

CDK Day May 2022 - Track 2



CDK Day
2.15K subscribers

Subscribe

32



Share

Download

Clip

Save

3,026 views Streamed live on May 26, 2022

Cloud Development Kit (CDK) is a developer tool built on the open source Constructs model. CDK Day is a one-day conference that attempts to showcase the brightest and best of CDK from AWS CDK, CDK for Terraform (CDKtf), CDK for Kubernetes (cdk8s), and projen. Let's talk serverless, Kubernetes and multi-cloud all on the same day.

This is the livestream for Track 2 of CDK Day, featuring talks from Ansgar Mertens, Jenna Krolick, Bill Penberthy, and more. For the full agenda, please see <https://www.cdkday.com/>.

For the livestream of talks from Track 1, please head over to <https://bit.ly/cdkday2022-track1>

For the livestream of talks from Track 3, please head over to <https://bit.ly/cdkday2022-track3>

00:00 Holding Page

38:39 Introductions

42:10 Building security with CDK

1:05:12 AWS Adapter: Using AWS CDK constructs with the CDK for Terraform

1:27:32 Using-Driver Composable Infrastructure with CDK

1:50:31 CDK & Team Topologies: Enabling the Optimal Platform Team

2:05:02 Build Event Driver Architectures with the AWS CDK

2:25:40 Snapshot Testing and CDK

2:50:02 Discussion: The local cloud - ideas to ensure developer productivity in serverless architectures

3:26:03 Schema-Driven OpenAPI Development with AWS CDK

3:47:52 Selling CDK to an Old-School DevOps org

4:17:12 Simplifying data pipelines by sharing AWS CDK Constructs within the team

4:36:44 Improved IAM through CDK

4:53:48 Closing comments

<https://www.cdkday.com/coc/>

-track1 #cdkday-track2 #cdkday-track3

cdk.dev #cdk



User-Driven Composable
Infrastructure with CDK

Luc van Donkersgoed

@donkersgood

4 Parts

1. Abstraction

2. CDK L4 Sub-Assemblies

3. Composable Infrastructure

4. Implementation Deep Dive

What is the CDK?

CloudFormation
Generator

Deployment
Toolchain

Abstraction
Framework

Abstraction

Secure Bucket

Ingestion Pipeline

Data Analysis

GraphQL Backend

The tools we use to build these abstractions are called Constructs

Constructs

L1 Constructs



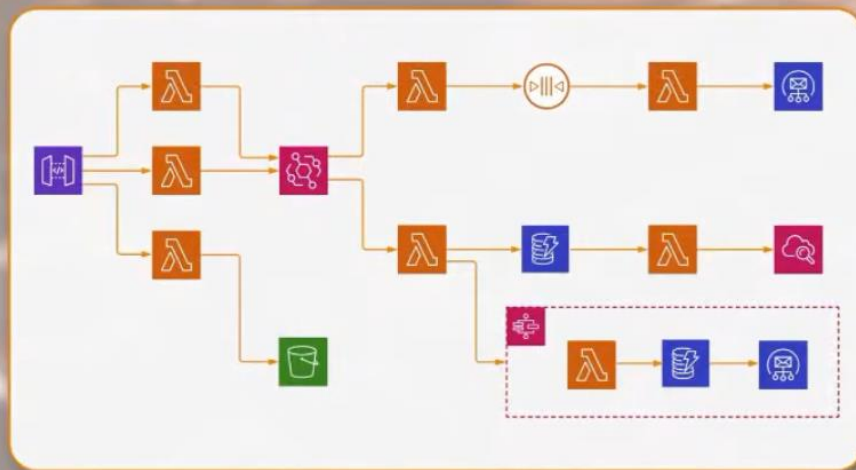
L2 Constructs



L3 Patterns



Application



Applications are generally stored as a single CDK project in Git.

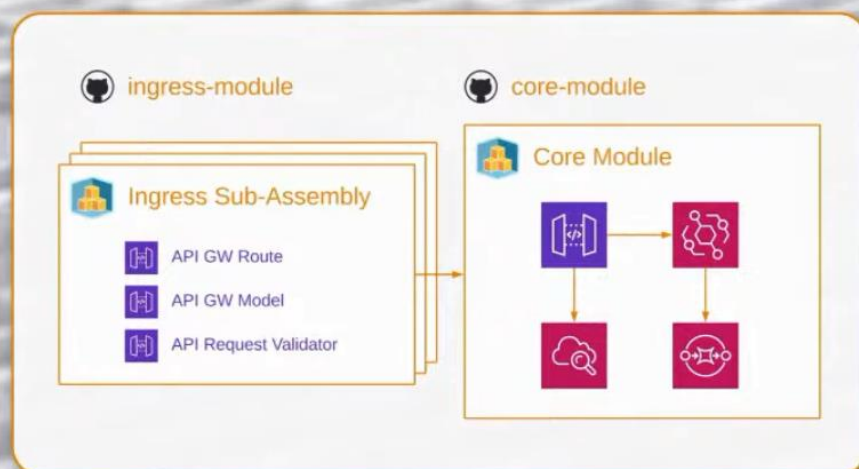
Common CDK Use Cases

Local deployment

Pipeline deployment

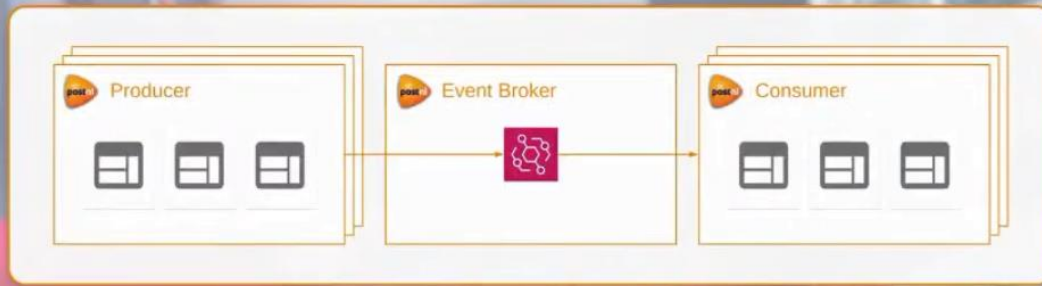
CDK L4
Sub-Assemblies

L4 Sub-Assemblies



Is a standalone CDK project that can and will be deployed as a standalone stack, it will not deliver complete functionality by itself but will integrate with other components in the core module that live in their own CDK projects and CF stacks.

Why

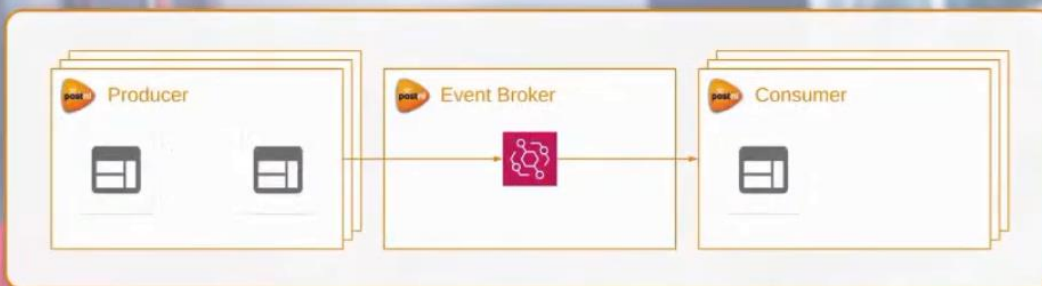


We have a central event broker that is based on the EventBridge, this allows all apps to publish events and subscribe to events in a decoupled manner.

