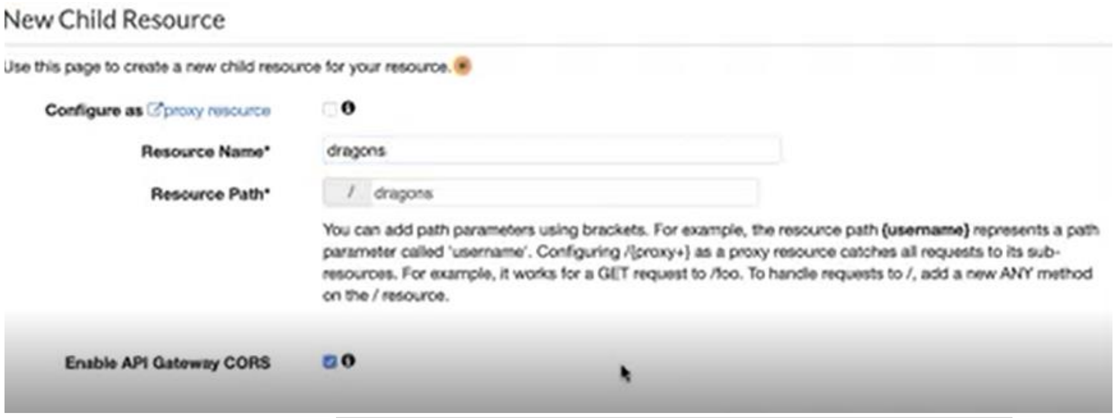
API Driven Development is a process that allows developers to focus on API design before writing code. The idea is, before you start writing your software, you first design and agree on what the API will be. You focus time up front putting thoughtful design into this API, and the first artifact out of the process is the API design. This API design could be captured in a document, or it could be built out as a Mock API. Creating the API before you create the code that backs the API provides benefits to both the developers creating the backend code, as well as the clients who will be consuming the API.
Both the backend API Code and Client code can be written in parallel once the API design is created

Stateful (Web-Socket) and stateless (HTTP and REST) APIs

Resource: is an abstract concept that allows you to expose thing to be consumed by a client.
Create resource:


New Child Resource

Use this page to create a new child resource for your resource.

Configure as proxy resource ☐

Resource Name* dragons

Resource Path* / dragons

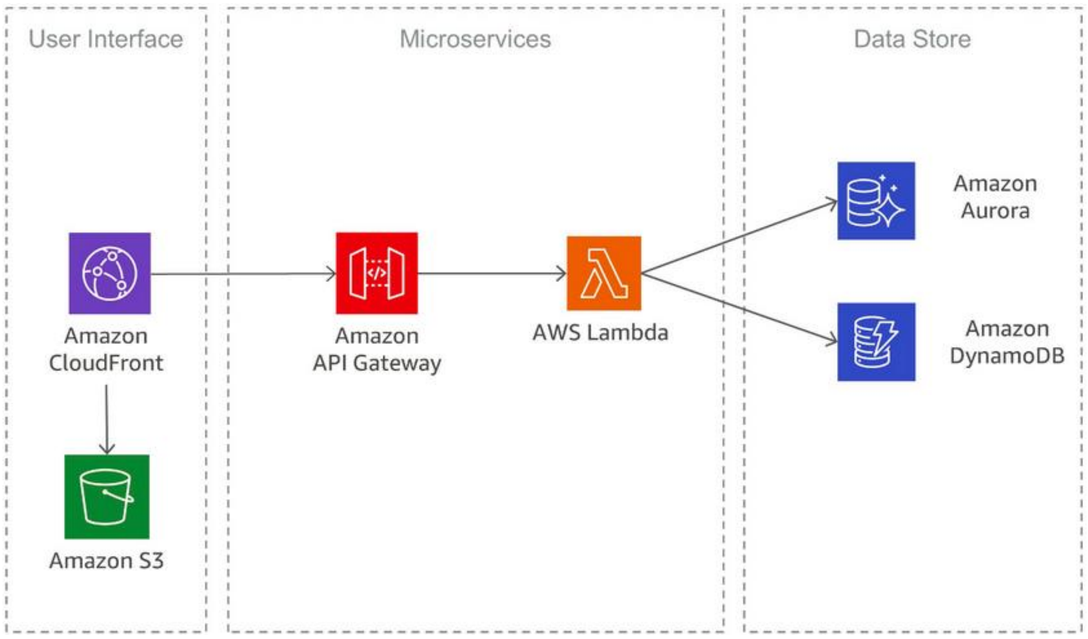
You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username'. Configuring /{proxy+} as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to /foo. To handle requests to /, add a new ANY method on the / resource.

