Unit 2 Project: Sustainable Packaging

Preliminaries: Been Here, Got the T-Shirt

You learned about how ATA projects work in the Unit 1 Project.

The Unit 2 project will be largely the same: we'll still have Project Preparedness Tasks (PPTs) and Project Mastery Tasks (MTs) that we check for completion and correctness with Task Completion Tests (TCTs). We'll still let you know when the lessons have covered the material to "unlock" each task. There will still be Project Buddies, Near-peer Reviewers, Office Hours, and the Community Question and Answer (CQA) board in SIM.

There will also be a few differences.

Welcome to the Cloud

In Unit 2 we'll start **deploying our code to AWS**. We will still develop and run code locally, but our code will also live in the cloud! This means that someone or something, a person or another team's code, can interact with your service. Think about pictures that we have on our phones. When they get added to the cloud, we can share them with other people. Other people can now see or use our photos, but they don't need our phones to do it. This is similar to how teams at Amazon make their services available for customers to use. They put their code in a shared place, and then invite customers to use it.

We will make our service available in `beta`. You will see this stage in your new pipeline. Your `beta` stage will use your very own AWS account to deploy your service code to the cloud. On the stage in the pipeline you will see a long number that is a link. If you hover over it, it will say "AWS Account". Clicking on the link will take you to a tool we use at Amazon to access AWS accounts. You can access your account by clicking the button labeled "Console Access". Feel free to take a look around, but not necessary. We'll take another look later in the project.

`Beta` is typically the first place a team will push and share code to. A team needs to have a place that isn't a laptop to test code before it gets released to customers. When code gets deployed to `beta` (we won't worry too much about how that happens just yet, just know it somehow get's moved to a computer in the cloud), it is now available to be used by the whole team. Teams use `beta` to run automated tests, just like our TCTs, to make sure the code does what it is expected to. There are often more stages that teams will use for different purposes, `Prod` being the final stage where code is deployed when it is finally ready to be used by customers.

Git Branching

In Unit 1 you may still have been getting your Git legs underneath you. They might still be wobbly, but we think you are ready to use Git branching to help in developing your project. The fast pace of ATA requires you to keep working on your project while you are waiting for CRs. This is tough if you only work on mainline. So, we've provided a Development Workflow How-To that describes how you can use `git branch` to work on more than one task at a time. The How-To also includes information about `git commit --amend` to keep all your changes in a single commit, and `git rebase` to move your changes back to

`mainline` so you can push. It includes all the troubleshooting steps to resolve the problems you're likely to encounter in Unit 2.

Local Code Coverage Check

Just like in last unit, a couple checks will run on your code after it has been pushed and builds into the version set. One check will verify that your code has been code reviewed by an approved reviewer, and the code that was pushed matches exactly the code that was approved in the CR. We know it may seem silly to put out a whole new code review for what seems like a minor fix, but this is a very typical practice on development teams at Amazon. It helps keep you from moving too fast and making a silly mistake.

The second check will verify that your code has the correct coverage. Meaning, your unit tests are validating that the majority of your code works. In this unit, that number will be 80%. We don't use a number like 100% because that would require us to write unit tests for methods like setters and getters, which isn't a great use of anyone's time. In Unit 2, we have added this check to run locally. It is a best practice to do testing as soon as you can, and discover any issues as early as possible. Having this check run against the code you are developing will help you identify a failure before you push your code!

Ambiguity, Complexity, and Scope.

We're notching up the ambiguity in this unit's project. You'll notice that in the mastery tasks we will not be as explicit about the exact implementation details. Remember, **we value success as a collaborative effort**, so ask and answer questions amongst your peers, on the CQA, or in Office Hours. Coincidentally, developers use these same collaborative skills every day, so you'll be getting good practice for when you graduate. Complexity and scope will also be increased in this project, but not hugely so.