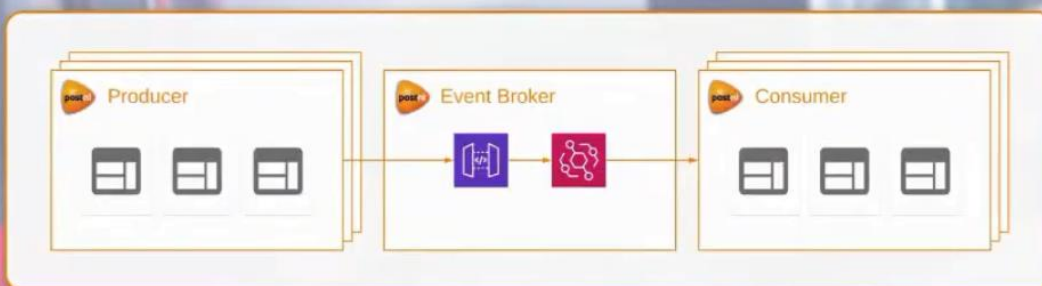
Innovate Faster

Self-Service

Self-Service

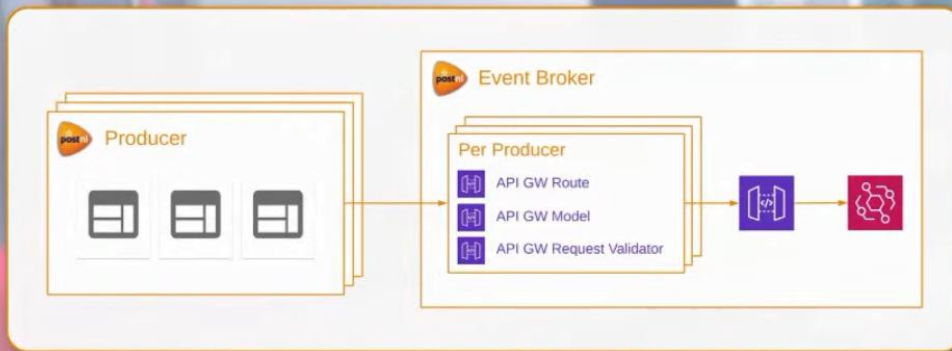


Reliability

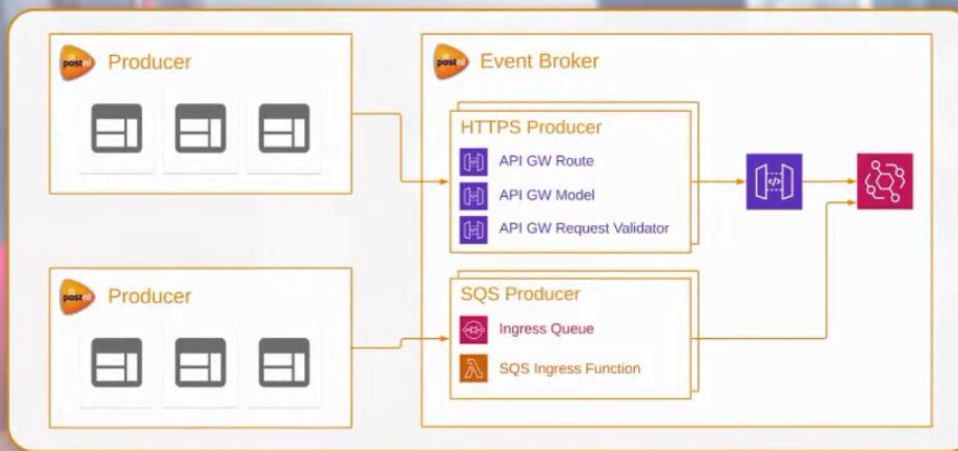


To guarantee reliability, we route events through an API Gateway that has a request validator that rejects invalid events.

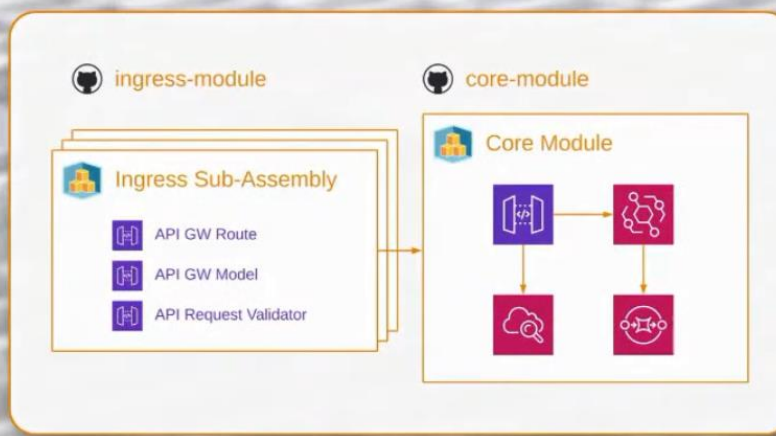
Reliability



Meet developers where they are



L4 Sub-Assemblies



We use the L4 Sub-Assemblies to deploy the relevant components per Producer, whether they need the HTTPS Producer component or the SQS Producer component. L3 Patterns are part of a larger application or service but the L4 Sub-Assemblies are standalone stacks or a full blown CDK application that does not deliver a full functionality by itself.

Real-Life Example

Stack name	Status
ProducerCeplaRetaillocationsV1	CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV2	CREATE_COMPLETE
ProducerFsnExecutioneventV1	CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV1	CREATE_COMPLETE
ProducerOompdDdaExecutioneventsDdV1	CREATE_COMPLETE
ProducerOompdpritemrouteupdateV1	CREATE_COMPLETE
ProducerOompdProtemrouteupdatesV1	CREATE_COMPLETE

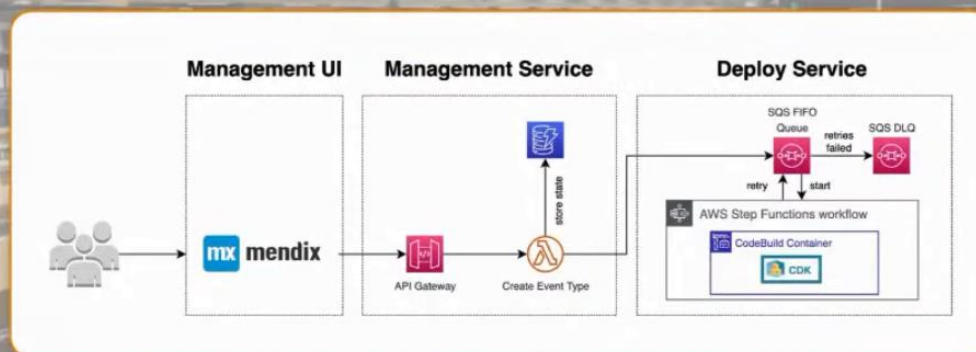
The above stacks are part of an L4 Sub-Assembly

ProducerEbeSanitytesteventV1
2022-05-20 15:45:28 UTC+0200
CREATE_COMPLETE

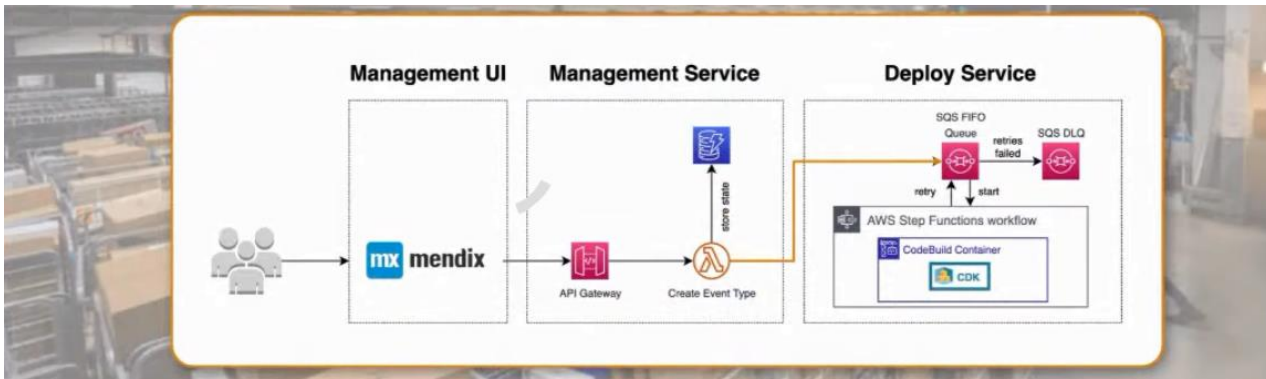
Physical ID	Type	Status
90147340-d287-11ec-988b-021b261d06b7	AWS::CDK::Metadata	CREATE_COMPLETE
ProducProdu618tSUVh2nhK	AWS::ApiGateway::Model	CREATE_COMPLETE
w4e4xi	AWS::ApiGateway::Resource	CREATE_COMPLETE
Produ-Produ-1OTOWAT1SKIPM	AWS::ApiGateway::Method	CREATE_COMPLETE
pmwynk	AWS::ApiGateway::RequestValidator	CREATE_COMPLETE

Each stack has a small number of resources needed like the ones above needed for ingesting events from the API Gateway.

