use the Address from AS4590

Now that we've imported the AS4590 Domain, we can reuse parts from it, and we'll cover three different approaches you can take.

Using properties from an imported Class

The local Address Class has four properties - each with pretty poor quality definitions, so let's use the definitions from as4590 instead,

1. Change the type of Address.street from Text to **'as4590.Address.StreetName'**
 - a. Notice that as you type, Jargon will offer you suggestions.
 - b. Notice also, that the definition screen has changed. The definition in the bottom half is from the import, and the top half is the locally declared definition. We want to use the imported definition, so delete the local one.
2. Change Address.city to **'as4590.Address.LocalityName'**, Address.state to **'as4590.Address.StateTerritory'** and Address.zip to **'as4590.Address.Postcode'** and delete each of their local definitions.

Now each of the Address properties is using an imported definition from AS4590, and should look like this:

The screenshot shows the Jargon IDE interface for a project named 'Training_PetStore'. The left pane displays a code editor with the following code:

```

1 Order
2   ^Id:Numeric
3   pet:Pet
4   quantity:Numeric
5   shipDate:Date
6   complete:Indicator
7   status:Code(OrderStatus)
8
9 Customer
10  ^Id:Numeric
11  username:User
12  address:Address
13  orders:Order[]
14
15 Address
16  street:as4590.Address.StreetName
17  city:as4590.Address.LocalityName
18  state:as4590.Address.StateTerritory
19  zip:as4590.Address.Postcode
20
21
22 Category
23  ^Id:Numeric
24  name:Text
25
26 User
27  id:Numeric
28  ^username:Text
29  firstName:Text
30  lastName:Text
31  email:Text
32  password:Text
33  phone:Text
34  userStatus:Numeric
35
36 Tag
37  ^Id:Numeric
38  name:Text
39
40 Pet
41  ^Id:Numeric
42  name:Text
43  category:Category
44  photoUrls:Text[]
45  tags:Tag[]
  
```

The middle pane shows a class diagram with the following classes and relationships:

- Store** (class) has a property **order:Order**.
- Order** (class) has properties **id:Numeric**, **pet:Pet**, **quantity:Numeric**, **shipDate:Date**, **complete:Indicator**, and **status:Code(OrderStatus)**.
- Customer** (class) has properties **id:Numeric**, **username:User**, **address:Address**, and **orders:Order[]**.
- Pet** (class) has properties **id:Numeric**, **name:Text**, **category:Category**, **photoUrls:Text[]**, and **tags:Tag[]**.
- User** (class) has properties **id:Numeric**, **username:Text**, **firstName:Text**, **lastName:Text**, **email:Text**, **password:Text**, **phone:Text**, and **userStatus:Numeric**.
- Address** (class) has properties **street:as4590.Address.StreetName**, **city:as4590.Address.LocalityName**, **state:as4590.Address.StateTerritory**, and **zip:as4590.Address.Postcode**.
- Category** (class) has properties **id:Numeric** and **name:Text**.
- Tag** (class) has properties **id:Numeric** and **name:Text**.

The right pane shows the 'Definitions' panel for the **Address.zip** property. It displays the definition from import:

```

Definition from import
The Australian numeric descriptor for a
postal delivery area, aligned with locality,
suburb or place
  
```

Using the entire AS4590 Address class

Instead of using just the properties from Address, we may want to use the entire Class

1. Comment out the entire Address Class, by placing `/*` on the line above Address and `*/` on the line after zip:...
2. Notice that there is now an Error, shown by the 1 in the notification bubble on the Error button. If you click the Error button, Jargon will tell you what the error is, and where it is coming from.
3. The error is caused because the Customer class was using the Address class, which we removed from the model by commenting it out.

We want to use the Address version from AS4590, and not the one that was previously in our model.