Enable API Gateway CORS ☑

Enable API Gateway CORS: this will enable Cross-Origin Resource Sharing for this API.

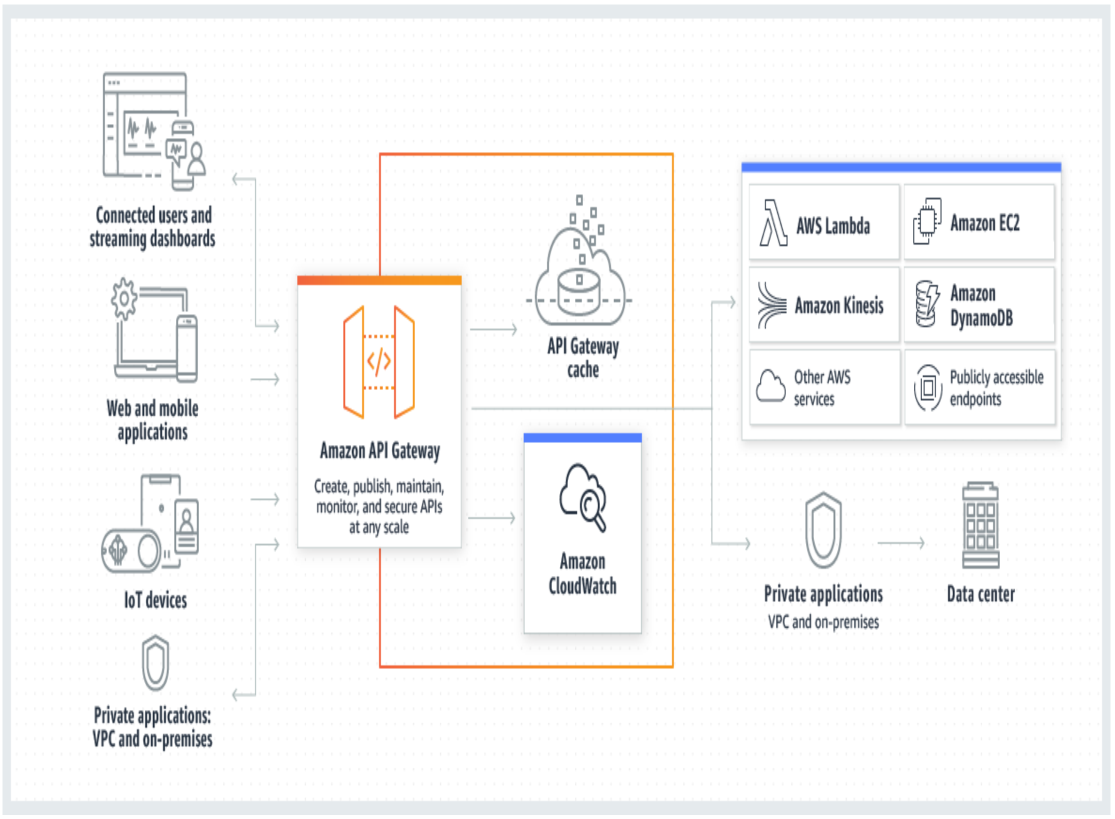
Mock integration type allows u to create API without using other service for the backend, so we won’t have backend.

Server-less micro-service using AWS Lambda:
Synchronous calls from API Gateway to Lambda



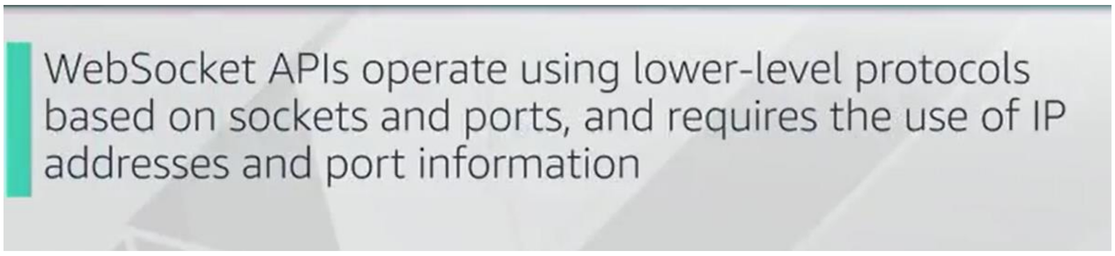
Note: You can use SAM Local to create a local testing environment that simulates the AWS runtime environment.