Unit 2 Project Progress and Tracking

Doneness checklist

You're done with this project when you've:

□ All Task Completion Tests are passing

□ Code Coverage Check and CR Verification are passing

□ Submitted Project Reflection response in Canvas

The Problem: Insisting on the Highest Packaging Standards

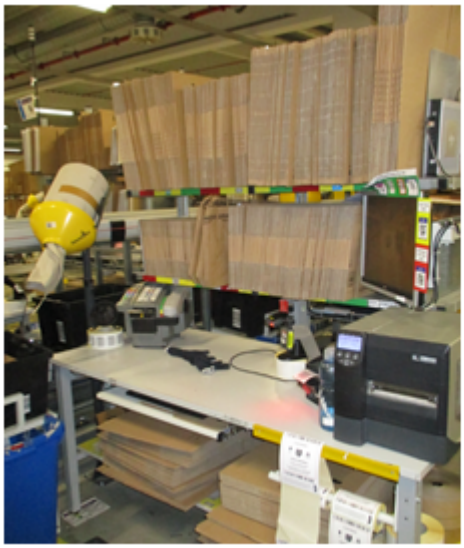
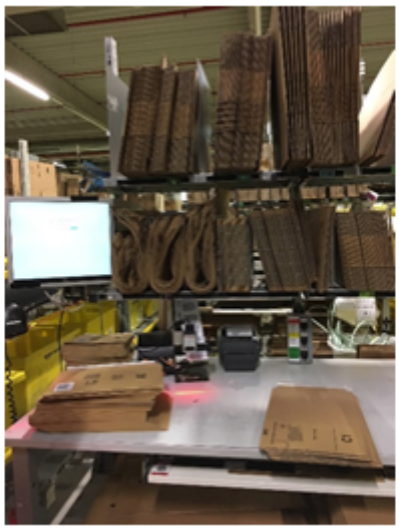
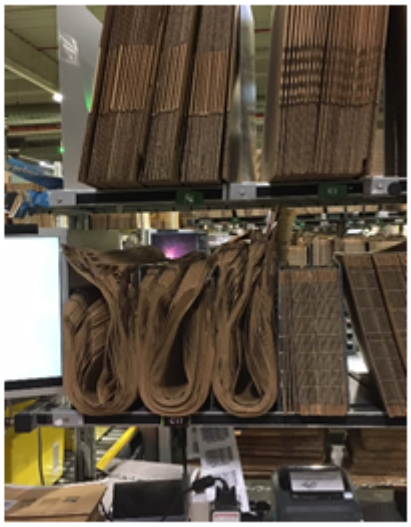
When customers order items from Amazon, they expect world-class service. That includes being responsible for the future of our planet as we deliver value to each customer. As a new member of the Sustainability team, you will help Amazon make environmentally smart decisions to live up to our customers' expectations.

Amazon chooses the best packaging option for each order from many available options. The "best" should not be determined by just its packaging cost, but also its environmental impact. For example, sometimes

the cheapest packaging option is not the best choice because it ends up in a landfill instead of a recycling center, or because it is not usable for returns. For the sake of this project, we will work with "singles" shipments, shipments that contain a single item.

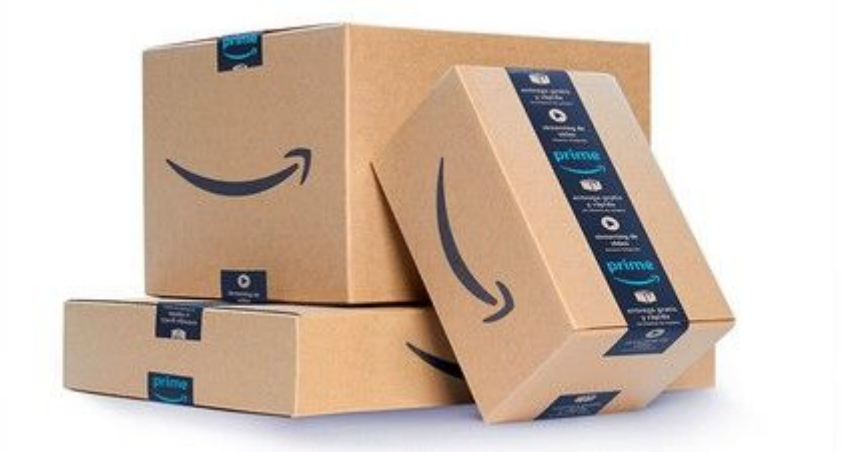
When an item is ready for shipment, it is sent to a packaging associate within the fulfillment center (FC) who places the item in its packaging and sends it to the loading bay. These associates work at "packing stations" that include a wide selection of packaging options. The packaging recommendation system tells the associate which packaging option to use for each order. Packaging options vary by **type** and by **dimension**.