4. There are two ways to do that
 - a. Because we already know there is an Address class, we can change Customer.address to '**as4590.Address**'
 - b. Jargon can automatically resolve unknown classes from classes in your imports. Clicking the Import Tree button shows you all of the imports, their classes and all their properties. At the top of the Imports Tree pane is a link called '**Resolve unknown Classes**'. Clicking that will automatically resolve the unknown Address reference to as4590.Address

When you're done, it will look like this:

The screenshot shows the Jargon IDE interface for a project named 'Training_PetStore'. The code editor on the left contains the following code:

```

1 Order
2   ^id:Numeric
3   pet:Pet
4   quantity:Numeric
5   shipDate:Date
6   complete:Indicator
7   status:Code(OrderStatus)
8
9 Customer
10  ^id:Numeric
11  username:User
12  address:as4590.Address
13  orders:Order[]
14
15 /*
16 Address
17 street:as4590.Address.StreetName
18 city:as4590.Address.LocalityName
19 state:as4590.Address.StateTerritory
20 zip:as4590.Address.Postcode
21 */
22
23
24 Category
25 ^id:Numeric
26 name:Text
27
28 User
29 id:Numeric
30 ^username:Text
31 firstName:Text
32 lastName:Text
33 email:Text
34 password:Text
35 phone:Text
36 userStatus:Numeric
37
38 Tag
39 ^id:Numeric
40 name:Text
41
42 Pet
43 ^id:Numeric
44 name:Text
45 category:Category
46 photoUrls:Text[]
  
```

The class diagram in the center shows the following classes and their relationships:

- Store** (highlighted with a red box) has a property `order: Order`.
- Order** has properties `id: Numeric`, `pet: Pet`, `quantity: Numeric`, `shipDate: Date`, `complete: Indicator`, and `status: Code(OrderStatus)`.
- Customer** has properties `id: Numeric`, `username: User`, `address: as4590.Address`, and `orders: Order[]`.
- Pet** has properties `id: Numeric`, `name: Text`, `category: Category`, and `photoUrls: Text[]`.
- User** has properties `id: Numeric`, `username: Text`, `firstName: Text`, `lastName: Text`, `email: Text`, `password: Text`, `phone: Text`, and `userStatus: Numeric`.
- Tag** has properties `id: Numeric` and `name: Text`.
- Category** has properties `id: Numeric` and `name: Text`.

The definitions pane on the right shows the definition for `Customer.address`:

```

The shipping address for the customer where
they would like their orders delivered

[exampleKey]=Definitions can also store
generic key-value pairs.

Definition from import
  
```

Customising the AS4590 Address class by SubClassing

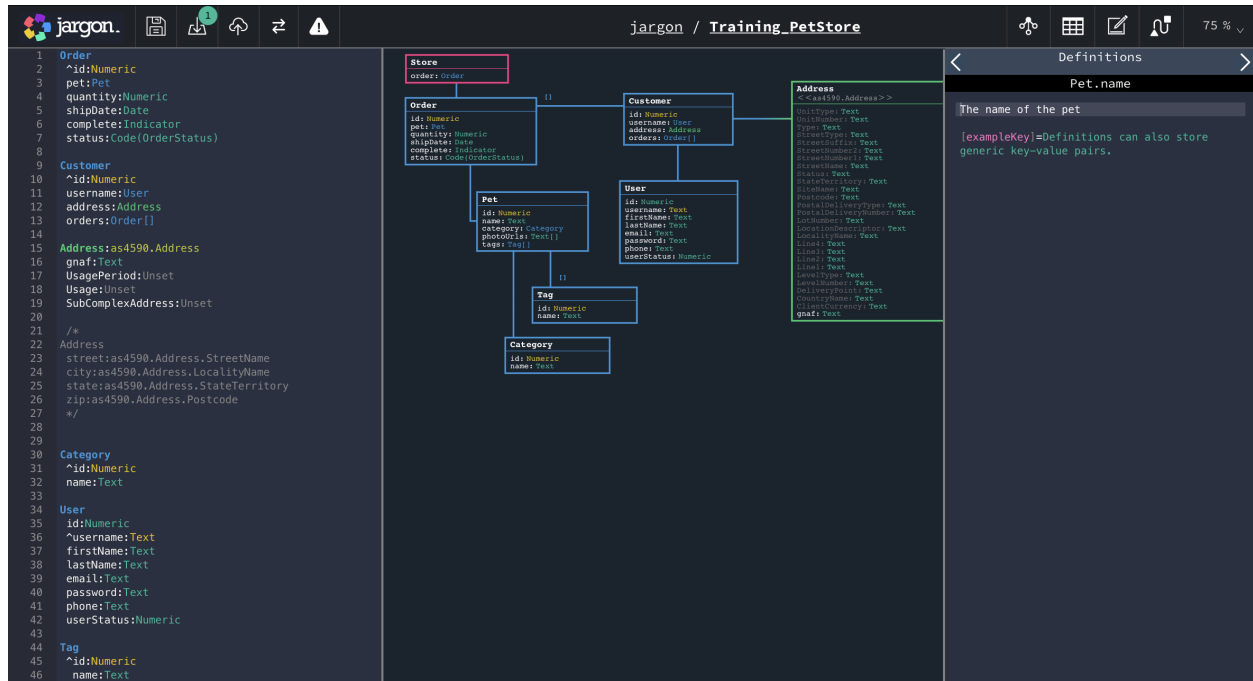
Instead of using the entire Address from AS4590, we might want to add some new properties to it, or remove some of the existing ones.

