Allan Simbajon
CIDM 6330-70
Dr. Babb

Applying Python Architectural Design to The Scents Company

Problem Identification

Implementing the Architectural Patterns in Python (APP) book to practical application such as solving a domain problem close to one's own experience is the essence of this task. The refactoring approach opens flexibility when writing scripts, making room for expansion and improvement in the future without breaking the system. The idea of compartmentalizing functionalities into sections makes sense, especially when dealing with only a portion of the code, such as adding more features or replacing an existing application with something better. The principle of Test Driven Development (TDD) allows for a resilient code because it is essentially writing a script that has already gone through a series of tests. Efficiency also plays an important role in TDD, with tests already being written and available when the code is complete. The following domain demonstrates how the APP structure is applied to the Scents Company, a company aiming to be the main producer of scents incorporated in the filmmaking and videogaming entertainment industry.

The entertainment industry, specifically filmmakers and video game developers, have been tremendously successful in delivering highly entertaining and realistic media, but where they still struggle is to incorporate the sense of smell with their movies and games for their audience to enjoy a fully immersive experience. No doubt that Hollywood and game creators have tried over the years, but to this day scent is still not mainstream in movies and games.

The idea has been around for ages, and there were some serious attempts that unfortunately failed along the way. The machines were complicated and expensive powered by compressed air containing various scents that were manually activated in certain movie scenes. The system was found to be rather noisy. Some areas of the theater were more saturated than others, and some viewers complained that the odors released were delayed. Clearly the technology needed was not available at that time.

Moviemakers have successfully satisfied the audiences' craving for realistic visual effects, as well as astounding sounds. Lately, innovators are incorporating body suits to deliver realistic sense of touch to the experience. This domain is clearly lucrative to filmmakers and video game makers, and the demand is definitely not going away.

Innovators incorporate scents into their own products in small scale. What would help is to develop a software system that could easily be programmed to calculate complex chemical mixtures, produce the odors cheaply, and be the major producer of scents for the entire entertainment industry. With economies-of-scale in mind, odors can be produced inexpensively, and with ease of use, scents in movies and video games can be mainstream. Another required part of the system is a software that runs and analyzes the logistics of the business demand. By running analytics, the software should be able to identify the most common scents applicable to movies and games, mass-produce them, and offer them cheaply to game creators and moviemakers. Also, startup innovators use their virtual reality goggles to dispense odor, so flexible scent dispensing designs must be kept in mind to offer to such customers. For movie theaters, scents can be released from the audiences' seats. Movie theaters are not required to

undergo major infrastructure upgrade to make this work. Each theater seat can be attached with scent-releasing device connected to vials of odor-producing chemicals, and the entire mechanism is controlled remotely via Bluetooth. A software can manage the release of odors timely for the entire theater as the movie is viewed. As audiences become more particular and critical, the software system must be easily updated to meet the changing demand. For instance, the sensation of ocean breeze scent varies per person, but running analytics from surveys collected can zero in on the most accepted scent of the ocean breeze and update the scent accordingly.

The scents are produced in the lab but have not been introduced for market research. There are expensive suppliers and cheap suppliers. The expensive supplier uses premium oils, while the cheap supplier uses lower grade oils, but both suppliers can produce the same list of scents (e.g., sea breeze, night club, jungle). The audience does not know which scents are produced by either supplier. The audience decides which scents are accepted and believable. The idea is to produce highly accepted scents as cheap as possible. The market research comes back with percentages of favorability per type of scent. The parameters mainly focus on the cheap supplier. If a cheap scent is 40% favored or higher, go with the cheap supplier. Otherwise go with the expensive supplier.

Collected data would look something like this:

scents	expensive supplier	cheap supplier	expensive costs	cheap costs
sea breeze	40%	60%	$ 25.00	$ 15.00
nightclub	35%	65%	$ 18.00	$ 16.00
jungle	70%	30%	$ 30.00	$ 25.00
forest	50%	50%	$ 22.00	$ 14.00
busy street	49%	51%	$ 27.00	$ 20.00

The business foundations mentioned above need to be fulfilled to start the Scents Company business, but once all these foundations have been established, the business will mainly focus on selling the scent products to customers, and this is when the principles from the Architectural Patterns in Python (APP) can be implemented. Fulfilling orders based on inventory and shipment schedule is the focus hereafter, and the allocation system from the APP book will provide a system that manages the Scents Company's inventory and ordering system. Filling orders require proper inventory management, shipment control, and flawless timing to avoid out of stock mishaps, filling the wrong products, and shipment timing issues. Scents Company requires an allocation system that produces a list of its products, a batch system that manages its products, and a reliable ordering system for its customers. All these data should be stored in a database. The system must manage all these areas at all times. There should be a mechanism to allocate product batches from the inventory to fill customer orders. A quantity change functionality should allow for updating batch quantity, should customers change their order logistics. The system should be able to handle product recordkeeping such as using stock-keeping unit (SKU), quantity, and estimated time of arrival (ETA). The system should also be able to handle sending notifications both internally for employees and externally for customers.

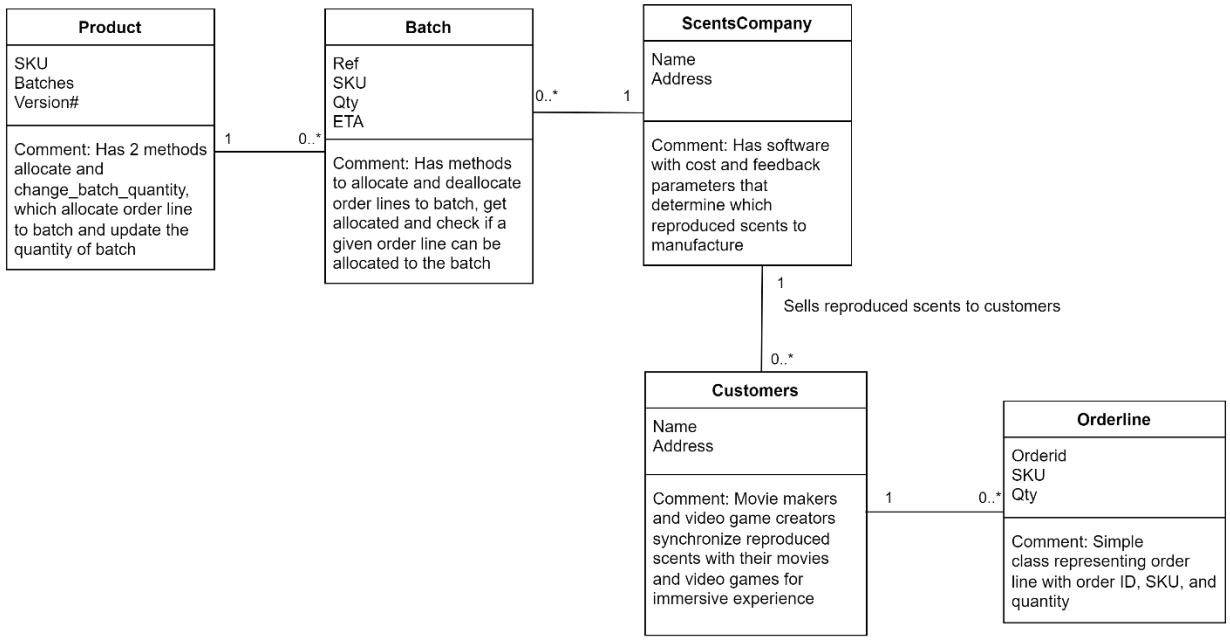
The APP book also emphasizes the importance of flexibility in the system to make room for improvements in the future. Encapsulation of domain logic is essential to encapsulate the domain logic and keep it separate and organized in a single module. Flexibility and extensibility allow for new commands and events to be added to the system without modifying the existing code. Testability allows