Architecture of API Gateway


Connected users and streaming dashboards
Web and mobile applications
IoT devices
Private applications: VPC and on-premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private applications
VPC and on-premises
Data center

Amazon cloud watch is for saving the logs: for example logs of http response.
This diagram illustrates how the APIs you build in Amazon API Gateway provide you or your developer customers with an integrated and consistent developer experience for building AWS server-less applications.
API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls.
These tasks include traffic management, authorization and access control, monitoring, and API version management.
API Gateway acts as a "**front door**" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, any web application, or real-time communication applications.
Integration with AWS WAF for protecting your APIs against common web exploits.
Integration with AWS X-Ray for understanding and triaging performance latencies.

Together with AWS Lambda, API Gateway forms the app-facing part of the AWS serverless infrastructure.
AWS API Gateway can create HTTP, Rest and Web-socket API.
Web-socket API is bidirectional, stateful, vertically scalable,
Horizontal scaling means scaling by adding more machines to your pool of resources (also described as "scaling out"), whereas vertical scaling refers to **scaling by adding more power** (e.g. CPU, RAM) to an existing machine (also described as "scaling up").

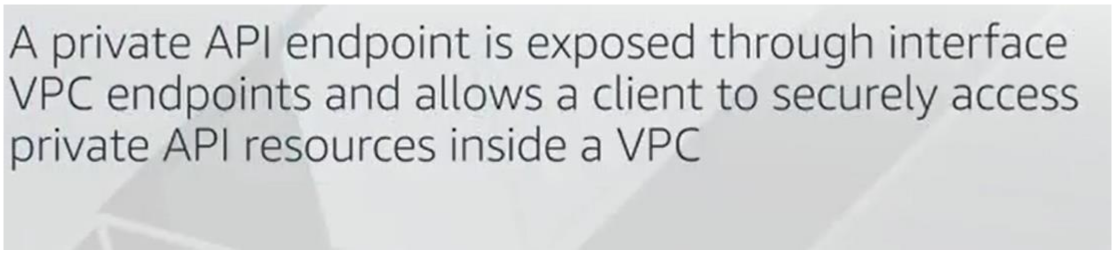

WebSocket APIs operate using lower-level protocols based on sockets and ports, and requires the use of IP addresses and port information

HTTP API enable u to create lower level rest APIs with lower latency and lower costs than REST APIs. We can use this to send the request to AWS Lambda Function or to any routable HTTP end-point.
When the client calls the API, the API Gateway send the request to Lambda function, and returns the functions response to client.

Rest API: fast, stateless, standard, horizontally scalable and dependable.
Other features of API Gateway:
Canary deployment for safety, rolling out changes, logging. Canary release deployments for safely rolling out changes.
AWS Cloud-Tail integration, CloudTrail logging and monitoring of API usage and API changes.
AWS Cloud-Watch integration, CloudWatch access logging and execution logging, including the ability to set alarms.

Different types of endpoint(hostname of API): edge optimized, regional or private.
Edge Optimized Endpoints are best for geographically distributed clients, Requests are routed to nearest cloud-front point of presence: this is default endpoint type of AWS Gateway API.
Regional API endpoint is intended for clients in the same region which reduces the connection overhead.

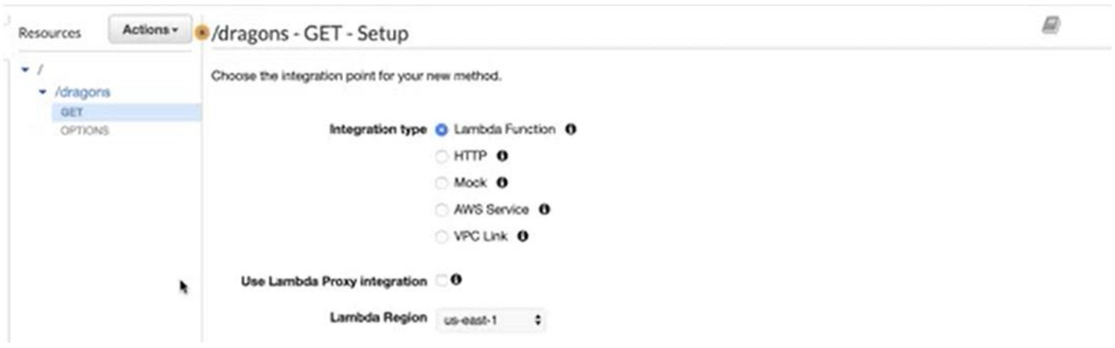
Private API Gateway:



A private API endpoint is exposed through interface VPC endpoints and allows a client to securely access private API resources inside a VPC

Private APIs are isolated from the public internet, and they are only accessible by VPC endpoints for API Gateway that have been granted access.