Each packing station has boxes and mailers of different sizes. The difference between a box and a mailer is a difference in **type**, whereas the difference between a small box and a large box is a difference in **dimension**. Refer to these typical packing stations to identify the boxes (the straight stacks) and the mailers (the curved bundles).

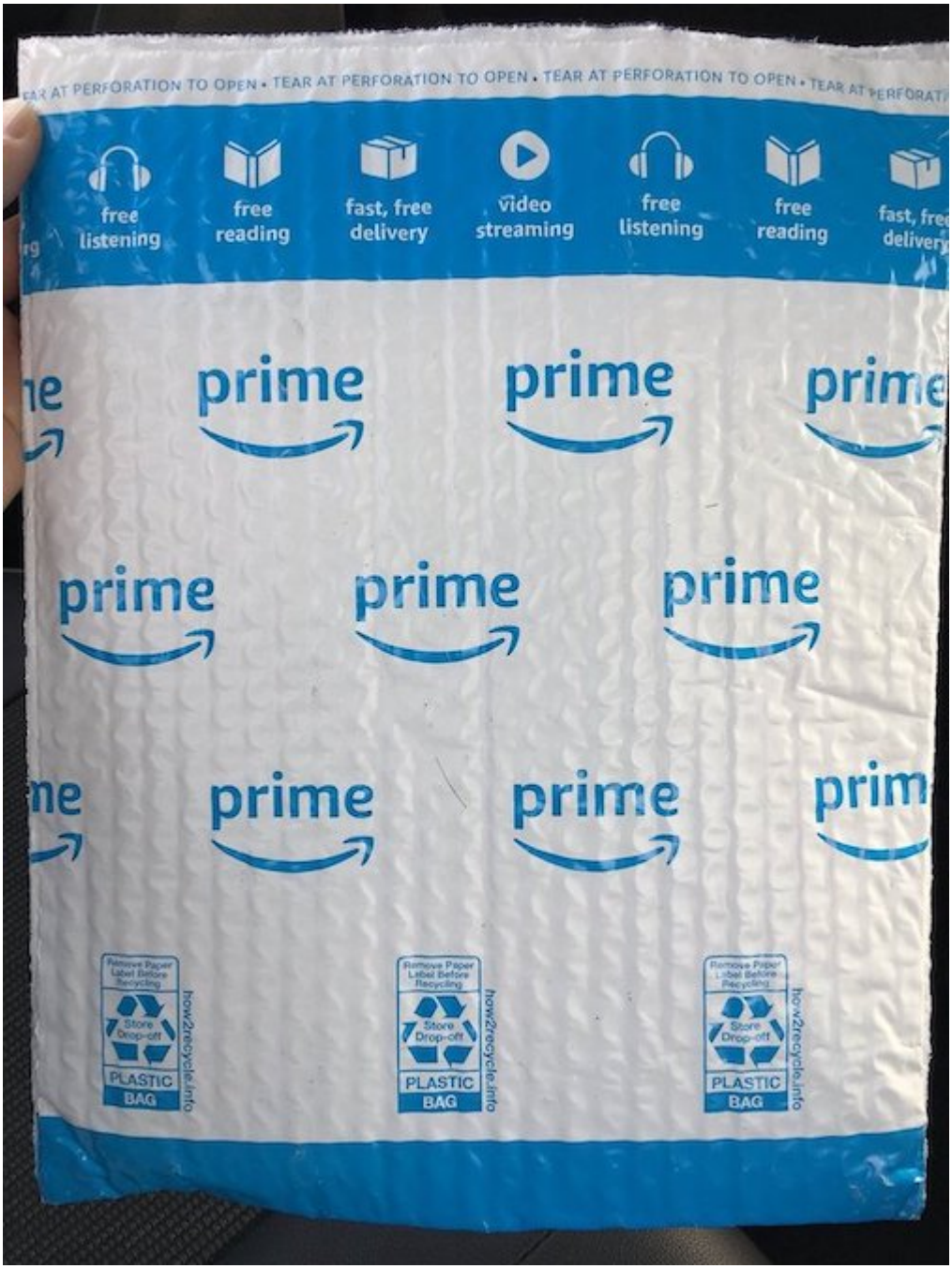
Let's take a closer look at some of the packaging **types** Amazon uses.