for testing scripts in isolation, making it easier to write unit tests for the domain model. Separation of concerns helps to make the code more modular and easier to maintain.

The use of a message bus improves modularity, scalability, and flexibility of the application for the Scents Company. It provides a centralized location to handle messages and decouples the handling of messages from the rest of the application's logic, which can simplify the code and make it more maintainable. This makes it easier to test and debug the message handling code. The use of a message bus also enables the implementation of complex domain logic that involves coordinating multiple events and commands, which can be difficult to do in a purely procedural application. It provides a way to handle events and commands asynchronously by queuing them up and processing them one at a time. It provides a way to handle any new events that are generated during the handling of events or commands.

Domain Modeling

Movies and Video Games Scents Company Revised Domain Model



Use Modeling

To apply the APP architecture in the Scents Company's allocation system, the company's inventory is set up as the Batch class, the list of products is the Product class, and the customer orders is the dataclass Orderline. The Product class represents a product and contains a list of batches of that product, a version number, and a list of events that have happened to the product. It has two methods, namely allocate and change_batch_quantity, which allocate an order line to a batch and update the quantity of a batch, respectively. The Batch class represents a batch of a product and contains a reference, SKU, quantity, ETA, a set of allocations, and methods to allocate and deallocate order lines to the batch. It also has methods to get allocated and available quantity and to check if a given order line

can be allocated to the batch. The OrderLine dataclass is a simple class that represents an order line with an order ID, SKU, and quantity.

The APP book's architecture provides event and command handlers for the Scents Company's allocation system which creates batches of products, allocates orders to available batches, and changes the quantity of a batch. The add_batch function creates a new batch or adds a batch to an existing product, the allocate function allocates an order to a batch, the reallocate function reallocates an order that was previously deallocated, and the change_batch_quantity function changes the quantity of a batch. The send_out_of_stock_notification function sends a notification when a product goes out of stock, the publish_allocated_event function publishes an allocated event to a message broker, and the add_allocation_to_read_model and remove_allocation_from_read_model functions add or remove an allocation from the read model. The script also defines two dictionaries, EVENT_HANDLERS and COMMAND_HANDLERS, which map event and command types to their respective handlers. When an event or command is received, the appropriate handler function is called to handle it.

Functional Test Plan

Unit Tests

Test Batches

This test contains a series of unit tests that handles order allocation. It tests for various methods of the Batch class in allocation.domain.model module and its ability to allocate and deallocate order lines.

The Batch class represents a group of items with a particular SKU that share an ETA. The class has methods to allocate and deallocate order lines, and to check if a given order line can be allocated to the batch based on its available quantity.

The unit tests check if the following functionalities of the Batch class work as expected. Allocating an order line reduces the available quantity of the batch by the quantity of the order line. The can_allocate logic returns True if the available quantity of the batch is greater than or equal to the required quantity of the order line, and False if not. The can_allocate logic returns False if the SKU of the order line does not match the SKU of the batch. Allocation of the same order line multiple times has no effect on the available quantity of the batch.

The first test called test_allocating_to_a_batch_reduces_the_available_quantity tests whether allocating a quantity to a batch reduces the available quantity of that batch by the same amount.

The next three tests labeled test_can_allocate_if_available_greater_than_required, test_cannot_allocate_if_available_smaller_than_required, and test_can_allocate_if_available_equal_to_required test whether a batch can allocate a line of a certain quantity based on its available quantity.

The fourth test named test_cannot_allocate_if_skus_do_not_match tests whether a batch can allocate a line only if the SKU of the batch matches that of the line.

The fifth test called test_allocation_is_idempotent tests whether allocating the same line to a batch twice will only reduce the available quantity of that batch once.

Overall, the tests ensure that the Batch class can properly allocate and deallocate order lines based on its available quantity and SKU.

Test Handlers

This test contains a set of unit tests that manages product allocations. The module handles commands related to product batches such as creating a batch, allocating from a batch, and changing the quantity of a batch. The unit tests ensure that the module works correctly in different scenarios.

The module defines a fake implementation of a repository and a unit of work, which are used to isolate the tests from the actual persistence mechanism. The module also defines a fake implementation of a notification service that records the notifications sent during the tests. The FakeRepository class provides an in-memory storage solution for products and their associated batches (i.e., groups of inventory units with the same SKU and creation date). The FakeUnitOfWork class provides an in-memory implementation of the unit of work pattern for coordinating changes to the repository. It includes a committed flag to indicate whether the changes have been persisted. The FakeNotifications class provides a simple notification service that records messages sent to each destination in a dictionary.

The module defines three classes, namely TestAddBatch, TestAllocate, and TestChangeBatchQuantity. Each of these classes contains one or more test methods that test the functionality of the corresponding commands. For example, TestAddBatch has two test methods test_for_new_product and test_for_existing_product. These methods create batches using the CreateBatch command and verify that the batch was created correctly. TestAllocate tests the Allocate command for reserving inventory units for a new order and updating the available quantity of the associated batch. TestChangeBatchQuantity tests the ChangeBatchQuantity command for adjusting the available quantity of a batch and reallocating inventory units as necessary.

The test methods use the bootstrap_test_app function to obtain an instance of the module's business logic. The function initializes the necessary dependencies (such as the fake repository and unit of work) and returns an instance of the business logic. A bootstrap_test_app function creates a test version of the application by calling the bootstrap.bootstrap function with the appropriate parameters (including the fake repository, unit of work, and notification classes).