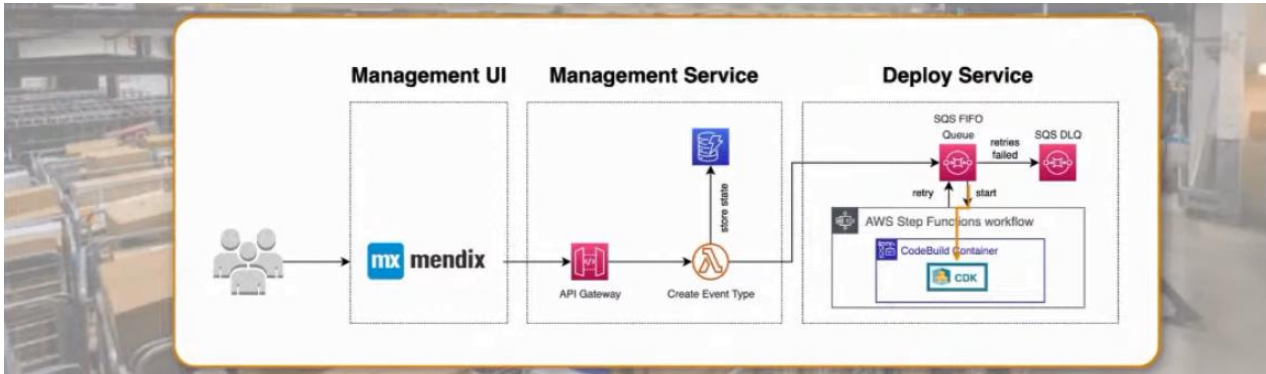
Implementation Deep Dive



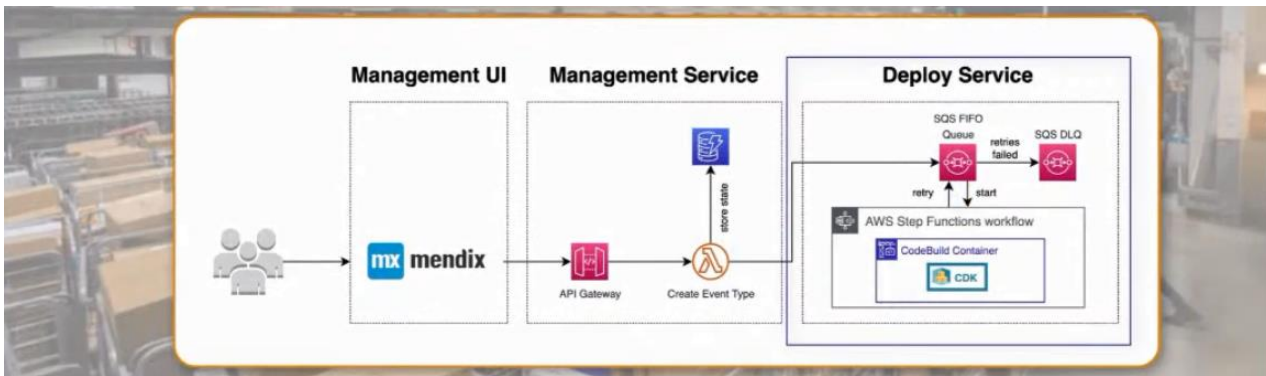
This shows an overview of the entire deployment process. It starts with a user registering an event in our management UI, the request is received by the API Gateway which authorizes the user, the request is then forwarded to a Lambda function that verifies the request itself.



Once verified, the request is forwarded to the Deploy Service as an SQS message

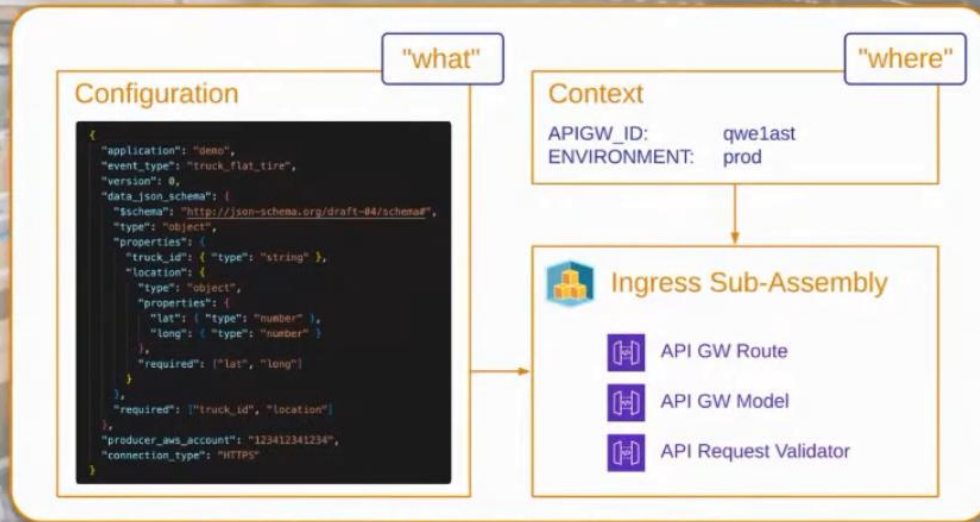


The SQS message then triggers a state machine/AWS Step Functions workflow, then through AWS CodeBuild it finally triggers the CDK.



Let us focus on the Deploy Service and start at the CDK -> Step Functions workflow -> SQS.

Implementation Deep Dive



The CDK project is an abstraction of the resources required for a **Producer**. To deploy these components, the project needs to know 2 things. **What** to deploy, which mentions the specifications of the event as provided by the user. These specifications will be different for every deployment, we will call this the **sub-assembly configuration**. Next, we need to know **Where** to deploy the project. This includes things like the AWS Account ID, environment name, and the identifiers of the core components the sub-assembly should integrate with. These values will be the same for every deployment which we will call the **sub-assembly context**. The **What** part is provided to the CDK as a configuration file in JSON, the **Where** part is provided through environmental variables.

Configuration

```
{
  "application": "demo",
  "event_type": "truck_flat_tire",
  "version": 0,
  "data_json_schema": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
      "truck_id": { "type": "string" },
      "location": {
        "type": "object",
        "properties": {
          "lat": { "type": "number" },
          "long": { "type": "number" }
        },
        "required": ["lat", "long"]
      },
      "required": ["truck_id", "location"]
    },
    "required": ["truck_id", "location"]
  },
  "producer_aws_account": "123412341234",
  "connection_type": "HTTPS"
}
```

This contains all the details provided by the Producer such as the app name, the event type, the schema that defines the event and the ingress connection type.

Context

APIGW_ID: qwe1ast
ENVIRONMENT: prod

The **Context** is a collection of environment variables which defines what the sub-assembly should interface with, above context includes the name of the API Gateway and the environment we want to deploy this sub-assembly into. These values are the same for every sub-assembly being deployed in our environments.

The CDK!

≡ .pylintrc

🔗 app.py

{ } cdk.json

▼ input_models

> __pycache__

🔗 __init__.py

🔗 models.py

```
from input_models.models import ProducerModel

if __name__ == "__main__":
    deploy_config = ProducerModel.model_from_file() ←
    build_app(deploy_config).synth()

@staticmethod
def model_from_dict(producer_deploy_config: dict) -> ProducerModel:
    """Instantiate a ProducerModel from an input dict."""
    producer_connection_type = producer_deploy_config.get("connection_type")

    if producer_connection_type == "HTTPS":
        return HttpsProducerModel(producer_deploy_config)
    if producer_connection_type == "SQS":
        return SqsProducerModel(producer_deploy_config)

    raise RuntimeError(f"Unknown connection type: {producer_connection_type}")

@classmethod
def model_from_file(cls) -> ProducerModel: ←
    """Instantiate a ProducerModel from an input file."""
    with open(
        BaseSystemContext().path_to_producer_config_file, encoding="utf-8"
    ) as deploy_request_handle:
        msg = json.load(deploy_request_handle)
        return cls.model_from_dict(msg)
```

In the main section, we start by loading the configuration JSON file as a Producer model to retrieve the dictionary values from it.

The CDK!

≡ .pylintrc
🔗 app.py
{ } cdkt.json

▼ input_models
> __pycache__
🔗 __init__.py
🔗 models.py

```
from input_models.models import ProducerModel

if __name__ == "__main__":
    deploy_config = ProducerModel.model_from_file()
    build_app(deploy_config).synth()

    @staticmethod
    def model_from_dict(producer_deploy_config: dict) -> ProducerModel:
        """Instantiate a ProducerModel from an input dict."""
        producer_connection_type = producer_deploy_config.get("connection_type")

        if producer_connection_type == "HTTPS":
            return HttpsProducerModel(producer_deploy_config)
        if producer_connection_type == "SQS":
            return SqsProducerModel(producer_deploy_config)

        raise RuntimeError(f"Unknown connection type: {producer_connection_type}")

    @classmethod
    def model_from_file(cls) -> ProducerModel:
        """Instantiate a ProducerModel from an input file."""
        with open(
            BaseSystemContext().path_to_producer_config_file, encoding="utf-8"
        ) as deploy_request_handle:
            msg = json.load(deploy_request_handle)
            return cls.model_from_dict(msg)
```

The values are then loaded into the next function `cls.model_from_dict` that takes the dictionary and turns it into a Producer model that we then can reuse in the rest of the application.

The CDK!

≡ .pylintrc
🔗 app.py
{ } cdkt.json

▼ input_models
> __pycache__
🔗 __init__.py
🔗 models.py

```
from input_models.models import ProducerModel

if __name__ == "__main__":
    deploy_config = ProducerModel.model_from_file()
    build_app(deploy_config).synth()

    @staticmethod
    def model_from_dict(producer_deploy_config: dict) -> ProducerModel:
        """Instantiate a ProducerModel from an input dict."""
        producer_connection_type = producer_deploy_config.get("connection_type")

        if producer_connection_type == "HTTPS":
            return HttpsProducerModel(producer_deploy_config)
        if producer_connection_type == "SQS":
            return SqsProducerModel(producer_deploy_config)

        raise RuntimeError(f"Unknown connection type: {producer_connection_type}")

    @classmethod
    def model_from_file(cls) -> ProducerModel:
        """Instantiate a ProducerModel from an input file."""
        with open(
            BaseSystemContext().path_to_producer_config_file, encoding="utf-8"
        ) as deploy_request_handle:
            msg = json.load(deploy_request_handle)
            return cls.model_from_dict(msg)
```

We determine the type of Producer and then return a specific model based on its connection type.