Corrugate Box



Typically just called a "box," this is the most common type of packaging. You probably recognize the big Amazon smile logo. They are relatively heavy compared to other packaging types. Some municipalities will recycle them, but the up front material cost tends to be relatively high, and the quality of the cardboard degrades the more times they are recycled. They are reusable as long as they are not damaged too much.

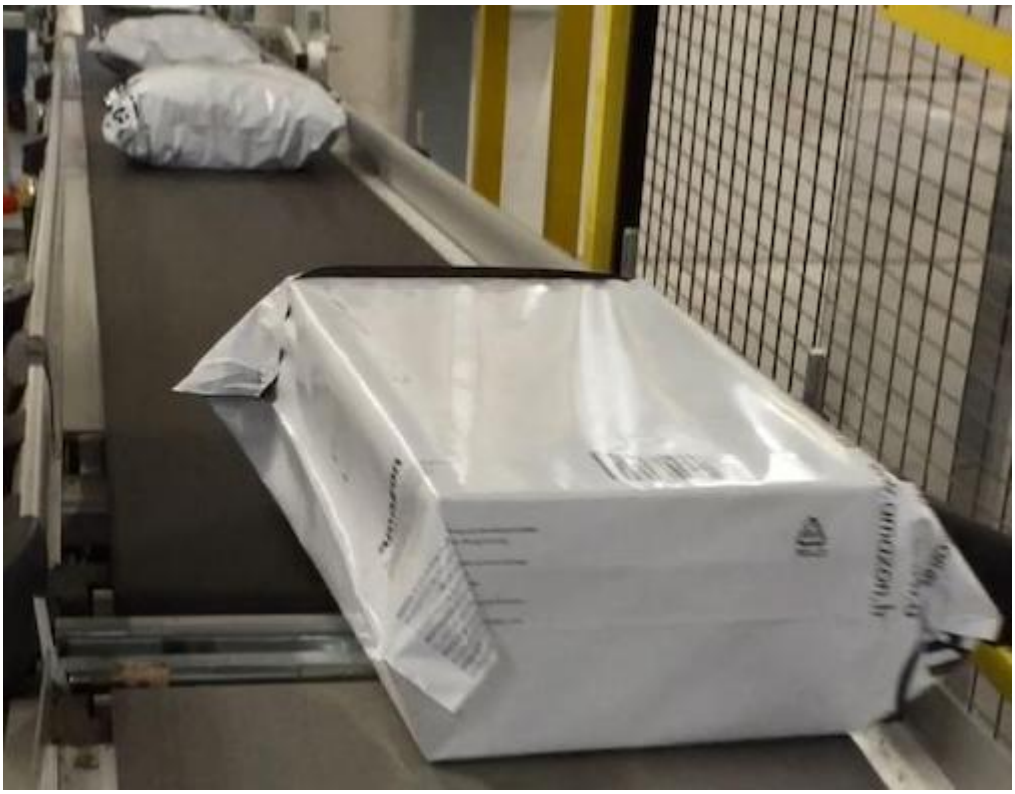
SmartPack



SmartPacks are bubble mailers that can be recycled via store drop-off. They are harder to reuse than boxes because the seal must be torn off to open the mailer. The bubbles offer some protection for fragile items,