The tests use the pytest framework to assert that the application behaves as expected. For example, the TestAllocate.test_allocates method creates a new batch, allocates some inventory units for an order, and verifies that the available quantity of the batch has been reduced accordingly. It uses the assert statement to check that the available_quantity attribute of the batch matches the expected value.

Overall, the module provides unit tests that test the functionality of the product allocation module. The tests ensure that the module works correctly in various scenarios and that it correctly handles errors and sends notifications when necessary.

Test Product

This module creates instances of the Product, Batch, and OrderLine classes and checks the behavior of their methods in different scenarios. Each test method creates an instance of Product and a corresponding OrderLine object, and then calls the allocate method on the Product instance, which attempts to allocate inventory from the product's batches to the order line. The tests assert various

properties of the Product and its batches before and after the allocation, as well as the creation of certain events related to the allocation process.

The test_prefers_warehouse_batches_to_shipments checks that the allocation system assigns an order line to an in-stock batch before considering a shipment batch. It creates a product with two batches of the same SKU, one with an ETA of None (representing in-stock inventory) and another with a future ETA (representing inventory that will be received from a supplier). It allocates an order line to the product and asserts that the allocation comes from the in-stock batch, leaving the shipment batch untouched.

The test_prefers_earlier_batches checks that the allocation system assigns an order line to the earliest batch that can fulfill it. It creates a product with three batches of the same SKU, each with a different ETA. It allocates an order line to the product and asserts that the allocation comes from the batch with the earliest ETA, leaving the other two batches untouched.

The test_returns_allocated_batch_ref checks that the allocate method of the Product class returns the reference of the batch that was allocated to the order line. It creates a product with two batches of the same SKU, one with an ETA of None and another with a future ETA. It allocates an order line to the product and asserts that the return value of the allocate method is the reference of the in-stock batch.

The test_outputs_allocated_event checks that an allocated event is produced when a batch is allocated to an order line. It creates a product with one batch of a particular SKU and allocates an order line to the product. It asserts that the last event in the product's event list is an allocated event with the correct properties.

The test_records_out_of_stock_event_if_cannot_allocate checks that an OutOfStock event is produced when there is no batch that can fulfill an order line. It creates a product with one batch of a particular SKU and allocates an order line to the product to exhaust the batch's available quantity. It attempts to allocate another order line for the same SKU to the product and asserts that the return value is None (since the allocation failed) and that the last event in the product's event list is an OutOfStock event with the correct SKU.

The test_increments_version_number checks that the version number of the Product instance is incremented when an order line is successfully allocated to a batch.test_increments_version_number. It creates a product with one batch of a particular SKU and sets its version number to 7. It allocates an order line to the product and asserts that the product's version number is incremented to 8.

Integration Tests

Test Email

This test is a test case for the test_out_of_stock_email function that verifies whether an email notification is sent when a product SKU goes out of stock. The @pytest.fixture decorator is used to define a fixture function called bus that is used to set up the test environment. This fixture initializes the bus object, which is responsible for handling commands and events. It uses the bootstrap function to set up the application, including initializing the ORM (Object-Relational Mapping) and setting up the unit of work. The unit of work is responsible for coordinating the transaction and ensuring data consistency

during the process of handling commands and events. The fixture also sets up email notifications and a message publisher.

The get_email_from_mailhog function is used to retrieve the email message containing the specified SKU from the MailHog API. This function sends a GET request to the MailHog API to retrieve all email messages, filters them based on the SKU, and returns the first email message that matches the filter.

The test_out_of_stock_email function is the main test case that verifies whether an email notification is sent when a product SKU goes out of stock. It uses the bus fixture to handle two commands, namely CreateBatch and Allocate, to create a new batch with a random SKU and allocate 10 units of the SKU. After the allocation is complete, the function retrieves the email message containing the SKU using the get_email_from_mailhog function and asserts that the From, To, and Data fields of the email message contain the expected values. If the assertions pass, the test case is considered successful.

Overall, this module is a part of a larger application that manages product inventory and notifies the stock management team when a product goes out of stock. The script verifies the correct functioning of the notification system in a test environment.

Test Repository

This module defines an abstract repository interface, AbstractRepository, and a concrete implementation of the repository using the SQLAlchemy ORM, SqlAlchemyRepository. The abstract repository interface declares methods for adding a product to the repository, getting a product by SKU, and getting a product by batch reference. The implementation of these methods is left to concrete subclasses of AbstractRepository. The implementation of _add adds the given product to the repository session. The implementation of _get queries the session for a Product object with the given SKU. The implementation of _get_by_batchref queries the session for a Product object that has a batch with the given batch reference.

This module tests whether the get_by_batchref method of the SqlAlchemyRepository class in the allocation.adapters.repository module can retrieve the correct Product object based on the reference of one of its batches. The SqlAlchemyRepository implementation provides concrete implementations of the _add, _get, and _get_by_batchref methods using SQLAlchemy queries. First, it creates a few Batch and Product objects and adds them to the SqlAlchemyRepository instance. Then it asserts that calling get_by_batchref with the reference of one of the batches returns the corresponding Product object, and that calling it with the reference of another batch returns a different Product object. The SqlAlchemyRepository implementation also maintains a set of products that have been added to or queried from the repository.

Test Unit of Work

This module defines the following four test cases, each with a unique purpose, and a set of helper functions to facilitate the tests.

The test_uow_can_retrieve_a_batch_and_allocate_to_it checks whether a batch of products can be retrieved from the database, and whether an order line can be allocated to the batch. It does this

by creating a new batch in the database, retrieving it using a unit of work (uow), allocating an order line to the batch, and then verifying that the correct batch was allocated to the order line.

The test_rolls_back_uncommitted_work_by_default checks whether uncommitted changes to the database are rolled back when a uow is exited without committing. It does this by creating a new batch in the database using a uow, and then verifying that the batch was not actually created in the database.

The test_rolls_back_on_error checks whether uncommitted changes to the database are rolled back when an error occurs during a uow. It does this by creating a new batch in the database using a uow and then raising an exception. It then verifies that the batch was not actually created in the database.

The test_concurrent_updates_to_version_are_not_allowed checks whether concurrent updates to a product version in the database are properly handled. It does this by creating a new product batch in the database, and then allocating two order lines to the product batch concurrently using separate threads. The test verifies that only one of the order lines is successfully allocated to the batch, and that an exception is raised for the other order line due to a concurrent update to the product version.