Store Context Locally

≡ .pylintrc

🔗 app.py

{ } cdk.json

{ } cdk.json

🔗 context.py

🔗 deploy.py

```
def build_app(producer_deploy_config: ProducerModel) -> App:
    """Build the CDK app. Wrapped in a function for testability."""
    local_app = App()
    context = BaseSystemContext()

def __init__(self) -> None:
    """Construct a new Context class."""
    self._stack_prefix = os.environ.get("STACK_PREFIX", "").lower()
    self._environment = self._osgetenv("ENVIRONMENT")
    self._deploy_role_arn = self._osgetenv("DEPLOY_ROLE_ARN", False)
    self._execution_role_arn = self._osgetenv("EXECUTION_ROLE_ARN", False)
    self._path_to_producer_config_file = "config/deploy.json"
```

Still in the app.py, we load the environment variables and store them in a system context object/class.

Create the Stack

≡ .pylintrc

🔗 app.py

{ } cdk.json

```
def build_app(producer_deploy_config: ProducerModel) -> App:
    """Build the CDK app. Wrapped in a function for testability."""
    local_app = App()
    context = BaseSystemContext()

    EbeProducerComponentsCdkStack(
        scope=local_app,
        construct_id="EbeProducerComponentsCdkStack",
        producer_deploy_config=producer_deploy_config,
        stack_name=(
            pascalcase(context.stack_prefix)
            + pascalcase(f"Producer_{producer_deploy_config.uniq_resource_id}")
        ),
        synthesizer=DefaultStackSynthesizer(
            deploy_role_arn=context.deploy_role_arn,
            file_asset_publishing_role_arn=context.deploy_role_arn,
            image_asset_publishing_role_arn=context.deploy_role_arn,
            lookup_role_arn=context.deploy_role_arn,
            cloud_formation_execution_role=context.execution_role_arn,
        ),
    )
```

Finally, we then create the stack that we want to deploy.

Create the Stack

≡ .pylintrc
🔗 app.py
{ } cdk.json

Stack name	Status
ProducerCepiaRetailLocationsV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsV2	🟢 CREATE_COMPLETE
ProducerFsnExecutionEventV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsDdV1	🟢 CREATE_COMPLETE
ProducerOompdPrtemroutupdateV1	🟢 CREATE_COMPLETE
ProducerOompdPrtemroutupdateV1	🟢 CREATE_COMPLETE

```
def build_app(producer_deploy_config: ProducerModel) -> App:
    """Build the CDK app. Wrapped in a function for testability."""
    local_app = App()
    context = BaseSystemContext()

    EbeProducerComponentsCdkStack(
        scope=local_app,
        construct_id="EbeProducerComponentsCdkStack",
        producer_deploy_config=producer_deploy_config,
        stack_name=(
            pascalcase(context.stack_prefix)
            + pascalcase(f"Producer_{producer_deploy_config.uniq_resource_id}")
        ),
        synthesizer=DefaultStackSynthesizer(
            deploy_role_arn=context.deploy_role_arn,
            file_asset_publishing_role_arn=context.deploy_role_arn,
            image_asset_publishing_role_arn=context.deploy_role_arn,
            lookup_role_arn=context.deploy_role_arn,
            cloud_formation_execution_role=context.execution_role_arn,
        ),
    )
```

We dynamically determine the stack name that allows us to dynamically generate stack name (different for each sub-assembly). This allows us to deploy the sub-assembly many times within the same environment.

Create the Stack

≡ .pylintrc
🔗 app.py
{ } cdk.json

Stack name	Status
ProducerCepiaRetailLocationsV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsV2	🟢 CREATE_COMPLETE
ProducerFsnExecutionEventV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsV1	🟢 CREATE_COMPLETE
ProducerOompdDdaExecutionEventsDdV1	🟢 CREATE_COMPLETE
ProducerOompdPrtemroutupdateV1	🟢 CREATE_COMPLETE
ProducerOompdPrtemroutupdateV1	🟢 CREATE_COMPLETE

```
def build_app(producer_deploy_config: ProducerModel) -> App:
    """Build the CDK app. Wrapped in a function for testability."""
    local_app = App()
    context = BaseSystemContext()

    EbeProducerComponentsCdkStack(
        scope=local_app,
        construct_id="EbeProducerComponentsCdkStack",
        producer_deploy_config=producer_deploy_config,
        stack_name=(
            pascalcase(context.stack_prefix)
            + pascalcase(f"Producer_{producer_deploy_config.uniq_resource_id}")
        ),
        synthesizer=DefaultStackSynthesizer(
            deploy_role_arn=context.deploy_role_arn,
            file_asset_publishing_role_arn=context.deploy_role_arn,
            image_asset_publishing_role_arn=context.deploy_role_arn,
            lookup_role_arn=context.deploy_role_arn,
            cloud_formation_execution_role=context.execution_role_arn,
        ),
    )
```

We overwrite the deploy roles used by the CDK, these roles will later be provided by AWS CodeBuild.

Select the L3 Pattern

```
> constructs
+ __init__.py
+ producer_stack.py
```

```
class EbeProducerComponentsCdkStack(Stack):
    """The EbeProducerComponentsCdkStack deploys the L3 pattern for the selected ingress type."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: ProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new producer sub-assembly stack."""
        super().__init__(scope, construct_id, **kwargs)

        construct_mapping = {
            "HTTPS": HttpsProducerConstruct,
            "SQS": SqsProducerConstruct,
        }

        # Fetch the right construct based on the ingestion type in the config
        construct_class = construct_mapping[producer_deploy_config.connection_type]

        # Initialize the construct
        construct_class(
            scope=self,
            construct_id="ProducerConstruct",
            producer_deploy_config=producer_deploy_config,
        )
```

In this stack, we provide the Producer model as a parameter.

Select the L3 Pattern

```
> constructs
+ __init__.py
+ producer_stack.py
```

```
class EbeProducerComponentsCdkStack(Stack):
    """The EbeProducerComponentsCdkStack deploys the L3 pattern for the selected ingress type."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: ProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new producer sub-assembly stack."""
        super().__init__(scope, construct_id, **kwargs)

        construct_mapping = {
            "HTTPS": HttpsProducerConstruct,
            "SQS": SqsProducerConstruct,
        }

        # Fetch the right construct based on the ingestion type in the config
        construct_class = construct_mapping[producer_deploy_config.connection_type]

        # Initialize the construct
        construct_class(
            scope=self,
            construct_id="ProducerConstruct",
            producer_deploy_config=producer_deploy_config,
        )
```

We looked at which L3 pattern to deploy based on the connection type in the configuration,

Select the L3 Pattern

```
> constructs
  _init__.py
  producer_stack.py
```

```
class EbeProducerComponentsCdkStack(Stack):
    """The EbeProducerComponentsCdkStack deploys the L3 pattern for the selected ingress type."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: ProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new producer sub-assembly stack."""
        super().__init__(scope, construct_id, **kwargs)

        construct_mapping = {
            "HTTPS": HttpsProducerConstruct,
            "SQS": SqsProducerConstruct,
        }

        # Fetch the right construct based on the ingestion type in the config
        construct_class = construct_mapping[producer_deploy_config.connection_type]

        # Initialize the construct
        construct_class(
            scope=self,
            construct_id="ProducerConstruct",
            producer_deploy_config=producer_deploy_config,
        )
```

We used a mapping to map the connection type to a L3 pattern.

Select the L3 Pattern

```
> constructs
  _init__.py
  producer_stack.py
```

```
class EbeProducerComponentsCdkStack(Stack):
    """The EbeProducerComponentsCdkStack deploys the L3 pattern for the selected ingress type."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: ProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new producer sub-assembly stack."""
        super().__init__(scope, construct_id, **kwargs)

        construct_mapping = {
            "HTTPS": HttpsProducerConstruct,
            "SQS": SqsProducerConstruct,
        }

        # Fetch the right construct based on the ingestion type in the config
        construct_class = construct_mapping[producer_deploy_config.connection_type]

        # Initialize the construct
        construct_class(
            scope=self,
            construct_id="ProducerConstruct",
            producer_deploy_config=producer_deploy_config,
        )
```

We then initialized that L3 pattern using the **producer_deploy_config** as a configuration parameter.

The HTTPS L3 Pattern

- ebe_alarm.py
- https_producer.py
- postnl_lambda_base.py

```
class HttpsProducerConstruct(Construct):
    """The EBE Backend Producer HTTPS Components L3 Pattern."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: HttpsProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new EBE Backend Producer HTTPS sub-assembly."""
        super().__init__(scope, construct_id, **kwargs)
        system_context = HttpsSystemContext()

        rest_api = apigateway.RestApi.from_rest_api_attributes(
            scope=self,
            id="RestApi",
            rest_api_id=system_context.rest_api_id,
            root_resource_id=system_context.root_resource_id,
        )
```

```
request_model = apigateway.Model(
    scope=self,
    id="Model",
    rest_api=rest_api,
    content_type="application/json",
    schema=apigateway.JsonSchema(),
)

# Need to set model directly to prevent issues with schema conversion
cfn_model: apigateway.CfnModel = request_model.node.default_child
cfn_model.schema = producer_deploy_config.data_json_schema

request_validator = apigateway.RequestValidator(
    scope=self,
    id="Validator",
    rest_api=rest_api,
    validate_request_body=True,
    validate_request_parameters=False,
)
```

Finally, in the CDK, we actually deploy the L3 pattern. We log the context/environmental variables.