In AWS GET method as below:



/dragons - GET - Setup

Choose the integration point for your new method.

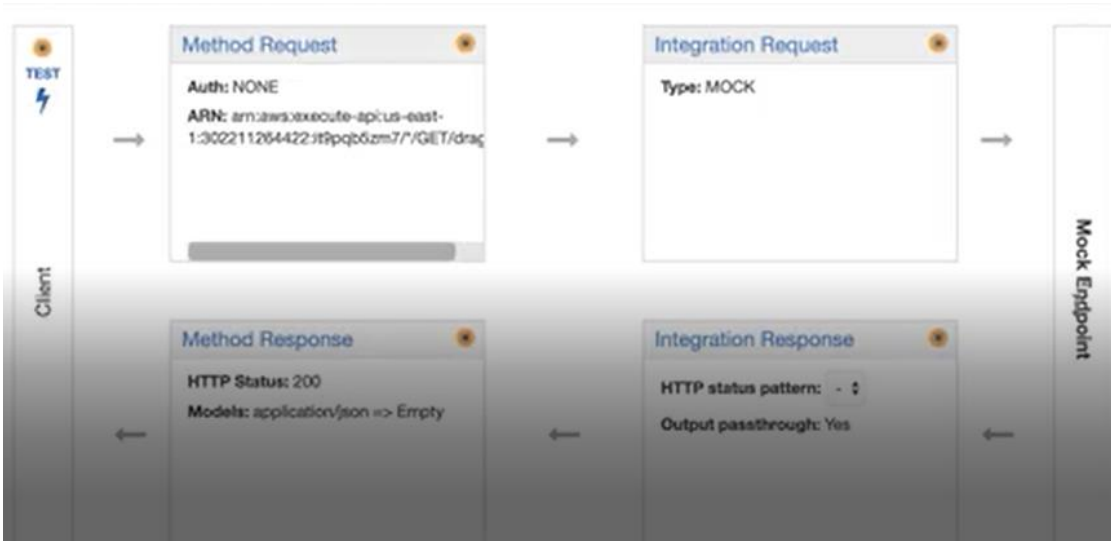
Integration type: [✓ Lambda Function] HTTP Mock AWS Service VPC Link

Use Lambda Proxy integration

Lambda Region: us-east-1

Integration type:
Lambda function: AWS API Gateway uses the pure proxy to Lambda Function.
Mock: we don't want anything real and won't use the services.

Flow of the request and response with API Gateway:



Auth: NONE
ARN: arn:aws:execute-api:us-east-1:302211264422:it9pqb6zm7/*/GET/drag

Type: MOCK

HTTP Status: 200
Models: application/json => Empty

HTTP status pattern: -
Output passthrough: Yes

First we have method request for API Gateway to accept the request, here we can apply authorization and data payload validation,
Next is integration request, which we configure what back-end service, any sort of data transformation, here is mock.
Once the request passed the validation and authorization and the transformation steps, it will send the req to beck-end, here our case is mock endpoint, do nothing and send the response,
Integration response: is http response encapsulating the back-end response, we can configure how our backend service responses map to http response and apply data transformation as well,
Finally method response: similar to method req, they do data validation (to response not client) and give the response to models,