This module also defines the following several helper functions to facilitate the tests. The insert_batch inserts a new batch of products into the database.vThe get_allocated_batch_ref retrieves the batch reference for an order line that has already been allocated to a batch. The try_to_allocate attempts to allocate an order line to a batch using a new uow. The script also defines several constants, such as random_sku, random_batchref, and random_orderid, which are used to generate random values for testing purposes.

Test Views

This module contains the following two test functions that test the allocation functionality.

The test_allocations_view tests the views.allocations function. It first sets up a bus object using the bootstrap function from the allocation module and a SQLite session factory. The bus object is used to execute a series of commands that create batches of items and allocate orders for those items. After executing the commands, the test function calls the views.allocations function with an order ID and the unit of work object associated with the bus. The function returns a list of dictionaries containing information about the allocated batches for the given order. The test function then asserts that the returned list is equal to an expected list of dictionaries containing information about the allocated batches for the given order.

The test_deallocation tests the deallocation of batches. It follows a similar setup as the first test function, but additionally changes the quantity of one of the created batches using the commands.ChangeBatchQuantity command. The test function then calls the views.allocations function again with the same order ID and unit of work object, and asserts that the returned list of allocated batches is equal to an expected list. In this case, the test expects only one batch to be allocated since the quantity of the other batch was reduced.

The script also defines a fixture called sqlite_bus which sets up a bus object for use in the test functions. The yield keyword is used to return the bus object to the test function and clear the mappers after the test function completes.

End-to-End Tests

Test API

The first test test_happy_path_returns_202_and_batch_is_allocated checks the successful allocation of a batch of products to an order. The test uses the api_client module to generate a random order ID and two random product SKUs. It creates three random batch references for the two SKUs. It adds the two product SKUs to the database along with their respective batches. It makes a request to allocate a quantity of the first product SKU to the order ID. It checks that the request returns a status code of 202 (Accepted). Lastly, it checks that the correct batch has been allocated to the order ID.

The second test function test_unhappy_path_returns_400_and_error_message checks that an error message is returned when an invalid SKU is requested. This test also uses the api_client module to generate a random order ID and an unknown product SKU. It makes a request to allocate a quantity of the unknown product SKU to the order ID, with the expect_success flag set to False to ensure that the request fails. It checks that the request returns a status code of 400 (Bad Request). It checks that the error message in the response body matches the expected error message. It checks that a subsequent request to retrieve the allocation for the order ID returns a status code of 404 (Not Found).

Overall, these tests ensure that the API client functions as expected, correctly allocating product batches to orders and returning appropriate error messages when invalid SKUs are requested.

Test External Events

The test_change_batch_quantity_leading_to_reallocation tests the scenario where a change in the quantity of an allocated batch leads to the reallocation of the order to a different batch. The test case begins by generating a random order ID and SKU, and two random batch references named earlier_batch and later_batch. The API client is then used to add two batches to the system with the specified references, SKUs, and quantities, and estimated times of arrival. The API client is then used to allocate 10 units of the SKU to the order with the generated order ID. The allocation is verified by checking the batch reference of the allocated batch using the API client. Next, the Redis client subscribes to a Redis pub/sub channel named line_allocated. The test then publishes a message to the Redis pub/sub channel named change_batch_quantity to change the quantity of the allocated batch with the earlier_batch reference to 5 units. The test then waits for a message on the line_allocated channel indicating that the order has been reallocated to a different batch. This is done using the Retrying class from the tenacity library to retry the subscription until a message is received or a timeout of 3 seconds is reached. When a message is received, it is appended to a list of messages and the order ID and batch reference of the message are verified.

Testable Implementation

Adapters

Notifications

This module defines two classes that handle notifications in the allocation system. The first class is an abstract base class named AbstractNotifications that defines a single abstract method send(destination, message). The abc module is used to declare this class as an abstract base class. The second class is named EmailNotifications and it extends the AbstractNotifications class. It provides an implementation for the send method that sends an email notification to the specified destination with the given message. The constructor of EmailNotifications takes two optional arguments, namely smtp_host and port, which default to the values obtained from the config module. This module is imported at the beginning of the script, and provides configuration settings for the allocation system, including email host and port. Inside the constructor, an SMTP server object is created using the provided smtp_host and port values. The noop method is called on the server object to check if the connection to the SMTP server is valid. The send method of EmailNotifications creates a message with a subject of "allocation service notification" and the specified message as the body. The message is then sent using the SMTP server object's sendmail method, with the from_addr set to allocations@example.com and the to_addrs set to the specified destination. Finally, the first line of the script includes a pylint comment to disable the "too-few-public-methods" warning for the module. This warning is raised by pylint when a class has too few public methods. Disabling it in this case indicates that it is intentional and not a problem.

ORM

This module defines database tables and mappers for SQLAlchemy ORM. It starts with the necessary imports, creates a metadata object to store table and mapper information, and sets up logging. There are four tables defined in this script, namely order_lines, products, batches, and allocations. The order_lines table stores information about individual order lines, including the sku, quantity, and order ID. The products table stores information about individual products, including the sku and version number. The batches table stores information about batches of products, including a unique reference, the product sku, the purchased quantity, and an estimated time of arrival. The allocations table is a many-to-many join table that associates order lines with batches. There is also an allocations_view table defined, which is not backed by a database table but instead is a view that combines data from the orders, batches, and allocations tables to display allocation information. The start_mappers function is designed to set up the mappings between the domain models and the database tables. It defines three mappers: one for the OrderLine model, one for the Batch model, and one for the Product model. The Product mapper includes a relationship to the Batch mapper and the Batch mapper includes a relationship to the OrderLine mapper via the _allocations attribute. Finally, there is an event listener that is triggered whenever a Product is loaded from the database. It sets the product's events attribute to an empty list, which is used by the domain model to track changes to the Product object.

Redis_eventpublisher

This module provides a function to publish events to a Redis channel. The script first imports necessary modules, namely json for encoding events as JSON, logging for logging, dataclasses for getting an object's field values as a dictionary, and redis for connecting to Redis. The script gets a Redis connection using the configuration settings in config.get_redis_host_and_port. It defines a function named publish(channel, event) that takes a channel name and an event object. In the publish function, it logs the event to be published using the logging module. It then publishes the event to the given Redis

channel by calling r.publish(channel, json.dumps(asdict(event))). Here, asdict from dataclasses module converts the event object to a dictionary and json.dumps serializes it to a JSON string. Finally, the script logs the publishing event using the logging module.