The HTTPS L3 Pattern

- ebe_alarm.py
- https_producer.py
- postnl_lambda_base.py

```
class HttpsProducerConstruct(Construct):
    """The EBE Backend Producer HTTPS Components L3 Pattern."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: HttpsProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new EBE Backend Producer HTTPS sub-assembly."""
        super().__init__(scope, construct_id, **kwargs)
        system_context = HttpsSystemContext()

        rest_api = apigateway.RestApi.from_rest_api_attributes(
            scope=self,
            id="RestApi",
            rest_api_id=system_context.rest_api_id,
            root_resource_id=system_context.root_resource_id,
        )
```

```
request_model = apigateway.Model(
    scope=self,
    id="Model",
    rest_api=rest_api,
    content_type="application/json",
    schema=apigateway.JsonSchema(),
)

# Need to set model directly to prevent issues with schema conversion
cfn_model: apigateway.CfnModel = request_model.node.default_child
cfn_model.schema = producer_deploy_config.data_json_schema

request_validator = apigateway.RequestValidator(
    scope=self,
    id="Validator",
    rest_api=rest_api,
    validate_request_body=True,
    validate_request_parameters=False,
)
```

Then we import an existing API Gateway using the REST API ID and the root resource ID provided to the context.

The HTTPS L3 Pattern

- ebe_alarm.py
- **https_producer.py**
- postnl_lambda_base.py

```
class HttpsProducerConstruct(Construct):
    """The EBE Backend Producer HTTPS Components L3 Pattern."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: HttpsProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new EBE Backend Producer HTTPS sub-assembly."""
        super().__init__(scope, construct_id, **kwargs)
        system_context = HttpsSystemContext()

        rest_api = apigateway.RestApi.from_rest_api_attributes(
            scope=self,
            id="RestApi",
            rest_api_id=system_context.rest_api_id,
            root_resource_id=system_context.root_resource_id,
        )
```

```
request_model = apigateway.Model(
    scope=self,
    id="Model",
    rest_api=rest_api,
    content_type="application/json",
    schema=apigateway.JsonSchema(),
)

# Need to set model directly to prevent issues with schema conversion
cfn_model: apigateway.CfnModel = request_model.node.default_child
cfn_model.schema = producer_deploy_config.data_json_schema

request_validator = apigateway.RequestValidator(
    scope=self,
    id="Validator",
    rest_api=rest_api,
    validate_request_body=True,
    validate_request_parameters=False,
)
```

We create a number of resources like the API Gateway model and the API Gateway request validator, we then used the configuration supplied through the configuration file.

The HTTPS L3 Pattern

- ebe_alarm.py
- **https_producer.py**
- postnl_lambda_base.py

```
class HttpsProducerConstruct(Construct):
    """The EBE Backend Producer HTTPS Components L3 Pattern."""

    def __init__(
        self,
        scope: Construct,
        construct_id: str,
        producer_deploy_config: HttpsProducerModel,
        **kwargs,
    ) -> None:
        """Construct a new EBE Backend Producer HTTPS sub-assembly."""
        super().__init__(scope, construct_id, **kwargs)
        system_context = HttpsSystemContext()

        rest_api = apigateway.RestApi.from_rest_api_attributes(
            scope=self,
            id="RestApi",
            rest_api_id=system_context.rest_api_id,
            root_resource_id=system_context.root_resource_id,
        )
```

```
request_model = apigateway.Model(
    scope=self,
    id="Model",
    rest_api=rest_api,
    content_type="application/json",
    schema=apigateway.JsonSchema(),
)

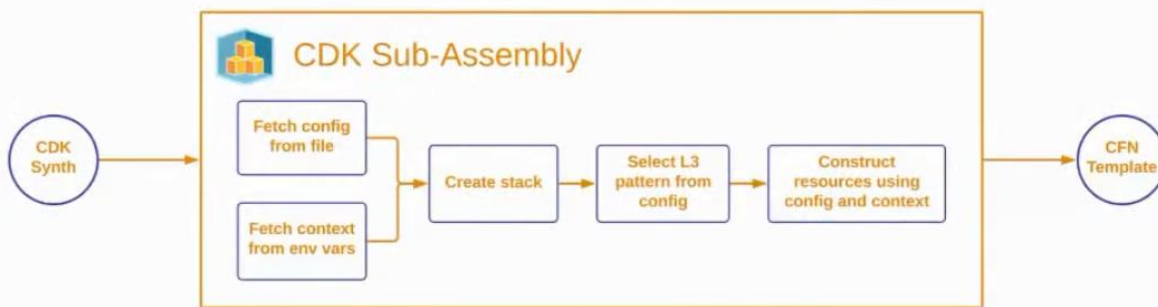
# Need to set model directly to prevent issues with schema conversion
cfn_model: apigateway.CfnModel = request_model.node.default_child
cfn_model.schema = producer_deploy_config.data_json_schema

request_validator = apigateway.RequestValidator(
    scope=self,
    id="Validator",
    rest_api=rest_api,
    validate_request_body=True,
    validate_request_parameters=False,
)
```

Physical ID	Type	Status
90147340-4287-11ec-98bb-021b261d98b7	AWS::CDK::Metadata	CREATE_COMPLETE
ProduxProdux6BHSUVhZvnhK	AWS::ApiGateway::Model	CREATE_COMPLETE
w4e4xi	AWS::ApiGateway::Resource	CREATE_COMPLETE
ProduxProdux1OTOWAT1SK0PM	AWS::ApiGateway::Method	CREATE_COMPLETE
pnwyrnk	AWS::ApiGateway::RequestValidator	CREATE_COMPLETE

These resources translate directly to the resources being deployed by CloudFormation

CDK Sub-Assembly Recap



The **CDK responsibilities** includes:

1. Fetching the config file and context and store them in objects.
2. It then creates a CF stack using values from the configuration to determine the stack name.
3. It then determines which L3 pattern to use based on the ingress type specified by the producer.
4. Finally, it constructs the resources in that pattern using the configurations specified.

Local Deployment

```
(.env) ➔ ebe-producer-components-cdk git:(feature/cdk_cleanup) ✗ cdk deploy --all --require-approval never --profile PostNL-EBE-Dev
⚡ Synthesis time: 9.29s

EbeProducerComponentsCdkStack (LucProducerDemoTruckFlatTireV0): deploying...
[0%] start: Publishing 69ddcd2a4a486fbc17af0f469a08c7af181240d5392a44eb6b7a7604004c03c6:current_account-current_region
[100%] success: Published 69ddcd2a4a486fbc17af0f469a08c7af181240d5392a44eb6b7a7604004c03c6:current_account-current_region
LucProducerDemoTruckFlatTireV0: creating CloudFormation changeset...
[.....] (1/6)

2:35:48 PM | CREATE_IN_PROGRESS | AWS::CloudFormation::Stack | LucProducerDemoTruckFlatTireV0
2:35:52 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Resource | EbeProducerCompone...truck_flat_tire_v0
2:35:54 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::RequestValidator | EbeProducerCompone...onstruct/Validator
2:35:54 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Model | EbeProducerCompone...cerConstruct/Model

✔ EbeProducerComponentsCdkStack (LucProducerDemoTruckFlatTireV0)
⚡ Deployment time: 23.45s

Stack ARN:
arn:aws:cloudformation:eu-west-1:927525665196:stack/LucProducerDemoTruckFlatTireV0/5c8b47d0-d839-11ec-8fe9-02c1dc863d97

⚡ Total time: 32.74s

(.env) ➔ ebe-producer-components-cdk git:(feature/cdk_cleanup) ✗
```

To the CDK project, the configuration file is just a JSON file in a predefined location. The context environment variables are just the variables available through the OS. This allows us to do local deployments which we use during testing and validation.