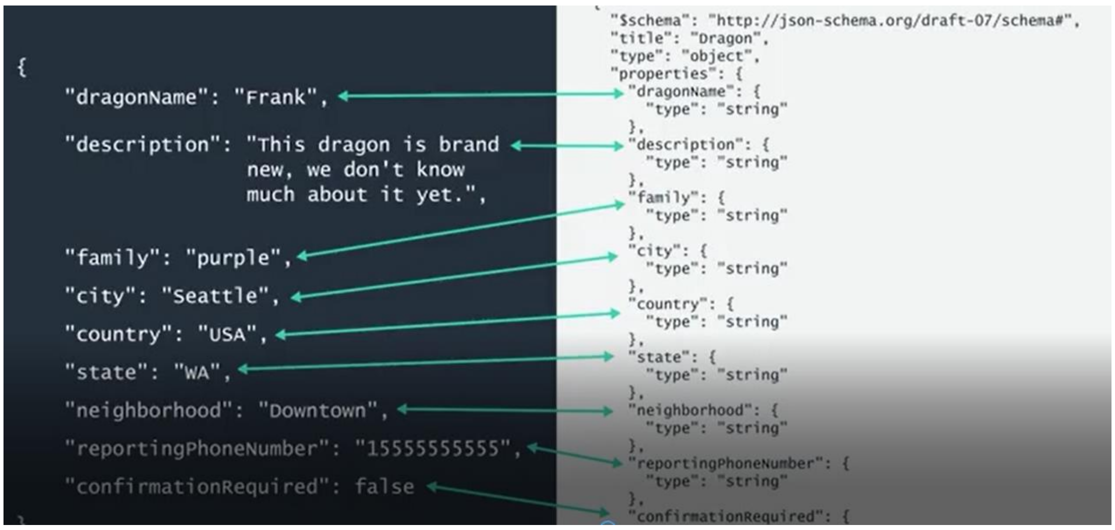
---

In API Gateway, data validation and transformation happen, i.e. check data is not null, the data type is as we expect, also data transformation (reshaping, mapping), which makes our code won't worry about these:

- Validate incoming requests/responses against models
- Transform requests/responses with mappings.

Note: if we only use the API Gateway as the proxy to the back-end, HTTP Request is enough but if we need data validation and transformation in API Gateway level, we need to use REST API.

Models: in API Gateway define the structure or the shape if request/response. Models are created using the JSON schemas (properties and type of payload), if request data is not based on JSON schema, it will return 400 error code.



```
{
    "dragonName": "Frank",
    "description": "This dragon is brand
                    new, we don't know
                    much about it yet.",
    "family": "purple",
    "city": "Seattle",
    "country": "USA",
    "state": "WA",
    "neighborhood": "Downtown",
    "reportingPhoneNumber": "15555555555",
    "confirmationRequired": false
```

```
"$schema": "http://json-schema.org/draft-07/schema#",
"title": "Dragon",
"type": "object",
"properties": {
  "dragonName": {
    "type": "string"
  },
  "description": {
    "type": "string"
  },
  "family": {
    "type": "string"
  },
  "city": {
    "type": "string"
  },
  "country": {
    "type": "string"
  },
  "state": {
    "type": "string"
  },
  "neighborhood": {
    "type": "string"
  },
  "reportingPhoneNumber": {
    "type": "string"
  },
  "confirmationRequired": {
```

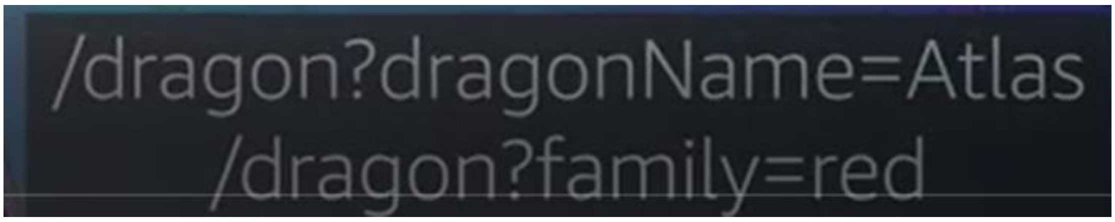
Where left part is JSON payload, but the left is JSON schema.

Each method in resource: get, post, etc., could have different models, so these get applied method level not resource level.

Mappings: instead of changing the code to support different types of data structures, API Gateway does this. Mapping applied to integration requests/responses> they are used for REST HTTP Integrations not Proxy Integrations. Also this like models are also applied to method levels.
Mappings are written in Velocity Template Language (VTL).
If we have defined the model, API Gateway can generate the VTL blueprint for the method, which we can modify.
Mapping Template assume that data coming as a JSON object by default, also do support other types: XML
Mapping Support conditional statements, can inject new parameters into payload, can hardcode values which is needed for mocking, map data in complex data structures and can reference data made available at runtime such as context and state variables.

Mock Integration:
Here we put some data as response (hard coded as VTL) to be returned back: modify the integration response.