Repository

This script contains the implementation of the AbstractRepository class, an abstract interface for repositories that manage the persistence of domain objects. It also includes a concrete implementation of this interface using SQLAlchemy. The AbstractRepository defines three methods, namely add, get, and _add. The add method takes a Product object and adds it to the repository, while the get method retrieves a Product object from the repository given its SKU. The _add method is an abstract method that needs to be implemented by concrete repositories to define how a Product object is actually added to the repository.

The SqlAlchemyRepository class is a concrete implementation of the AbstractRepository interface that uses SQLAlchemy to persist Product objects. It takes a session object in its constructor, which is an instance of sqlalchemy.orm.session.Session. The _add, _get, and _get_by_batchref methods are implemented using SQLAlchemy queries to add, retrieve, and search for products respectively.

This attribute is defined in the AbstractRepository and is used to keep track of products that have been accessed by the repository during its lifetime. This is useful for implementing caches or other performance optimizations.

Overall, the purpose of this script is to define an abstract repository interface and a concrete implementation of this interface that uses SQLAlchemy to persist Product objects. This allows the application to interact with a persistence layer without needing to know the specific implementation details.

Domain

Commands

This module defines three data classes that represent commands that can be executed in a system. The @dataclass decorator is used to automatically generate a constructor and other methods for each class based on their attributes. The Command class is an abstract base class that is used as a parent class for the other classes. It does not have any attributes or methods of its own and is used only to identify that the other classes are commands. The first command class Allocate has three attributes, namely orderid, which is a string representing the ID of the order being allocated, sku, which is a string representing the SKU of the item being allocated, and qty, which is an integer representing the quantity of the item being allocated. The second command class is CreateBatch and has four attributes, namely ref, which is a string representing the reference ID of the batch being created, sku, which is a string representing the SKU of the item being batched, qty, which is an integer representing the quantity of the item being batched, and ETA, which is an optional attribute representing the estimated time of arrival for the batch. The Optional type hint is used to indicate that this attribute is not required. The third command class ChangeBatchQuantity has two attributes, namely ref, which is a string representing the reference ID of the batch being modified, and qty, which is an integer representing the new quantity of the batch. The script also includes a pylint directive to disable a specific warning about having too few

public methods. This warning is likely related to the fact that the command classes do not have any methods other than the automatically generated ones from the dataclass decorator.

Events

This module defines three data classes that represent events that can occur in a system. The @dataclass decorator is used to automatically generate a constructor and other methods for each class based on their attributes. The Event class is an abstract base class that is used as a parent class for the other classes. It does not have any attributes or methods of its own and is used only to identify that the other classes are events. The first event Allocated has four attributes, namely orderid, which is a string representing the ID of the order that was allocated, sku, which is a string representing the SKU of the item that was allocated, qty, which is an integer representing the quantity of the item that was allocated, and batchref, which is a string representing the reference ID of the batch that the item was allocated from. The second event class Deallocated has three attributes, namely orderid, which is a string representing the ID of the order that was deallocated, sku, which is a string representing the SKU of the item that was deallocated, and qty, which is an integer representing the quantity of the item that was deallocated. The third event class OutOfStock has one attribute sku, which is a string representing the SKU of the item that is out of stock. The script also includes a pylint directive to disable a specific warning about having too few public methods. This warning is likely related to the fact that the event classes do not have any methods other than the automatically generated ones from the `dataclass` decorator.

Model

This module defines two classes, Product and Batch, as well as a dataclass OrderLine and imports a few other classes from the Commands and Events modules. The Product class represents a product and contains a list of batches of that product, as well as a version number and a list of events that have happened to the product. The Product class has two methods, namely allocate and change_batch_quantity. The allocate method takes an OrderLine and tries to allocate it to a batch of the product. If successful, it returns the reference of the batch that was allocated to. If there are no batches that can fulfill the order, an OutOfStock event is added to the list of events for the product. The change_batch_quantity method takes a batch reference and a new quantity and updates the quantity of the specified batch. If the new quantity causes the available quantity of the batch to become negative, the method tries to deallocate orders from the batch until the available quantity becomes non-negative. The Batch class represents a batch of a product and contains a reference, a SKU, a quantity, an ETA, a set of allocations, and methods to allocate and deallocate order lines to the batch. The Batch class also defines methods to get the allocated quantity and available quantity of the batch, as well as a method to check if a given OrderLine can be allocated to the batch. The OrderLine dataclass is a simple class that represents an order line with an order ID, a SKU, and a quantity. The __future__ import enables the use of annotations in function signatures. The unsafe_hash=True parameter in the @dataclass decorator for OrderLine allows instances of the class to be hashable.

Entrypoints

Flask App

This module sets up a Flask web application with three endpoints for managing inventory allocations. The first endpoint namely /add_batch, is a POST request that creates a new batch of inventory. The endpoint expects a JSON payload with the following properties, namely ref, which is a reference string for the batch, sku, the SKU (stock keeping unit) code for the product being added, qty, the quantity of the product in the batch, and ETA (optional), an ISO-formatted date string representing the estimated time of arrival for the batch. The endpoint creates a commands.CreateBatch object from the JSON payload, and passes it to the bus.handle method for processing. If the command is successful, the endpoint returns an HTTP status code of 201 ("Created") with the message "OK". The second endpoint, /allocate, is a POST request that allocates inventory to an order. The endpoint expects a JSON payload with the following properties, namely orderid, a unique identifier for the order being allocated to, sku, the SKU code for the product being allocated, and qty, the quantity of the product being allocated. The endpoint creates a commands.Allocate object from the JSON payload, and passes it to the bus.handle method for processing. If the command is successful, the endpoint returns an HTTP status code of 202 ("Accepted") with the message "OK". If the SKU is invalid, the endpoint returns an HTTP status code of 400 ("Bad Request") with an error message. The third endpoint /allocations/<orderid> is a GET request that retrieves a list of allocations for a given order. The endpoint expects an orderid parameter in the URL. The endpoint calls the views.allocations function to retrieve the list of allocations for the given order from the database. If the order is not found, the endpoint returns an HTTP status code of 404 ("Not Found") with the message "not found". If the order is found, the endpoint returns an HTTP status code of 200 ("OK") with a JSON payload containing the allocations.