Local Deployment

```
(.venv) ➔ ebe-producer-components-cdk git:(feature/cdk_cleanup) ✕ cdk deploy --all --require-approval never --profile PostNL-EBE-Dev
✕ Synthesis time: 9.29s

EbeProducerComponentsCdkStack (LucProducerDemoTruckFlatTireV0): deploying...
[0%] start: Publishing 69ddcd2a4a486fbc17af0f469a08c7af181240d5392a44eb6b7a7604004c03c6:current_account-current_region
[100%] success: Published 69ddcd2a4a486fbc17af0f469a08c7af181240d5392a44eb6b7a7604004c03c6:current_account-current_region
LucProducerDemoTruckFlatTireV0: creating CloudFormation changeset...
[.....] (1/6)

2:35:48 PM | CREATE_IN_PROGRESS | AWS::CloudFormation::Stack | LucProducerDemoTruckFlatTireV0
2:35:52 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Resource | EbeProducerCompone...truck_flat_tire_v0
2:35:54 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::RequestValidator | EbeProducerCompone...onstruct/Validator
2:35:54 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Model | EbeProducerCompone...cerConstruct/Model

✔ EbeProducerComponentsCdkStack (LucProducerDemoTruckFlatTireV0)
✕ Deployment time: 23.45s

Stack ARN:
arn:aws:cloudformation:eu-west-1:927525665196:stack/LucProducerDemoTruckFlatTireV0/5c8b47d0-d839-11ec-8fe9-02c1dc863d97

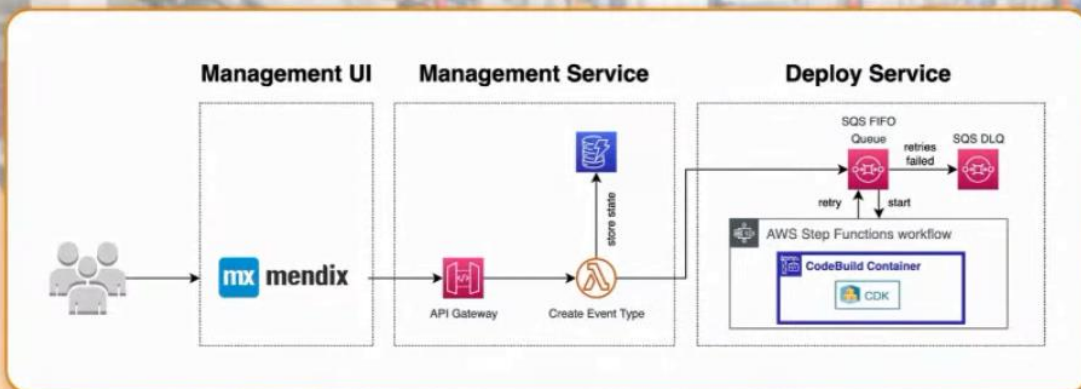
✕ Total time: 32.74s

(.venv) ➔ ebe-producer-components-cdk git:(feature/cdk_cleanup) ✕
```

User-Driven Deployments

But a local deployment isn't very user-driven, we need a built environment that can be triggered by a user request. AWS CodeBuild is a perfect use-case for this because it allows us to spin up containers for a one-time job like a CDK deployment.

CodeBuild



Let us look at how to invoke the CDK project on the lower right corner.

User-Driven Deployments

```
"version": "0.2",
"phases": {
  "install": {
    "commands": [
      "npm install -g aws-cdk@2",
    ],
  },
  "pre_build": {
    "commands": [
      "echo 'Fetching producer sub-assembly artifact'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_components/$PRODUCER_COMPONENTS_VERSION.zip .",
      "unzip -o $PRODUCER_COMPONENTS_VERSION.zip",
      "echo 'Fetching producer configuration'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_config_payloads/$STATE_MACHINE_EXECUTION_ID.json config/deploy.json",
      "echo 'Installing requirements'",
      "pip install -r requirements.txt",
    ],
  },
  "build": {
    "commands": [
      "cdk deploy --require-approval never",
    ],
  },
}
```

38

The CodeBuild container's responsibilities are defined in a buildSpec script that runs when the container is started.

User-Driven Deployments

```
"version": "0.2",
"phases": {
  "install": {
    "commands": [
      "npm install -g aws-cdk@2",
    ],
  },
  "pre_build": {
    "commands": [
      "echo 'Fetching producer sub-assembly artifact'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_components/$PRODUCER_COMPONENTS_VERSION.zip .",
      "unzip -o $PRODUCER_COMPONENTS_VERSION.zip",
      "echo 'Fetching producer configuration'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_config_payloads/$STATE_MACHINE_EXECUTION_ID.json config/deploy.json",
      "echo 'Installing requirements'",
      "pip install -r requirements.txt",
    ],
  },
  "build": {
    "commands": [
      "cdk deploy --require-approval never",
    ],
  },
}
```

38

User-Driven Deployments

```
"version": "0.2",
"phases": {
  "install": {
    "commands": [
      "npm install -g aws-cdk@2",
    ],
  },
  "pre_build": {
    "commands": [
      "echo 'Fetching producer sub-assembly artifact'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_components/$PRODUCER_COMPONENTS_VERSION.zip .",
      "unzip -o $PRODUCER_COMPONENTS_VERSION.zip",

      "echo 'Fetching producer configuration'",
      "aws s3 cp s3://$ARTIFACT_BUCKET/producer_config_payloads/$STATE_MACHINE_EXECUTION_ID.json config/deploy.json"

      "echo 'Installing requirements'",
      "pip install -r requirements.txt",
    ],
  },
  "build": {
    "commands": [
      "cdk deploy --require-approval never",
    ],
  },
}
```

38

```
environment_variables={
  "PRODUCER_COMPONENTS_VERSION": codebuild.BuildEnvironmentVariable(value=config.producer_components_version),
  "ARTIFACT_BUCKET": codebuild.BuildEnvironmentVariable(value=params.artifact_bucket.bucket_name),
  "REST_API_ID": codebuild.BuildEnvironmentVariable(value=params.rest_api.rest_api_id),
  "EVENT_BUS_ARN": codebuild.BuildEnvironmentVariable(value=params.event_bus_arn),
  "APIGATEWAY_INGESTION_FUNCTION_ARN": codebuild.BuildEnvironmentVariable(value=params.apigateway_ingestion_function.function_arn),
  "ROOT_RESOURCE_ID": codebuild.BuildEnvironmentVariable(value=params.rest_api.root_resource_id),
  "STACK_PREFIX": codebuild.BuildEnvironmentVariable(value=config.stack_prefix or ""),
  "DEPLOY_ROLE_ARN": codebuild.BuildEnvironmentVariable(value=deploy_role.role_arn),
  "EXECUTION_ROLE_ARN": codebuild.BuildEnvironmentVariable(value=cfn_execution_role.role_arn),
  "ENVIRONMENT": codebuild.BuildEnvironmentVariable(value=config.environment),
},
```

```
[Container] 2022/05/20 13:45:20 Phase complete: PRE_BUILD State: SUCCEEDED
[Container] 2022/05/20 13:45:20 Phase context status code: Message:
[Container] 2022/05/20 13:45:20 Entering phase BUILD
[Container] 2022/05/20 13:45:20 Running command /deploy.py --require-approval never

^ Syntax time: 5.95s

EbeProducerComponentsCdStack (ProducerBeSanitytesteventV1): deploying...
[OK] start: Publishing ea20a023fed4074d93f8275f85425d7a5b8d4ac323a0a21462425799c9e4b8cc:current_account-current_region
[INFO] success: Full image eab0b32fe0b074091f175f54520fa19b404e323a0a21462425799c9e4b8cc:current_account-current_region
ProducerBeSanitytesteventV1: creating CloudFormation change set...
ProducerBeSanitytesteventV1 8/8 1:45:28 PM | REVISION_IN_PROGRESS | AWS::CloudFormation::Stack | ProducerBeSanitytesteventV1 User Initiated
ProducerBeSanitytesteventV1 8/8 1:45:33 PM | CREATE_IN_PROGRESS | AWS::CloudFormation::Stack | ProducerBeSanitytesteventV1 User Initiated
ProducerBeSanitytesteventV1 8/8 1:45:37 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::RequestValidator | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:37 PM | CREATE_IN_PROGRESS | AWS::CloudWatch::Alarm | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:37 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Resource | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:37 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Model | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:38 PM | CREATE_IN_PROGRESS | AWS::CloudWatch::Alarm | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:38 PM | CREATE_IN_PROGRESS | AWS::CDK::Metadata | EbeProducerComponentsCdStack/CDKMetadata/Default
ProducerBeSanitytesteventV1 8/8 1:45:38 PM | CREATE_IN_PROGRESS | AWS::CloudWatch::Alarm | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:38 PM | CREATE_IN_PROGRESS | AWS::CloudWatch::Alarm | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 2/8 1:45:38 PM | CREATE_COMPLETE | AWS::CloudWatch::Alarm | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 2/8 1:45:39 PM | CREATE_IN_PROGRESS | AWS::CDK::Metadata | EbeProducerComponentsCdStack/CDKMetadata/Default
ProducerBeSanitytesteventV1 3/8 1:45:39 PM | CREATE_COMPLETE | AWS::ApiGateway::Resource | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 4/8 1:45:39 PM | CREATE_COMPLETE | AWS::CDK::Metadata | EbeProducerComponentsCdStack/CDKMetadata/Default
ProducerBeSanitytesteventV1 4/8 1:45:40 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::RequestValidator | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 4/8 1:45:40 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Model | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 5/8 1:45:40 PM | CREATE_COMPLETE | AWS::ApiGateway::RequestValidator | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 6/8 1:45:40 PM | CREATE_COMPLETE | AWS::ApiGateway::Model | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 6/8 1:45:42 PM | CREATE_IN_PROGRESS | AWS::ApiGateway::Method | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 6/8 1:45:43 PM | CREATE_COMPLETE | AWS::ApiGateway::Method | EbeProducerComponentsCdStack/ProducerConstruct/
ProducerBeSanitytesteventV1 8/8 1:45:44 PM | CREATE_COMPLETE | AWS::CloudFormation::Stack | ProducerBeSanitytesteventV1

EbeProducerComponentsCdStack (ProducerBeSanitytesteventV1)
^ Deployment time: 21.94s

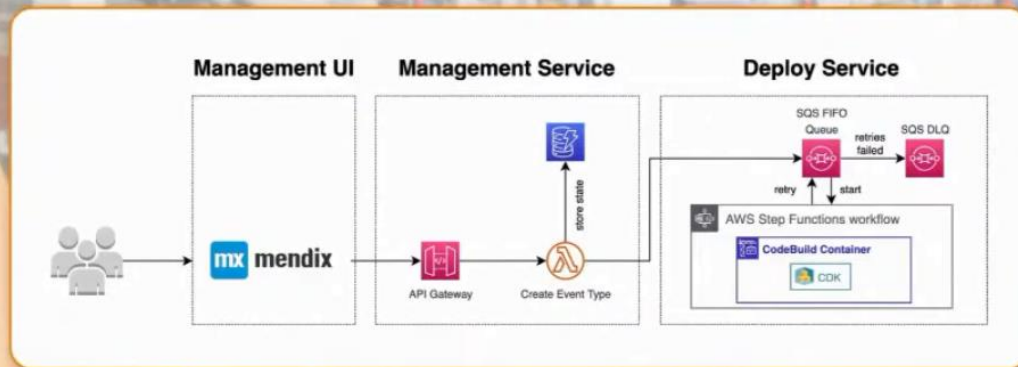
Stack ARN:
arn:aws:cloudformation:eu-west-1:123456789012:stack/ProducerBeSanitytesteventV1/1be0f0e4b43-11ec-93fa-4ac2d5008861

^ Total time: 27.86s
```