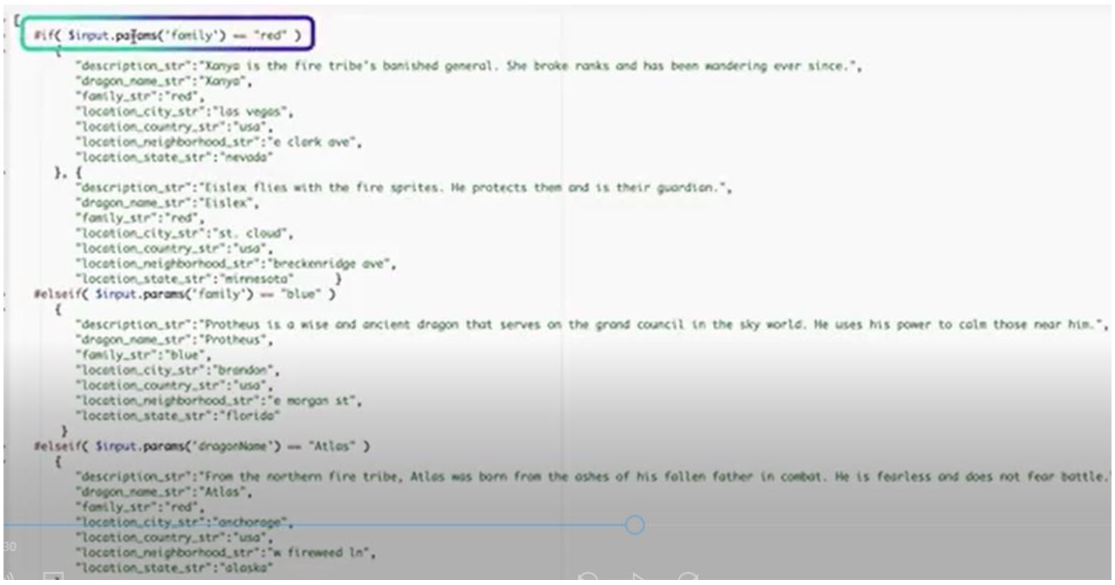
Dragon api: using mappings
For the mock get we had, we want to List dragons by name, family.
Send the request as below:



```
/dragon?dragonName=Atlas
/dragon?family=red
```

Modify the integration response to check the parameters using the conditional statements and

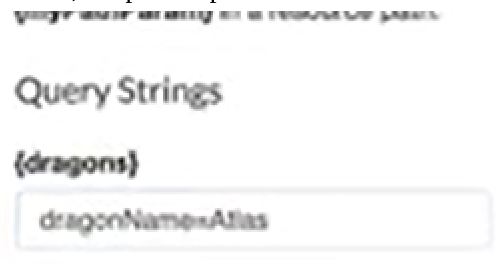
VTL.



```
[
  #if( $input.params('family') == "red" )
    {
        "description_str":"Xanya is the fire tribe's banished general. She broke ranks and has been wandering ever since.",
        "dragon_name_str":"Xanya",
        "family_str":"red",
        "location_city_str":"las vegas",
        "location_country_str":"usa",
        "location_neighborhood_str":"e clark ave",
        "location_state_str":"nevada"
    }, {
        "description_str":"Eislex flies with the fire sprites. He protects them and is their guardian.",
        "dragon_name_str":"Eislex",
        "family_str":"red",
        "location_city_str":"st. cloud",
        "location_country_str":"usa",
        "location_neighborhood_str":"breckenridge ave",
        "location_state_str":"minnesota"      }
  #elseif( $input.params('family') == "blue" )
    {
        "description_str":"Protheus is a wise and ancient dragon that serves on the grand council in the sky world. He uses his power to calm those near him.",
        "dragon_name_str":"Protheus",
        "family_str":"blue",
        "location_city_str":"brandon",
        "location_country_str":"usa",
        "location_neighborhood_str":"e morgan st",
        "location_state_str":"florida"
     }
  #elseif( $input.params('dragonName') == "Atlas" )
    {
        "description_str":"From the northern fire tribe, Atlas was born from the ashes of his fallen father in combat. He is fearless and does not fear battle.
        "dragon_name_str":"Atlas",
        "family_str":"red",
        "location_city_str":"anchorage",
        "location_country_str":"usa",
        "location_neighborhood_str":"w fireweed ln",
        "location_state_str":"alaska"
```

The first Condition is inside the rectangular. Use the $ to indicate the payload or context variables.

For test, we put the parameter as below:

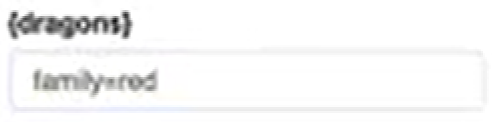


Query Strings

{dragons}

dragonName=Atlas

Or



{dragons}

family=red

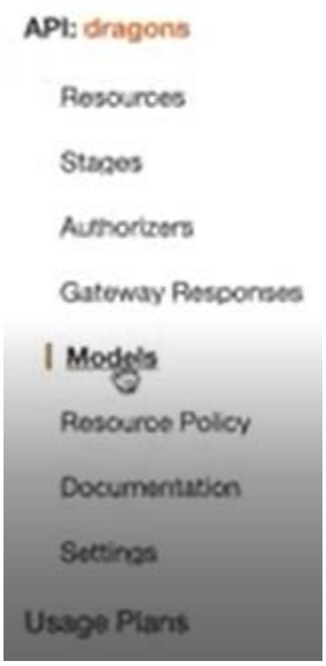
Dragon API: using models:

Validate the incoming request based on models:

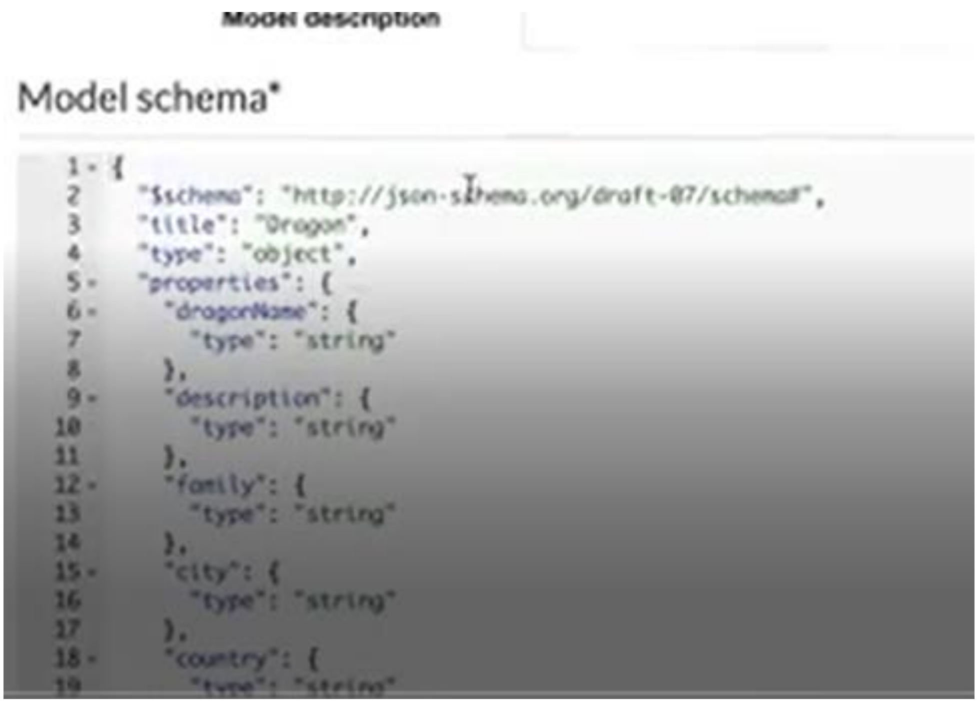
Here in first step we create the POST method.

First we create the model:

Click on the model on left:



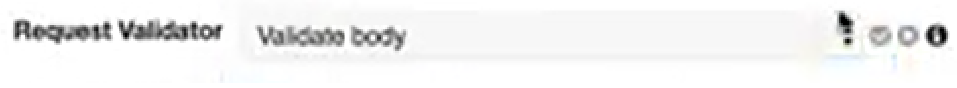
Past the pre-written JSON schema:



```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Dragon",
  "type": "object",
  "properties": {
    "dragonName": {
      "type": "string"
    },
    "description": {
      "type": "string"
    },
    "family": {
      "type": "string"
    },
    "city": {
      "type": "string"
    },
    "country": {
      "type": "string"
```

Then again click the resources:

Then click the created method (POST), then method request, then click the request body, then add model, select the model name(here dragon), to actually validate this, request the request validator,