1. Reset the Customer.address property to Address - which will create an error as there's no Address class - but we'll fix that
2. Create a new Class, called '**Address:as4590Address**' - which will base Address on the imported AS4590 Address Class
3. Notice that the diagram has been updated to show the full as4590 Address - which is quite big.

Now our Customer Address contains all the inherited properties from AS4590 - but it's still a regular Class, and we can add properties to it just like any other Class.

4. Let's add a new property for the Geocoded National Address File, or gnaf, to this address. Create a new property '**gnaf:Text**' and update it's definition to '**The GNAF identifier for this address**'
5. Notice that gnaf is added to the diagram, and that it's white while all the inherited properties are grey.
6. We can remove some of the inherited properties by changing their type to '**Unset**'. Remove UsagePeriod, Usage and SubComplexAddress by changing all their types to '**Unset**'.
7. Notice that they are all removed from the diagram - they no longer exist in your model

When done, it will look like this:

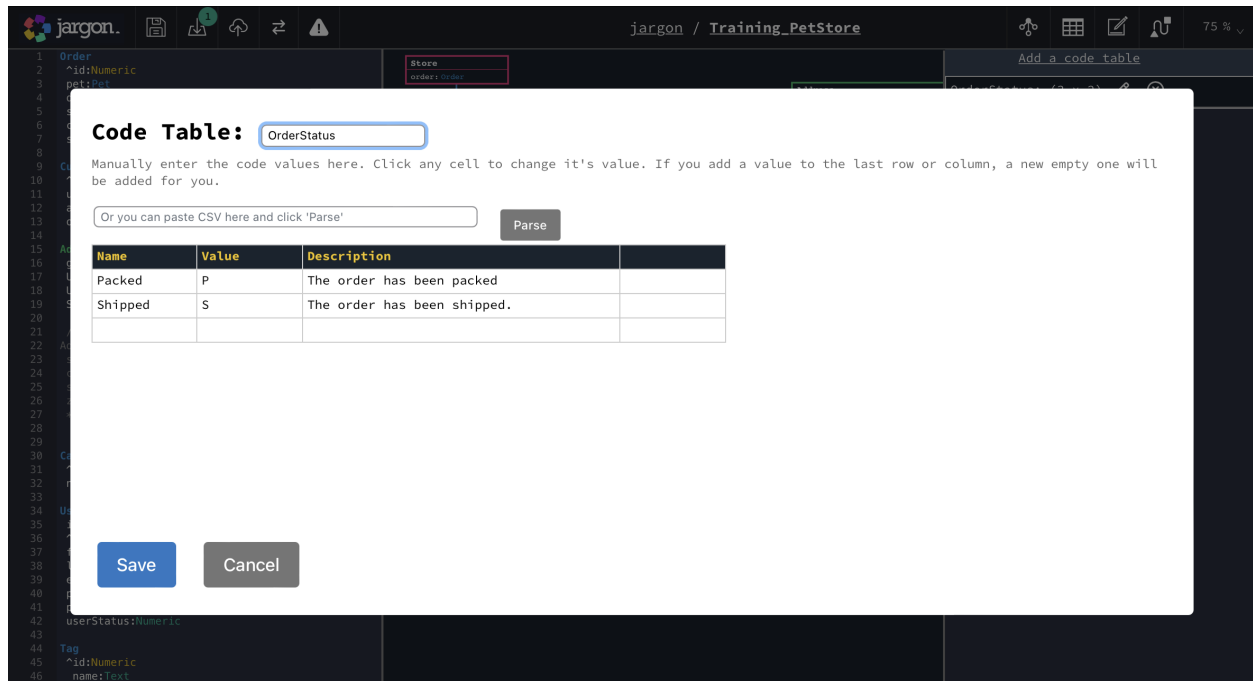


Add an Order status code

Customers would like to know where their orders are up to, so let's add a status code to Orders.

1. Click the Code Table button, and then add a new Code Table called Order Status. The other code types will be explained in later training sessions.
2. Edit the OrderStatus code and add the following rows. Split up the rows into columns using the below commas.
 - a. Packed, P, The order has been packed
 - b. Shipped, S, The order has shipped
3. Save the code table

Instead of manually creating the code values, you can also import them as CSV values by pasting the CSV text into the above box and clicking 'Parse'



4. Create a status property on Order: **'status:Code(OrderStatus)'**
5. When this code is used in an API, Jargon will default to using the first column of a code, but you can override that by adding a key-value attribute to the Order.status definition.
6. Add the following definition to Order.status:

The status of the Order

[oas.codeColumn]=Value