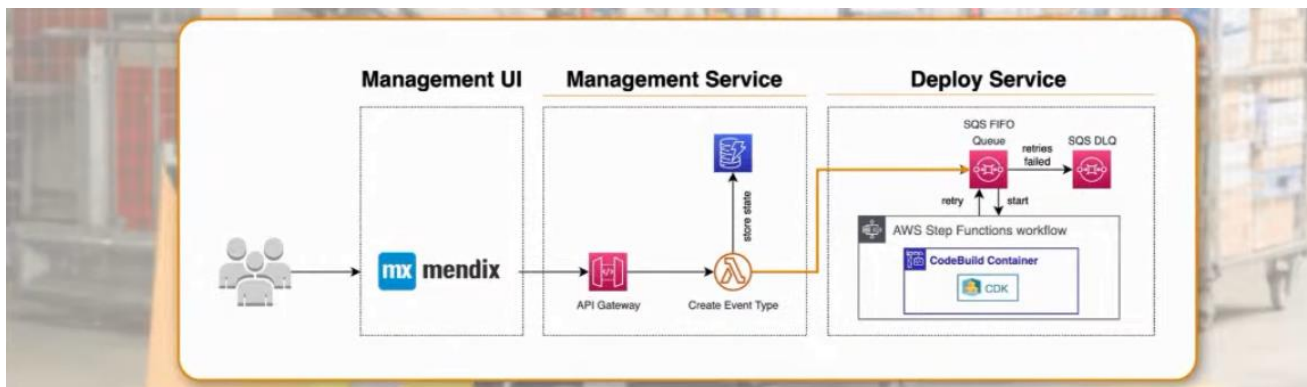
40

We then run the CodeBuild container to deploy a new sub-assembly.

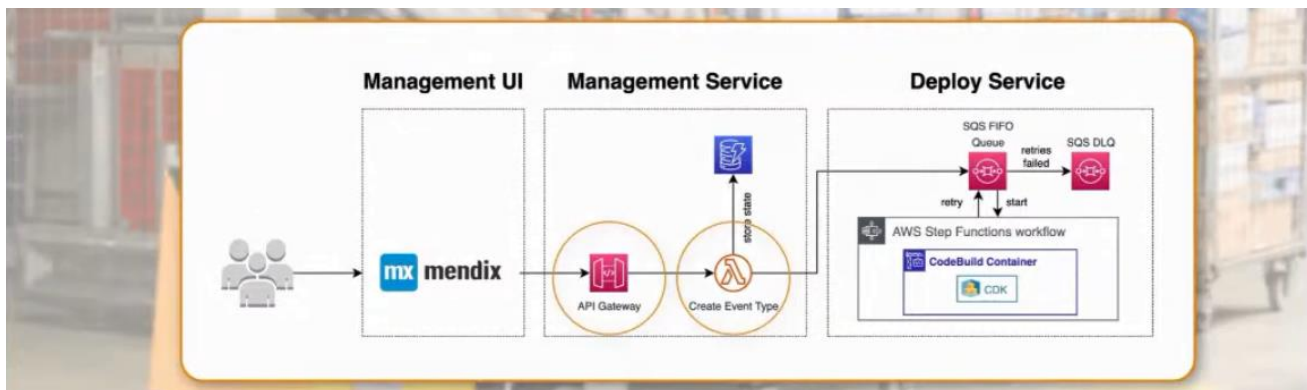
Trigger CodeBuild



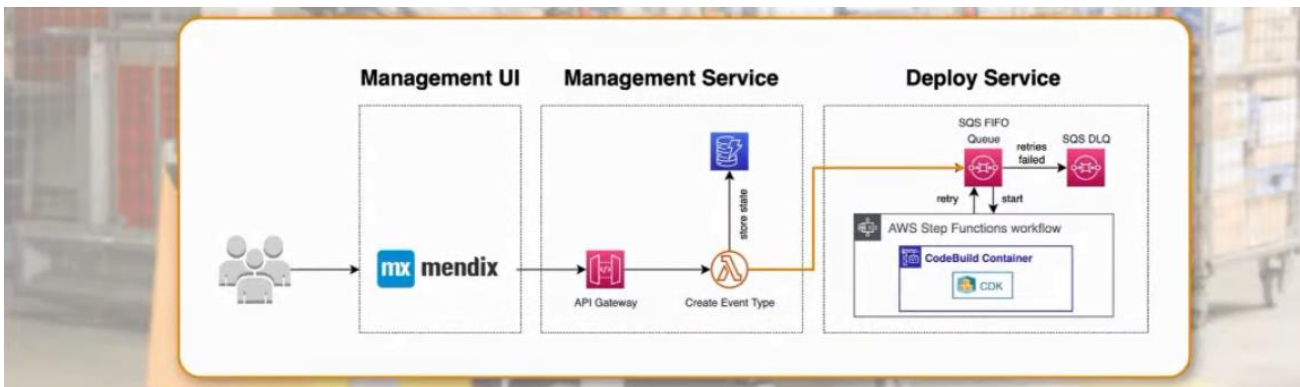
We have seen how to use CodeBuild to deploy the sub-assembly, but the user still does not know how to trigger CodeBuild. Next, let us see how a user can invoke a sub-assembly deployment.



The interface between the Management Service and the Deploy Service is an SQS message.



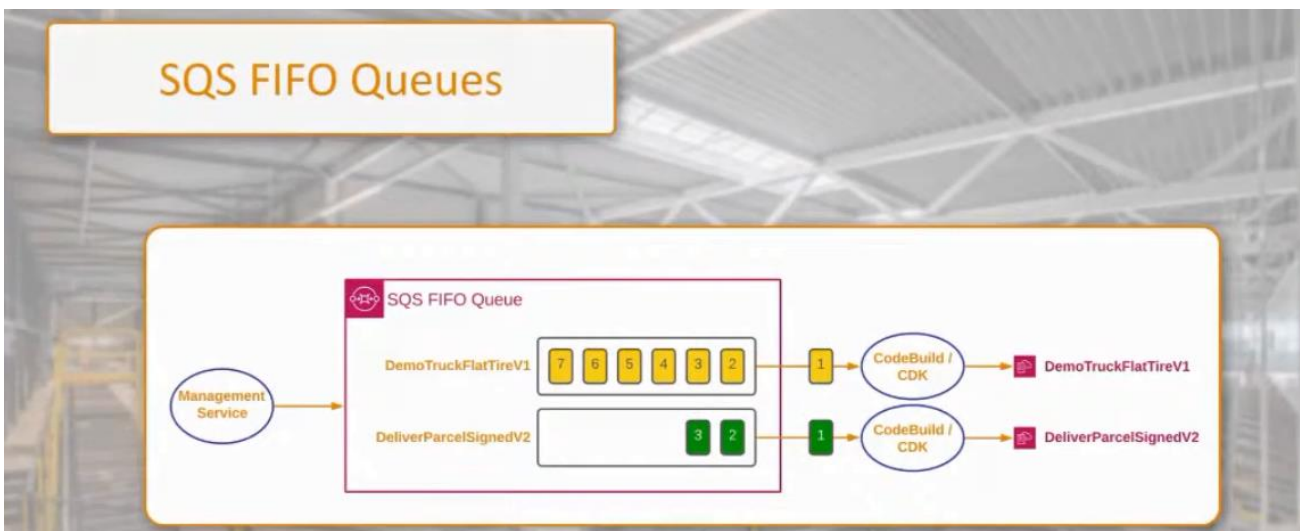
When a user makes a UI request via the Management UI, the Management Service authorizes the user and then validates their request.



