Compulsory Assignment 2016 System Integration Designing a SOA for a B2B Online Retailer

**Team Members:**

*Hassan Mahmud*

*Nicholas Olsen*

*Isaura Almar*

*Michael Dukes*

**GitHub Repository Link:**

<https://github.com/hass123uk/SOA_Compulsory.git>

# Initial Backlog Items:

* As a customer I cannot submit an order unless I am in good credit standing and I have no outstanding bills.
* As a customer I cannot submit an order unless we have enough products in stock.
* As a customer I want to be able to see the shipping charge and a delivery estimate of many orders, before I decide to submit an order
* As a customer I want to submit an order of many different products and receive an order confirmation.
* As a customer I want to register on submitting an order or separately with:
  + Name
  + Register number (ID)
  + Email address.
  + Billing address
  + Shipping address
* As a customer I should be able to change my:
* (This may happen either as a separate action or as part of a new order submission)
* Name
* Email address
* Phone number,
* Billing and Shipping address
* As a customer I should at any time be able to get an overview of my past orders, both completed orders and outstanding orders (incl. the order status).
* As a customer I should be to cancel any order that has not yet been dispatched.
* As a costumer I want to view a / the product(s)
* As an Employee I want to ship an order changing its status and print an invoice of the order.
* As an Employee I want to be able to email the customer his invoice. (Optional)
* As a developer we want to add, read and update products with:
* Registration number (ID)
* Name
* Description,
* Price,
* Category
* Number of items in stock
* An order has: (Incomplete)
* Id
* Shipped
* Customer Id
* List of products (how many of them they ordered)
* Shipped
* An Order has to be able to change the Number of items in stock for a product.

# Service Inventory Blueprint:

## Entity Services

**Name of the service: Product**

**Purpose: Manage product information**

**Service Model: Entity**

**Service Attributes:**

* **Product:**
* Registration number (ID)
* Name
* Description
* Price
* Category
* Number of items in stock

**List of capabilities:**

* Product Get(int id)
* List<Product> GetAll()
* Int Create(Product p)
* Int Edit(Product p)

**Name of the service: Customer**

**Purpose: Manage customer information**

**Service Model: Entity**

**Service Attributes:**

* **Customer:**
* Register number (ID)
* Name
* Email address
* Billing address
* Shipping address
* **Address:**
* Id
* House number
* Street name
* Post code
* City
* Country

**List of capabilities:**

* Customer Get(int id)
* List<Customer> GetAll()
* Int Create(Customer c)
* Int Edit(Customer c)
* Bool ValidateCreditStanding(Customer c)

**Name of the service: Order**

**Purpose: Manage order information**

**Service Model: Entity**

**Service Attributes:**

* **Order:**
* Id
* Customer Id
* List of products
* Enum of OrderStatus
* Order Date
* Estimated delivery date
* Shipping charge
* **Enum for OrderStatus**
* Requested
* Accepted
* Canceled
* Shipped
* Completed

**List of capabilities:**

* Order Get(int id)
* List<Order> GetAll()
* Int Create(Order)
* Int Cancel(int id)
* List<Order> GetAllByCustomerId(int id)

## Task Services

**Name of the service: Order Processing**

**Purpose: Process an order**

**Service Model: Task**

**Service Attributes:**

* **Order Request Response:**
* Enum for customer status
* Order ID
* Shipping charge
* Cost for all products
* Estimated delivery date
* **Customer Status:**
* Accepted
* Rejected

**List of capabilities:**

* OrderRequestResponse CreateOrderRequest(Customer c, List of products)
* Order AcceptOrder(int id)
* Void CancelOrder(int id)

