# Assignment 1

## Overview

In this assignment you’ll start to implement an “online retailer” Spring Boot application. The application will enable customers to view items on sale, and to add items to their shopping cart.

You’ll define two component classes in this lab:

* A **service** component that lets a customer see what items are available in a catalog, and to add items to their cart (via the repository component, see below).
* A **repository** component that stores items for a customer's shopping cart. Initially, you’ll store items in memory for simplicity. Later in the course, you’ll implement proper persistence by storing this info in a database.

## IntelliJ projects

Starter project: **assignment-student/assignment1**

Solution project: **assignment-solutions/assignment1**

## Roadmap

There are 6 steps in the assignment for this week. Here's a quick summary, see the following pages for all the details:

1. Defining a repository component
2. Defining a service component
3. Writing client code
4. Defining properties in application.properties
5. Defining properties in application.yml
6. Defining profile-specific properties

## Familiarization

Start IntelliJ and open the project **assignment-student/assignment1**. The project contains 4 classes/interfaces to get you started:

* **Item** class – Represents an item in the shop catalog. Each item will have a unique id.
* **CartRepository** interface – Specifies methods for the “repository” component.
* **CartService** interface – Specifies methods for the “service” component.
* **Application** class – This is the usual Spring Boot application class.

## Step 1: Defining a repository component

Define a Spring component class named **CartRepositoryImpl** that implements the **CartRepository** interface. The class will store items in memory initially, for simplicity.

In order to store information about the items in the user’s shopping cart, we suggest defining an instance variable of type **HashMap<Integer,Integer>**. In this map, each entry will hold:

* A key, representing an item id.
* A value, indicating the quantity of that item in the user’s cart.

Implement the **add()**, **remove()**, and **getAll()** methods accordingly, to add/remove/query items in the map. Note the following points:

* In the **add()** method, check to see if the user already has that item in the cart. If so, just increment the quantity for that item (rather than having two separate map entries).
* In the **remove()** method, remove the item completely from the cart. For example, if the customer has 20,000 Swansea City shirts in the cart, and they remove that item, then the **remove()** method should remove the map entry completely (rather than decrementing the count to 19,999).

## Step 2: Defining a service component

Define a Spring component class named **CartServiceImpl**, which implements the **CartService** interface. The class will allow the customer to choose items from a catalog and add them to his/her cart.

Think about where the catalog data should come from… In a real application, this data would almost certainly come from a database, but we’ll adopt a different strategy in this application, for the purposes of reinforcing Spring Boot concepts 😊 ...

So, follow these steps:

* Open the **Application** class, which is annotated with **@SpringBootApplication**. This annotation, amongst other things, means the class is a Spring configuration class, which means you can define **@Bean** methods inside it...
* Define a **@Bean** method named **catalog** that returns a **Map<Integer,Item>**. In this map, the **Integer** key represents an item id, and the **Item** value represents an **Item** instance.
* Implement the **@Bean** method so that it creates a **HashMap**, populates it with some sample items, and then returns it.

You’re now ready to implement the **CartServiceImpl** class. Follow these steps:

* Autowire the catalog bean you just created. This bean will be the source of catalog data for your service bean.
* Also autowire a **CartRepository** bean. This bean will provide persistence to store the customer’s shopping cart.
* Implement the **addToCart()** method. The method takes an item id and a quantity as parameters. The method should check the item id is valid (i.e. it exists in the catalog); if so, add the specified item/quantity to the customer’s cart (i.e. call the **add()** method on the repository bean).
* Implement the **removeFromCart()** method. You can just call the **remove()** method on the repository bean.
* Implement the **getAllItemsInCart()** method. You can just call the **getAll()** method on the repository bean.
* Implement the **calculateCartCost()** method. The method should get all items from the cart (via the repository’s **getAll()** method), multiply the cost of each item by the quantity, and return the total cost.

## Step 3: Writing client code

In the **Application** class, write some client code to get the **CartService** bean and exercise its methods to add/remove items in the customer’s cart, get all items in the cart, and calculate the total cost. Run the application and verify everything is hunky-dory.

## Step 4: Defining properties in application.properties

Define and use some simple properties in **application.properties**. For example:

* Define a property named **contactEmail**, to hold the company's email address.
* Inject this value into your **CartServiceImpl** class.
* Write some client code to verify **CartServiceImpl** picks up the value correctly.

## Step 5: Defining properties in application.yml

Define and use some complex properties in **application.yml**. Here are the properties we’d like you to define:

**Name of property Value Description**

onlineRetailer:salesTaxRate 0.20 Sales tax rate.

onlineRetailer:deliveryCharge:normal 2.50 Delivery charge (GBP) on normal deliveries.

onlineRetailer:deliveryCharge:threshold 3000 Threshold cart value (GBP), for free delivery.

## Inject these values into your CartServiceImpl class. Then write some client code to verify CartServiceImpl picks up the values correctly.

## Step 6: Defining profile-specific properties

Define some properties that have different values depending on the current profile. For example:

**Name of property Value if "development" profile Value if "production" profile**

resources.db H2 Oracle

resources.logs C:\temp\logs\ //PROD\_SERVER/logs/

resources.secure false true

Define a bean class named **ResourcesBean** and inject these properties into it. Then run the application in "development" profile and "production" profile, and verify the correct values are injected into the **ResourcesBean**.