Redis Eventconsumer

This module that listens for messages published to a Redis channel called change_batch_quantity. When a message is received, the script parses the message data, creates a command object based on the data, and passes the command object to a bus object that can handle commands. The script is designed to be run continuously as a background process, processing any new messages as they arrive on the Redis channel. The script sets up a logger object to log messages. The logger is named after the current module (__name__), which is the name of the script file. The script creates a Redis client object using the Redis constructor, passing in the host and port parameters from the config module. The main function is the entry point for the script. The function starts by logging a message that indicates the script is starting to listen for messages on the Redis channel. It then calls the bootstrap function to set up the system and get a bus object that can handle commands. The script creates a Redis pubsub object using the pubsub method of the Redis client object. The script subscribes to the change_batch_quantity channel using the subscribe method of the pubsub object. The script then enters a loop that listens for messages on the Redis channel. The loop calls the listen method of the pubsub object, which blocks until a new message is received. When a message is received, the loop calls the handle_change_batch_quantity function to handle the message. The function is passed the message object and the bus object. The handle_change_batch_quantity function logs a message indicating that it is handling the message. It then parses the message data (which is assumed to be in JSON format) using the json.loads method. The function creates a ChangeBatchQuantity command object using the ref and qty fields from the message data. Finally, the function calls the handle method of the bus object, passing in the command object. The script ends by checking if the script is being run as the main program (__name__ == "__main__") and if so, calling the main function to start the script.

Service Layer

Handlers

This moduel defines the event and command handlers for the allocation system. The system receives commands to create batches of products, allocate orders to available batches, and change the quantity of a batch. The event handlers handle events such as when an order is allocated to a batch or when a product goes out of stock. The add_batch function creates a new batch for a product or adds a batch to an existing product. It takes a CreateBatch command and a unit of work object as arguments. The function retrieves the product associated with the SKU of the command and adds the new batch to its list of batches. If the product does not exist, it creates a new one with the given SKU and adds the batch to it. The allocate function allocates an order to a batch of products. It takes an Allocate command and a unit of work object as arguments. The function creates a new OrderLine object from the command and retrieves the product associated with the SKU of the line. If the product does not exist, it raises an InvalidSku exception. Otherwise, it calls the allocate method of the product, which attempts to allocate the order to one of its batches. The reallocate function reallocates an order that was previously deallocated. It takes a Deallocated event and a unit of work object as arguments. The function creates a new Allocate command from the event and calls the allocate function to allocate the order again. The change_batch_quantity function changes the quantity of a batch. It takes a ChangeBatchQuantity command and a unit of work object as arguments. The function retrieves the product associated with the batch reference of the command and calls its change_batch_quantity method to update the quantity of the batch. The send_out_of_stock_notification function sends a notification when a product goes out of stock. It takes an OutOfStock event and a notification object as arguments. The function calls the send method of the notification object to send an email to the stock management team. The publish_allocated_event function publishes an Allocated event to a message broker. It takes an Allocated event and a publish function as arguments. The function calls the publish function with the event type and the event data. The add_allocation_to_read_model function adds an allocation to the read model. It takes an Allocated event and a SQL Alchemy unit of work object as arguments. The function inserts the allocation data into the allocations_view table. The remove_allocation_from_read_model function removes an allocation from the read model. It takes a Deallocated event and a SQL Alchemy unit of work object as arguments. The function deletes the allocation data from the allocations_view table. The script also defines two dictionaries, namely EVENT_HANDLERS and COMMAND_HANDLERS. These dictionaries map event and command types to their respective handlers. When an event or command is received, the appropriate handler function is called to handle it.

MessageBus

This module defines a MessageBus class that provides a way to handle messages in a message-driven architecture. The MessageBus class takes in an instance of a unit_of_work.AbstractUnitOfWork, a dictionary of event handlers, and a dictionary of command handlers. The handle method of the MessageBus class takes a Message object, which can either be a commands.Command or an events.Event. The MessageBus handles the message by first adding it to a queue and then processing the messages in the queue one by one. For each message, the handle method determines whether it is a command or an event and calls either the handle_command or handle_event method, respectively. The handle_event method looks up the appropriate handler function for the given event type in the event_handlers dictionary and calls it, passing in the event as an argument. If there are multiple handlers registered for the event, all of them are called in the order they appear in the list. If an exception is raised during the handling of the event, it is caught, logged, and the loop continues to the next event

handler. The handle_command method looks up the appropriate handler function for the given command type in the command_handlers dictionary and calls it, passing in the command as an argument. If an exception is raised during the handling of the command, it is caught, logged, and re-raised. Both the handle_command and handle_event methods collect any new events generated during their respective processing and add them to the message queue.

Overall, the MessageBus class provides a way to loosely couple message producers and consumers in a message-driven architecture by abstracting the handling of messages away from the individual components of the system.

Unit of Work

This module manages a unit of work using the SQLAlchemy ORM library. The AbstractUnitOfWork abstract base class defines the interface for a unit of work, which provides a consistent view of the database to the application. It has two abstract methods, namely _commit and rollback. It also has an attribute products, which is of type repository.AbstractRepository. This is a placeholder attribute that is expected to be implemented by subclasses to provide access to the domain objects being persisted. The SqlAlchemyUnitOfWork is a concrete implementation of AbstractUnitOfWork. It takes a session_factory argument that defaults to DEFAULT_SESSION_FACTORY, which is a sessionmaker object that creates a new Session object that is bound to a database engine specified in the config module. The SqlAlchemyUnitOfWork overrides the __enter__ and __exit__ methods to create a new Session and to close the session, respectively. The __enter__ method also initializes the products attribute with a SqlAlchemyRepository object that is bound to the session. The _commit method commits the changes made in the session to the database, and the rollback method rolls back any uncommitted changes made in the session. The collect_new_events method of AbstractUnitOfWork is used to collect any new domain events that are created during the execution of the unit of work. This method yields any events that have not been processed yet from the events list of each Product object in the products attribute of the unit of work.

Bootstrap

This module defines a bootstrap function that initializes a message bus with event and command handlers and injects their dependencies. It also defines a helper function inject_dependencies to perform the injection. The bootstrap function takes several optional parameters. The start_orm indicates whether to start the ORM and is True by default. The uow is an instance of unit_of_work.AbstractUnitOfWork, which is unit_of_work.SqlAlchemyUnitOfWork by default. The notifications is an instance of AbstractNotifications, which is EmailNotifications by default. The publish is a callable that publishes events to a message broker, which is redis_eventpublisher.publish by default. The function first checks if the notifications parameter is None and initializes it with an instance of EmailNotifications if it is. Then, if start_orm is True, it calls orm.start_mappers to start the ORM. The function then creates a dependencies dictionary with the uow, notifications, and publish parameters, which will be injected into the handlers. Next, it uses a dictionary comprehension to create a new dictionary of injected_event_handlers. The keys of this dictionary are the event types, and the values are lists of handlers with their dependencies injected. To do this, it iterates over the EVENT_HANDLERS dictionary in the handlers module, which maps event types to handler functions. For each event type and its handlers, it calls the inject_dependencies function on each handler, passing in the dependencies dictionary. The resulting list of injected handlers is then stored in the injected_event_handlers dictionary

under the corresponding event type. Similarly, the function creates an injected_command_handlers dictionary that maps command types to injected handler functions, using a dictionary comprehension and the inject_dependencies function. Finally, the function creates and returns a messagebus.MessageBus instance with the uow, injected_event_handlers, and injected_command_handlers as parameters. The inject_dependencies function takes a handler function and a dependencies dictionary, and returns a new lambda function that calls the handler with the given dependencies injected as keyword arguments. It does this by inspecting the signature of the handler function, and finding the parameters that match the keys in the dependencies dictionary. It then creates a new dictionary of the matching parameters and their corresponding dependencies and passes this dictionary to the handler using keyword argument unpacking. The resulting lambda function is returned as the injected handler.