but mailers often result in unnecessary packaging for smaller items. (We won't work with these on this project.)

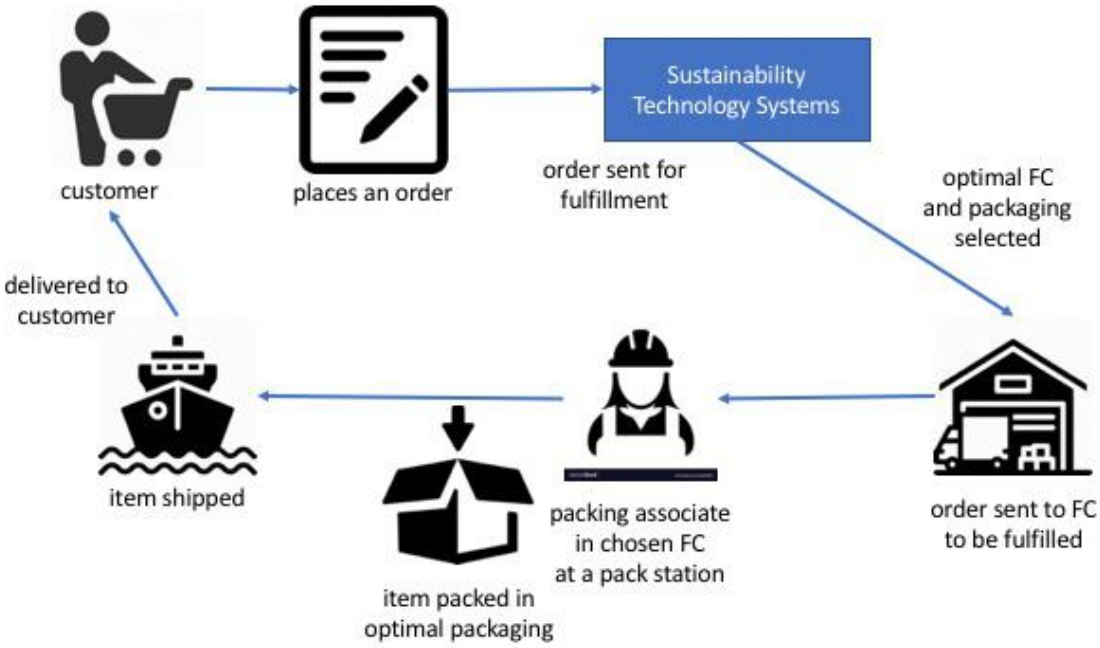
Polybag



Smaller and lighter than both boxes and SmartPacks, polybags are made of 80% recycled plastic and are 100% recyclable. They are also flexible, allowing a range of item shapes and sizes to be packed easily. Although they can be reused by replacing the tape holding them shut, they do not protect their contents very well. (This project will involve adding polybags as a packaging option to the existing shipment service.)

Your task, should you choose to accept it...

As it stands, the software architecture of our service allows for a single packaging **type** with several **dimensions**. As a member of the Sustainability team, you will extend the packaging recommendation system to make environmentally smarter packaging selections. The packaging software runs in the "Sustainability Technology Systems" part of the fulfillment process shown in this simplified diagram:



You will extend the service to support different packaging types after calculating and considering their environmental trade-offs. You will integrate these calculations into the selection process that decides how each item Amazon fulfills is packaged.