7. This will use the Value column, instead of the first column.
8. There are a variety of these key-value pairs that Jargon knows and understands. Each one will modify how Jargon works. A full list is available in the Jargon documentation -

https://docs.jargon.sh/#/pages/data_definitions?id=key-value-pairs

Take a snapshot of the work we've just done

Snapshots are a point-in-time view of your Domain, and are helpful to share your progress with others. Giving your snapshots descriptive names will also help you when it's time to write up your release notes.

1. Click the save icon, and enter a description of the changes you've made - **'Created rough model'**. Then click **'Save a snapshot'**
2. You can also save your changes without creating a snapshot, if you haven't made any significant changes worth mentioning.

Traverse your model to create API paths

APIs in Jargon are created by choosing a starting point in your model, and walking through its properties.

1. Click the Paths button, and add an API path group, starting at Pet.
2. Add a path to the group, and leave all the defaults, but click **GET**, **POST**, and **DELETE** as the actions to perform on this path.

The screenshot shows the Jargon API editor interface. The main window displays the configuration for a new API path. The 'Target' is set to 'Pet'. Below this, there is a section for 'Manual overrides' with 'Request override' and 'Response override' both set to 'None'. A checkbox labeled 'This filter acts on the collection, not on individuals' is unchecked. The 'Does this Filter perform an operation?' section has a text input field containing 'findByStatus, uploadPhoto, etc'. The 'What Actions can be taken on this Path?' section lists several actions with checkboxes: 'GET Consumers can retrieve the object' (checked), 'POST Consumers can insert a new object. Note: This will operate on the collection, and not an individual.' (checked), 'PUT Replaces the object with a new one' (unchecked), 'DELETE Removes the object' (checked), 'PATCH Updates part of the object' (unchecked), and 'OPTIONS Queries what actions this object supports' (unchecked). At the bottom, there are 'Save' and 'Cancel' buttons.