## Utility Services

**Name of the service: Email**

**Purpose: Manage when Emails are sent to the customer**

**Service Model: Utility**

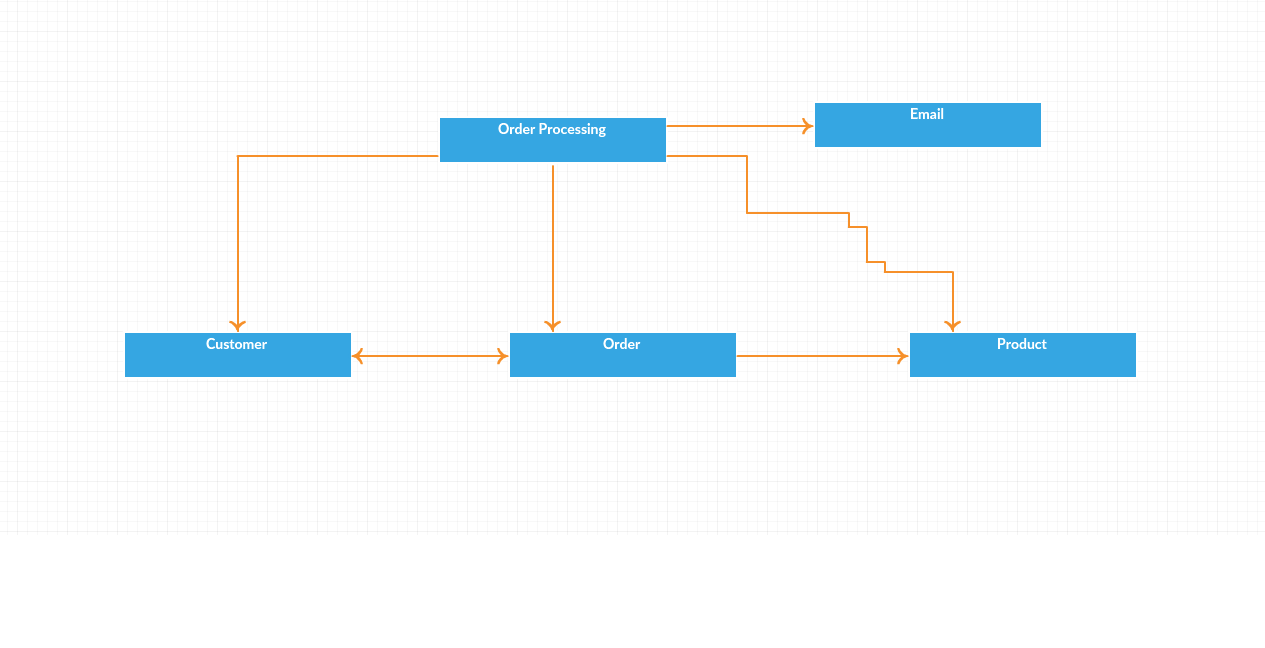
**Service Attributes:**

* **Document:**
* Sender Email
* Recipient Email
* Document text

**List of capabilities:**

* Void Send(Document d)

Service Inventory blueprint diagram



# Inventory Design Patterns:

## Inventory boundary patterns

### Service Normalization

**Problem**

When delivering services as part of a service inventory, there is a constant risk that services will be created with overlapping functional boundaries, making it difficult to enable widespread reuse.

**Solution**

The service inventory needs to be designed with an emphasis on service boundary alignment.

**Implementation**

We were able to accomplish this by starting to create a backlog with user stories. Then we split the user stories into more of a focus of service entities, tasks and utilities. Lastly we defined all the functions of each of these services to look for any services having duplicate functionality. This allows us to have an assurance that service boundaries are and remain well-aligned and introduces ongoing governance effort.

### Logic Centralization

**Problem**

If agnostic services are not consistently reused, redundant functionality can be delivered in other services, resulting in problems associated with inventory renormalization and service ownership and governance.

**Solution**

Access to reusable functionality is limited to official agnostic services.

### Service Layers

**Problem**

Arbitrarily defining services delivered and governed by different project teams can lead to design inconsistency and inadvertent functional redundancy across a service inventory.

**Solution**

The inventory is structured into two or more logical service layers, each of which is responsible for abstracting logic based on a common functional type.

**Implementation**

We went through the background of the specifications to identify the different services that would be needed. We identified 3 services that would belong in an entity service. Those services are, Customer, Product and Order. These services share the same layer due to sharing the same purpose.

In the next layer we identified a task service. This service is the Order Processing service, and does not share the same functionality as the previous layer. This layer contains different functionality of that of entity. The service here requires logic and the use of the different services from the entity service layer.

We also have a utility service in the form of Email. This is a service used for notifications sent to the customer. This can be automated with logic but is separated due to automation and only meant to be a notification.

## Inventory centralization patterns

### Schema Centralization

**Problem**

Different service contracts often need to express capabilities that process similar business documents or data sets, resulting in redundant schema content that is difficult to govern.

**Solution**

Select schemas that exist as physically separate parts of the service contract are shared across multiple contracts.

**Implementation**

We have a location on where you can see how each data object is represented and what data type it consists of. This allows us to share with everyone what to expect when retrieving objects from the API or what the API expects if you are sending something to it and how it is represented in JSON.

## Inventory governance patterns

### Canonical Expression

**Problem**

Service contracts may express similar capabilities in different ways, leading to inconsistency and risking misinterpretation.

**Solution**

Service contracts are standardized using naming conventions.

**Implementation**

We discussed this during the creation of the initial service blueprint on what naming conventions we would use. As we decided to use restful service, we used the standard naming conventions related to RESTful. An example would be post in the entity service to create a new object. We also made sure to avoid giving them names, and leave them generic to increase the reusability unless it is very specific for that service.

# Detailed design and implementation of a single service

The service we decided to implement was the products service and after considering a number of patterns that we could use to influence the way we create the application we agree that the following are relevant and we would like to consider adding them to the application.

## Contract centralization:

**Problem**

Consumer programs can be designed to access underlying service resources using different entry points, resulting in different forms of implementation dependencies that inhibit the service from evolving in response to change.

**Solution**

Access to service logic is limited to the service contract, forcing consumers to avoid implementation coupling.

**Implementation**

In order to ensure that consumers are designed based on the the contract and not the implementation is a natural process in our strategy to developing RESTful services. As the API is considered the service contract that consumers connect to we can ensure that external communication to the service is through the API.  An important point though is to ensure that we keep the underlying business logic within the APIs aligned with the purpose of the service and not allow this business logic to be shared among services.

Another important point is that consumers should have access to the contract and not the implementation so that the developed consumer application is not based on the underlying logic which can cause some undesired coupling.

## Contract denormalization:

**Problem**

Services with strictly normalized contracts can impose unnecessary functional and performance demands on some consumer programs.

**Solution**

Service contracts can include a measured extent of denormalization, allowing multiple capabilities to redundantly express core functions in different ways for different types of consumer programs.

**Implementation**

An example of this within our application can be seen in the Order service. There are two methods that can be considered as denormalized which are GetAll orders and GetAll By CustomerId. While we could filter all orders by customer Id after we retrieve them from the service this might be inefficient if the number of Orders was very large.