Config

This module provides functions to retrieve the necessary configuration parameters from environment variables for the allocation system. The get_api_url returns the API URL of the system. The function retrieves the API_HOST environment variable to get the hostname of the API server. If this environment variable is not set, the default value of localhost is used. The function sets the port to 5005 if the hostname is localhost, otherwise it sets the port to 80. The get_redis_host_and_port returns a dictionary with the Redis host and port configuration. The function retrieves the REDIS_HOST environment variable to get the hostname of the Redis server. If this environment variable is not set, the default value of localhost is used. The function sets the port to 63791 if the hostname is localhost, otherwise it sets the port to 6379. The get_email_host_and_port returns a dictionary with the email server host and port configuration. The function retrieves the EMAIL_HOST environment variable to get the hostname of the email server. If this environment variable is not set, the default value of localhost is used. The function sets the SMTP port to 11025 if the hostname is localhost, otherwise it sets the SMTP port to 1025. The function sets the HTTP port to 18025 if the hostname is localhost, otherwise it sets the HTTP port to 8025.

Views

This module defines a function allocations that takes an orderid string and a SqlAlchemyUnitOfWork object as parameters. It queries the database using the uow object to get the sku and batchref values for all allocations associated with the given orderid. The function returns a list of dictionaries where each dictionary contains a sku and batchref key-value pair. The function is using a context manager (`with`) to automatically handle committing and rolling back the changes made during the database transaction. The uow object is a SqlAlchemyUnitOfWork which is a concrete implementation of an abstract class AbstractUnitOfWork that defines the contract for interacting with the database. The AbstractUnitOfWork class has a method _commit that is used to commit the changes made during the transaction and a method rollback to roll back the changes in case of any errors. The SQL query is a parameterized query that uses named placeholders to avoid SQL injection attacks. The placeholders are replaced with actual values at runtime using a dictionary that maps the placeholder names to their corresponding values. The query returns a result set that is converted to a list of dictionaries using a list comprehension. Each dictionary in the list represents an allocation and contains sku and batchref key-value pairs.

Overall, this module provides a way to retrieve allocation data from the database using a transactional approach with automatic rollback and commit features.

Setup

This module is a basic setup configuration file for the allocation package. It imports the setup function from setuptools and uses it to specify the package name, version, and packages included in the distribution. The name parameter specifies the name of the package, which in this case is allocation. The version parameter specifies the version number of the package, which in this case is 0.1. The packages parameter specifies a list of packages to be included in the distribution. In this case, the only package included is the top-level allocation package.

Validation and Justification

The APP architecture is essential in implementing an efficient, methodical, and organized inventory management and ordering system for the Scents Company. The following enumerate the architectural benefits of the APP architecture for the Scents Company.

Domain

Entity and Aggregate Benefits

The Product and Batch are two entity components. Product represents a product that has a SKU and is composed of one or more Batch instances. It contains the logic to allocate a Batch to an OrderLine and to change the quantity of a Batch. It also keeps track of the events related to it, such as when it is allocated or out of stock. Batch represents a batch of products that share the same SKU and ETA. It contains the logic to allocate and deallocate OrderLine instances, and to determine if it can allocate an OrderLine. It also keeps track of its own state, such as the purchased and allocated quantity, and the ETA.

The Product and Batch components have several benefits, including:

1. Modularity: By encapsulating the logic related to products and batches into separate classes, the code becomes more modular, easier to read and maintain.

2. Reusability: The Product and Batch classes can be reused in other parts of the codebase, which reduces the amount of duplicated code and makes it easier to add new features or modify existing ones.

3. Abstraction: By abstracting away the details of the product and batch logic into separate classes, the rest of the codebase can interact with them through well-defined interfaces, which makes it easier to reason about the code and reduces the likelihood of bugs.

4. Scalability: The Product and Batch classes provide a solid foundation for scaling the system, as they allow for easy management of large numbers of products and batches and provide a clear separation of concerns.

5. Testability: The Product and Batch classes can be easily tested in isolation, which makes it easier to write comprehensive tests for the codebase and catch bugs early on in the development process.

Value Object Benefits

The OrderLine class in the Domain module is a value object because it is immutable and its equality is based on its attributes. This is indicated by the @dataclass(unsafe_hash=True) decorator, which automatically generates the __eq__ and __hash__ methods for the class. Additionally, the OrderLine instances are used as elements of the set _allocations in the Batch class, which further reinforces their immutability.

1. Immutability: As a value object, instances of OrderLine are immutable, which means that their state cannot be changed after they are created. This ensures that they remain consistent throughout the application and prevents accidental modification of their values.

2. Type Hinting: By defining the attributes of the OrderLine class, one can provide better type hinting for the code. This makes it easier for other developers to understand the expected data types and enables better type checking at compile time.

3. Safety: Because OrderLine is immutable, it is safer to use in multi-threaded or concurrent applications, where shared mutable objects can cause data race conditions and other concurrency issues.

4. Clarity: The OrderLine class provides a clear and concise way to represent an order line item, which makes the code more readable and easier to understand. It also enables one to encapsulate the logic related to order lines in a single class, which makes it easier to maintain and modify over time.

Events Benefits

This feature defines three events, namely Allocated, Deallocated, and OutOfStock. These events are used to model the state of Scents Company's inventory system. By using such events, Scents Company can keep track of changes to the inventory in a consistent and scalable way, which can help them make better decisions about how to allocate resources and fulfill orders. The Allocated event can be raised when a batch of products is reserved for an order, the Deallocated event can be raised when a batch is returned to the inventory, and the OutOfStock event can be raised when there is not enough inventory to fulfill an order. By defining events as dataclasses, it makes it easy to create new instances of the event with appropriate data, and to serialize or deserialize them as needed. Additionally, by inheriting from a base Event class, it makes it easier to identify and filter events from other types of messages or data that might be flowing through the system. Overall, this function provides a simple and straightforward way to define and work with events. This can make it easier to understand the data being passed between different parts of the system and can help prevent errors that can occur when passing data between different parts of the system in different formats.