3. **Save** the Path.
4. Let's add some API paths to manage a Pet's Tags. Click the **'Add a path to this group'** link, and have the Path travers from Pet, through it's **'tags'** property by entering **'tags'**
5. Notice that the Target of the path is now **'Tag[]'** - indicating that Jargon has understood your path and changed the target accordingly.

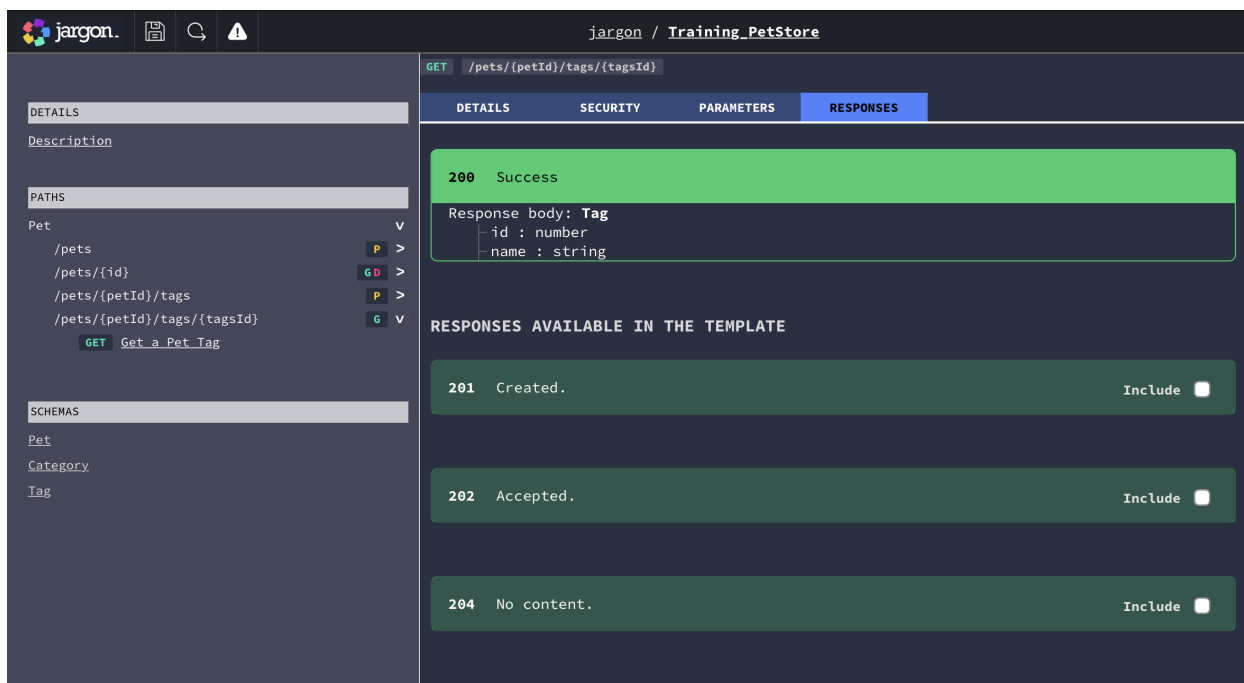
6. Leave all the defaults, but click **GET**, **POST**, and **DELETE** as the actions to perform on this path.
7. **Save** the Path, then save the Domain.

Create an OpenAPI specification

Jargon lets you create different artefacts to further explain and add value to your Domains, one of which is an OpenAPI specification.