Technical Design

The SustainabilityShipmentService exposes one API, `PrepareShipment`. This API provides the most cost effective way to ship a single item from a specific fulfillment center (FC). The code to support this API is located in the `ShipmentService` class. It is responsible for providing the shipment recommendation.

The `ShipmentService` uses the `PackagingDAO` to identify all shipping options available in an FC that will fit the item. The `ShipmentService` then picks the shipment option with the lowest monetary cost and sends a shipping recommendation back to the associate in the FC who is packing the item.

You will update the service to support different packaging types, and to also consider sustainability, not just monetary cost, when choosing the best shipment option.

Project Preparedness Tasks

We're taking our first steps in native AWS with this project, so first an overview.

Thus far, we've written programs that work on our laptops. Java is designed to run on any computer that has a Java Virtual Machine (JVM), so we could copy our code to almost any computer and run it there (this is sometimes called "write once, run anywhere"). Even a computer in an entirely different building could run the program, as long as there was some way to give it input and receive its output. A computer that runs programs without a human directly interacting with it is often called a **server**.

Most of our programs aren't all that challenging for a server, though. If we dedicated an entire server to running our program, we'd be wasting a lot of its potential: after all, even our little laptops can run IntelliJ,

email, Chime, and a bunch of browser windows at the same time as our programs. An entire server dedicated to something like that would be a huge waste of computer time, electricity, and money.

Ideally, we'd want a server *just* powerful enough to run our program. Computers that small are not available, but we could use a larger computer to run a program that *emulates* many smaller computers at once, effectively converting a large computer into multiple small servers. We can then provide one such small server for each program someone wants to run.

AWS provides such servers. They're called Elastic Compute Capacity (EC2) instances. But we can be even more frugal.

Many programs spend most of their time waiting for input. The rest of the time they're just wasting electricity waiting. Ideally, we could reduce power and computer usage by waiting until the service is called by a client, at which point we could start an EC2 instance, install our program, run it, send back the result, and then turn off the EC2 instance so somebody else could use it. There's a lot of overhead in turning the computer on and installing our program, so we might want to leave it on for a minute or two just in case the same user had some more input.

AWS provides that capability, too. It's called Lambda.

Because Lambda runs programs without a dedicated server, it's considered "serverless". To be clear, there *is* a server involved; we're just not guaranteed anything about it except that it can run our program. It might run on one physical computer the first time, and a completely different computer the next time. A better name might have been "transient", but naming is hard, and "serverless" sounds more awesome. The team running the service doesn't need to know anything about the servers involved; Lambda takes care of it. And the team only pays for the time it actually spends running its service. The waiting time between requests incurs no cost (unlike paying for an EC2 instance 24 hours per day).

To run a serverless program on demand, we must package it in a way Lambda understands, along with instructions that tell Lambda what method to run and what resources the program accesses. AWS provides a way to define all this information called the Serverless Application Model (SAM), along with a `sam` tool that lets us run Lambda programs locally for testing.

To make use of these benefits, the `ShipmentService` runs on Lambda, so you're going to get first-first hand experience running a Lambda service! The RDE tools we used in Unit 1 allow us to combine all these serverless tools and processes into RDE workflows, so we can keep our development process simple. Let's get started!

Project Preparedness Task 1: Use the Source

Project Preparedness Task 2: Hello, Project Buddy

Project Preparedness Task 3: The Lay of the Land

Project Mastery Tasks

Mastery Task 1: Testing the Waters

Mastery Task 2: That Cloud Looks Like a Log

Mastery Task 3: Time is Marching On

Mastery Task 4: Time is Still Marching On

Mastery Task 5: The Cost of Progress

Mastery Task 6: I Fits, I Sits

Mastery Task 7: We Will Mock You

Project Reflection

Think over the entire project, from the introduction to the final mastery task completion.

Find the Canvas quiz titled: Project Reflection (in the Unit 2 Project Module), and answer three or more questions.