Request Validator Validate body

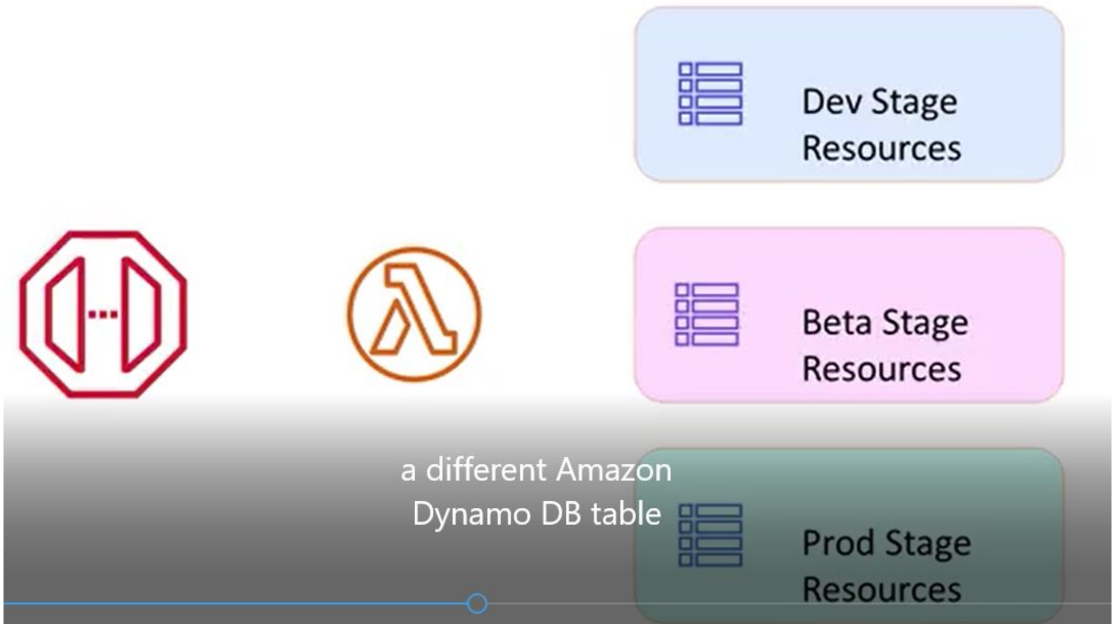
Then test it, we need to put the req data in request body.

Notes from reading:
For the basic validation, API Gateway verifies either or both of the following conditions:
The required request parameters in the URI, query string, and headers of an incoming request are included and non-blank
Validation is performed in the Method Request and Method Response of the API, we can associate models. This means that different methods under a resource can use different models.

Mappings are templates written in Velocity Template Language (VTL) which applies to integration req/response.
The mapping template allows you to transform data, including injecting hardcoded data, or changing the shape of the data before it passes to the backing service or before sending the response to the client
Stage Variables
They are name-value pairs that you can define as configuration attributes associated with a deployment stage of a REST API. They act like environment variables which can be used in API setup and mapping template.
With deployment stages in API Gateway, you can manage multiple release stages for each API,

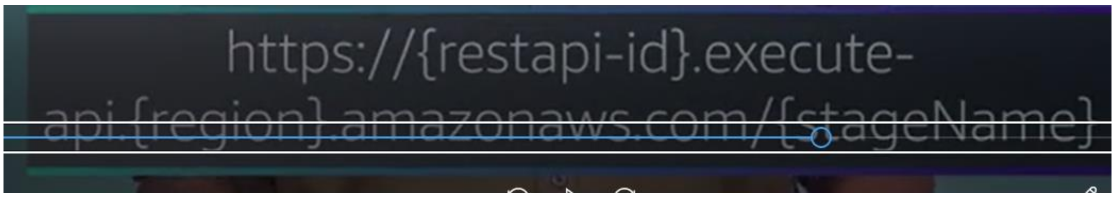
such as alpha, beta, and production
In VTL, we can reference it by $stageVariables.

Publish API:
Is represented by API deployment, and instantiated when create deployment. A stage (stage resource) is named reference to a deployment which is snapshot of API. Use stage to manage and optimize the API deployment, i.e. enable caching, customized request throttling, configure logging, or attach the Canary release for testing.
When we update the API, we can redeploy it by associating a new stage with existing deployment.
We have stage variables (key-value): configuration attributes, can be used in API setup and mapping templates.
We can also use the stage variables to pass configuration parameters to a Lambda Function through the mapping templates. i.e. u may re-use the Lambda function for different stages in API. The function read the data from different amazon dynamoDB table depending which stage is called. In mapping templates which generate the request, we can use the stage variables to pass the table name to the Lambda function.



Stage variables allows u to override stage level settings for individual methods and define the variables to pass stage specific environment contexts to the API integration at runtime.
Benefits:
1. Because of capability of having multiple versions of stages, we can have multiple versions of API.

The various stages of APIs can be called By API default domain name and appending stage name to the end.



https://{restapi-id}.execute-api.{region}.amazonaws.com/{stageName}

2. We can optimize the API performance by adjusting the default account level request throttling limits, we can enable logging for API calls to AWS cloud trail or Amazon Cloud Watch, and we can select a client certificate for back-end authenticate the API requests. Being able to throttling and monitoring as per stage, gives u a lot more control and visibility.
3. In order to create a platform or language level SDK for an API, we must first deploy the API in a stage (stage creation is necessary for SDK).

Summary:
1. Provide method for deployment
2. API versioning
3. Performance Management
4. SDK generating