Commands Benefits

By using this pattern, the implementation of an action can be decoupled from the request that triggers it, making it easier to modify, extend and maintain the code. By defining commands as separate classes, it allows for easier separation of concerns in the codebase. Commands represent the intention of the user or system and can be passed around between different parts of the application without needing to know the implementation details. This makes it easier to reason about the behavior of the system and maintain a consistent state. The use of dataclasses also simplifies the creation of instances of these command classes. It also allows for type annotations to be added to each field, which can help with static analysis and debugging. Also, using dataclasses to define commands can provide several benefits:

1. The code is more readable because the attributes of a command are clearly defined and documented in one place.

2. Dataclasses provide a concise way to define immutable objects, which can make the code safer and easier to reason about.

3. Dataclasses can be easily serialized to and deserialized from JSON, which can be useful in distributed systems.

Service Layer

Handler Benefits

This feature defines handlers for various commands and events in the allocation domain of the Scents Company. The main benefit of this script is that it decouples the implementation details of the domain from the rest of the application, making it easier to maintain and evolve over time. This feature benefits the Scents Company Overall because it encapsulates the business logic of the allocation domain in a modular and extensible way. It also allows for greater testability and scalability by separating concerns and minimizing dependencies between different parts of the system.

1. Encapsulation of domain logic: Each function in the script represents a specific operation that can be performed in the domain model, such as adding a new batch or allocating an order line. This helps to encapsulate the domain logic and keep it organized in a single module.

2. Flexibility and extensibility: The use of dictionaries (COMMAND_HANDLERS and EVENT_HANDLERS) allows for a flexible and extensible way to map commands and events to their corresponding handlers. This means that new commands and events can be added to the system without modifying the existing code, by simply adding a new handler function to the appropriate dictionary.

3. Testability: The functions in this script are designed to be testable in isolation, which can make it easier to write unit tests for the domain model. For example, the add_batch function takes a unit_of_work.AbstractUnitOfWork object as an argument, which can be replaced with a mock object during testing.

4. Separation of concerns: The functions in this script are responsible for handling specific types of commands and events and are not concerned with the implementation details of the system as a whole. This separation of concerns helps to make the code more modular and easier to maintain.

Unit of Work Benefits

The benefit of this feature is that it provides the Scents Company a standardized way to manage transactions and access the database within the allocation system using SQLAlchemy, which can help simplify the code and reduce the risk of errors when interacting with the database. This helps to ensure data integrity and consistency. The unit of work pattern can also simplify database code by abstracting away the details of the transaction management and providing a clean interface for working with the database. This function provides a useful abstraction for managing transactions and database access within the allocation domain, and makes it easier to write clean, testable code that interacts with a database.

Message Bus (Internal) Benefits

The benefits of using a message bus include improved modularity, scalability, and flexibility of the application for the Scents Company. It provides a centralized location to handle messages and decouples the handling of messages from the rest of the application's logic, which can simplify the code and make it more maintainable. This makes it easier to add new event and command handlers without modifying the application's core logic and makes it easier to test and debug the message handling code. Additionally, the use of a message bus enables the implementation of complex domain logic that involves coordinating multiple events and commands, which can be difficult to do in a purely procedural application. It provides a way to handle events and commands asynchronously by queuing them up and processing them one at a time. It provides a way to handle any new events that are generated during the handling of events or commands.

1. Decoupling: The MessageBus class allows for loose coupling between message producers and consumers, which can be helpful in large and complex systems. Producers only need to publish messages to the bus, without having to know who will handle them or how.

2. Scalability: A message-driven architecture using a MessageBus can scale horizontally, by adding more message consumers as needed.

3. Separation of concerns: By separating the handling of commands and events, the MessageBus class makes it easier to reason about the different types of messages and their handling. It also allows for different handlers to be used for different types of messages, based on the needs of the system.

Adapters

Repository Benefits

The benefit of having Repository for the Scents Company is that it provides an abstract repository class and a concrete implementation of the repository using SQL Alchemy. This implementation allows for adding and retrieving products by SKU and batch reference. It also tracks the products that have been added or retrieved so that it does not need to fetch them again from the database. This can improve performance and reduce unnecessary database calls. Using an abstract repository class allows for easy swapping of the underlying storage implementation, which is useful when switching from one database to another or when testing the code. It also makes the code more modular and easier to understand, as the implementation details of the database are hidden behind the abstract repository interface.

Event Publisher Benefits

This script defines a publish function that publishes events to a Redis message broker. The benefit of using a message broker like Redis to the Scents Company is that it decouples the different parts of the system, allowing them to operate independently and asynchronously. This means that when an event occurs, it can be published to Redis without waiting for any other part of the system to process it, improving the overall performance and responsiveness of the system. The Scents Company needs a publish function that is reliable, scalable, and flexible.

1. Event-driven architecture: The ability to publish events to a message broker is a key feature of an event-driven architecture. This allows for loose coupling between different parts of a system, where each component can subscribe to relevant events and respond accordingly.

2. Scalability: Redis is a high-performance, distributed data store that can handle large volumes of data and requests. By using Redis as a message broker, this script allows for horizontal scaling of the system, as multiple instances of the application can be deployed to handle different parts of the system.

3. Flexibility: The use of a message broker allows for flexibility in the system design, as new components can be added or removed without affecting the existing system.

4. Reliability: Redis is a robust and reliable data store that can handle high volumes of data and requests. This ensures that messages are delivered reliably to the subscribers, even in high-load scenarios.

5. Logging: The script uses the Python logging module to log messages at the INFO level when events are published. This provides visibility into the system and can be used for debugging or monitoring purposes.

Entrypoints

Web Benefits

Using the Flask web application that interacts with the allocation service, it allows the Scents Company user to add a new batch of products, allocate a specific quantity of a product to an order, and view allocations for a specific order. This function provides a convenient way for users to interact with the warehouse management system using a simple HTTP API.

Event Consumer Benefits

This feature implements a Redis pub/sub mechanism, which allows multiple processes to communicate with each other in a publish-subscribe pattern. The main benefit of this function is that it enables a distributed system to react to events or changes happening in other parts of the system in real-time, without the need for constant polling or waiting for responses. This feature allows for a scalable, asynchronous, and decoupled architecture, where different components of the system can communicate with each other through Redis channels without directly knowing about each other.

Github repository link:

https://github.com/asimbajon1/FinalProject.git