When the user and request (event) are validated, the Management Service then generates the SQS message and puts it on the FIFO queue.

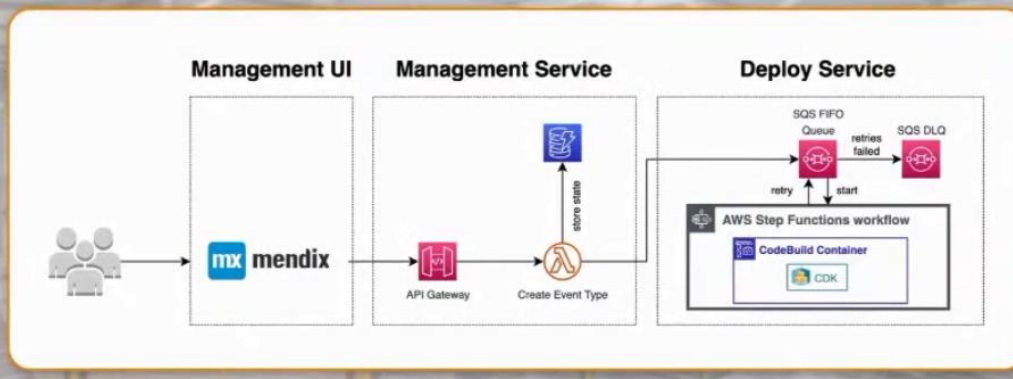
SQS & Step Functions

Let us see how SQS and Step Functions are used to trigger CodeBuild.



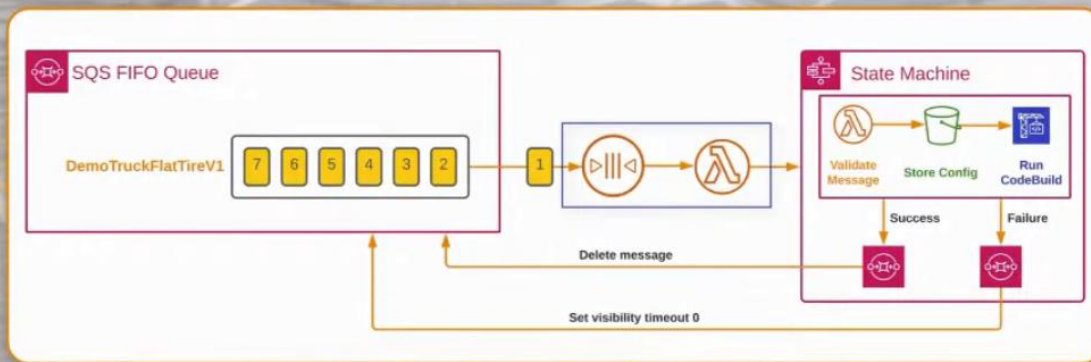
Only one operation can be performed on a single CF Stack at a time, so we need a method that can allow multiple operations to be executed on the same stack in order. We can use an SQS FIFO queue that are based on the concept called groupIds and messages placed on the same queue having the same groupIds are guaranteed to be processed in order. We can use the unique stack names to serve as the groupId value for this use-case.

SQS FIFO Queues

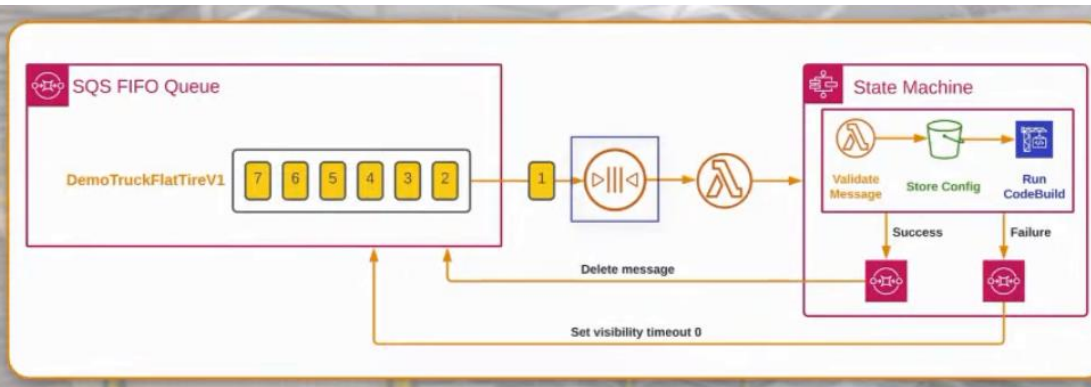


The last part is how to trigger CodeBuild from an SQS queue. Since SQS is a Pull-based mechanism and Step Functions is a Push-based mechanism, we need some sort of glue to connect the them together.

SQS FIFO Queues

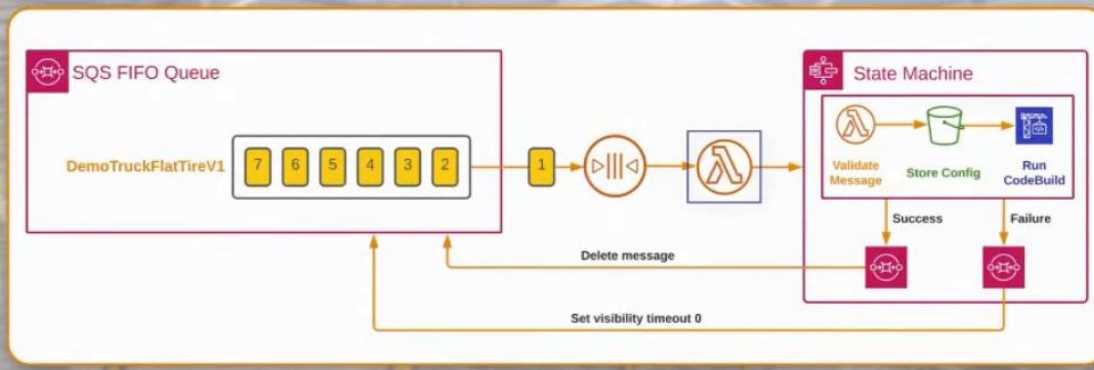


We can use a Lambda function with an event-source mapping as above.



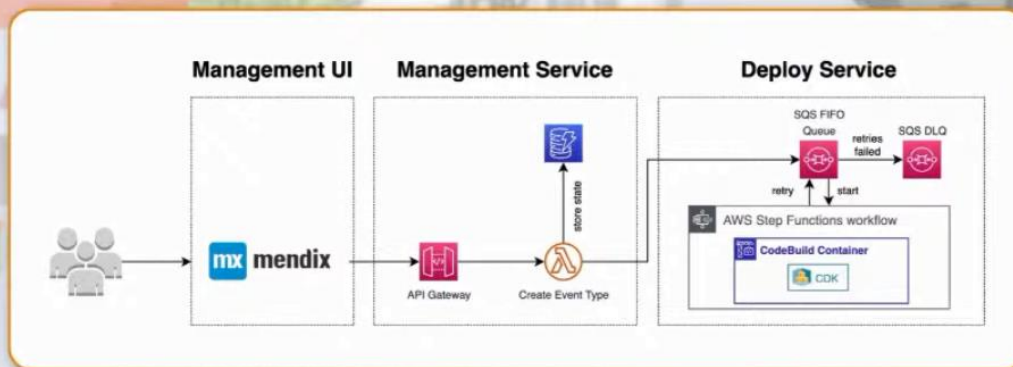
The event-source mapping continuously polls the queue and when new events arrive on any groupId, it invokes the Lambda function.

SQS FIFO Queues



The Lambda function then forwards the message to the state machine/Step Function which runs our CodeBuild process. When all the steps in the state machine succeeds, the SQS message is deleted from the FIFO queue and the next operation can start. If a state machine step fails, the message is returned to the queue and its visibility timeout is set to zero so that it will be retried.

Wrap-up



That is the entire flow.

Wrap-up

Stack name	Status
ProducerCeplaRetaillocationsV1	✔ CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV2	✔ CREATE_COMPLETE
ProducerFsnExecutioneventV1	✔ CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV1	✔ CREATE_COMPLETE
ProducerOompdDdaExecutioneventsDdV1	✔ CREATE_COMPLETE
ProducerOompdpritemrouteupdateV1	✔ CREATE_COMPLETE
ProducerOompdProtemrouteupdatesV1	✔ CREATE_COMPLETE

Physical ID	Type	Status
90147340-d287-11ec-988b-021b261d06b7	AWS::CDK::Metadata	✔ CREATE_COMPLETE
ProducProdu6IBtSUvh2nhK	AWS::ApiGateway::Model	✔ CREATE_COMPLETE
w4e4xi	AWS::ApiGateway::Resource	✔ CREATE_COMPLETE
Produ-Produ-1OTOWAT1SKIPM	AWS::ApiGateway::Method	✔ CREATE_COMPLETE
prwvnyk	AWS::ApiGateway::RequestValidator	✔ CREATE_COMPLETE