1. Navigate back to the Domain Details page, and click the Artefacts tab.
2. Click the '**Create an OpenAPI Specification**' button, which will open the API editor in a new tab.
3. Jargon bases API specifications on templates. There are two default templates: empty, and the National API Design Standards - choose the '**NAPIDS - OpenAPI**' template.
4. The main page shows the details for the API, which are written in Markdown.
5. Jargon only includes the Classes from your model that were used in the paths: **Pet**, **Category** and **Tag** - and you can inspect them in the lower right of the window.
6. Higher up on the left-hand side are the API paths that you designed in the model editor. Expanding the Pet group shows four specific paths, and the HTTP verbs that operate on each. For the paths that operate on individual Pets, Jargon has inserted the Pet Class identifier '**id**' into the path.
7. For paths that target multiple individuals - such as updating a Tag for a Pet - Jargon also includes the identifiers for both. Note that Jargon has changed the path parameters from '**id**' to '**petId**' and '**tagId**' to make it clear which id goes where.
8. Paths that operate on a collection, not individuals, don't have identifiers for those Classes.
9. Each path has a variety of descriptions, and options to include from the template, such as parameters, headers and response bodies. Each of these will be covered in additional training modules

10. For now, all the default settings that Jargon has chosen are good enough, so save the editor and create a snapshot '**Added rough API spec**'



View the generated OpenAPI specification

The generated OpenAPI specification is available on the Artefacts tab on the Domain Details page. Jargon makes it available to you in four different ways:

1. The '**API Docs**' link shows you a human rendering of the specification, allowing you to click through all the paths and see what their description and requirements are.
2. Clicking the '**JSON**' link opens a modal dialog that you can copy and paste the raw specification into, such as additional API tooling.

3. At the bottom of the Artefacts tab, are links to all your raw artefacts so that you can download them and use them elsewhere.
4. Jargon also supports Webhooks - covered in additional training material, and can push the contents of your release to a downstream system, like a GitHub actions workflow.

jargon. Explore Community Documentation Pricing

Search domains Search Sign Out

jargon / Training_PetStore Bleeding-edge

Open Settings Private

A training domain that demonstrated how to create and modify domains and releases

Domain 0 Issues 0 Snapshots 1 Imports 0 Uses 1 Releases **Artefacts**

Use these artefacts for a better understanding of this Domain

OpenAPI / Swagger

Specifications describe the technical specifics required to create or consume an API of this Domain

Open the Editor API Docs | JSON

Specifications page

A Specification is a webpage that contains everything somebody would need to know before using your Domain, or any of it's artefacts

Classes:

Order Customer Address Category User Tag Pet Store

API Paths:

/pets /pets/{id}

Filters: 1 Definitions: 33 Codes: 1

Create a Release so others can import your Domain

Now that you have a rough domain model, and an associated rough API specification, it's time to create a release.

Releases are immutable, meaning that their content can't change - this may seem like a limitation, but it's an important part of allowing others to import and use your work without worrying about it changing.

1. On the Domain Details page, click the Releases tab, then the **'New Release'** button. If you had other releases, they would be shown here.
2. Jargon will gather all the required information for your release, and suggest a Semantic Version number for your release. As this is the first release, Jargon recommends **v0.0.1**, but depending on what breaking or other changes you make in your next release, Jargon will recommend the next appropriate Semantic Version release number for you.
3. You can review all the changes that you have made, before scrolling to the very bottom to create release notes.
4. Jargon groups together all your snapshot descriptions, giving you an idea of the work done since the last release. Copy and paste the text into the bottom of the release notes text-area, and update the other text to suit. **You won't be able to create a release until you've changed the release notes.**
5. Click the **'Create Release'** button, and you're all done!

Write

Preview

Training PetStore

About The Domain

A Training Domain to demonstrate to to use Jargon

Roadmap


Some checkboxes to show what you're planning, and what you've done so far

- ☒ Model of reference data types
- ☒ GET API paths
- ☐ Documentation
 - ☐ API Description
 - ☐ Specification

Change Log

Copy and paste anything interesting in the 'Release Notes' box above, and paste it at the top of this list eg:

- v0.0.1
 - Created rough model
 - Added rough API spec



You won be able to change any of this informaiton later, you'll need to create a new release to correct anything.

Create release

Wrapping up

Congratulations, you've successfully created your first Domain and its associated API design.

The Domain you have just created can now also be imported into other Domains, so they can reuse the work you have just done.

Each time you finish adding enough value that might be relevant to others, making a new Release will let them access it.

