High-Level Program Design

Cloud-Native Accelerator

GENERAL ASSEMBLY

Scale Engineering Capabilities

Cloud-Native Accelerator

Instructor-Led | Onsite or Remote | ~35 Hours

Overview:
Take full advantage of cloud infrastructure by leveling up your Java engineers to build scalable web applications with the Spring framework.

Prerequisites:
Students must be comfortable building Java/Spring applications.

Business Outcomes:

- Upskill engineers to tackle technology projects at the enterprise level.
- Form the team you need for complex cloud migration projects.
- Save time and resources by moving away from traditional infrastructures.

~35 Hours of Cloud-Native Training

Cloud Infrastructure	Containerization with Docker	Microservices	Platform as a Service	Applied Practice
Dive into twelve-factor design and learn the benefits of adopting a cloud-based infrastructure.	**Use containers** and understand how they fit into modern development workflows.	**Explain the benefits of microservice architecture** and build scalable microservice applications.	**Leverage AWS** to enable rapid development and deployment.	**Apply topics** learned throughout the course into a final project.

Why Cloud-Native Accelerator?

- **Validated tools and approaches** for building scalable applications that are ready for the cloud, developed in partnership with top organizations.
- **Built with subject matter experts** with experience in cloud infrastructure, Java development, and systems architecture.
- 35 hours of expert-led, hands-on learning, including:
  - **Projects, labs, and assignments** that mimic real-world tasks and workflows.
  - **Case studies and examples** that demonstrate how businesses use the concepts.
  - **Regular feedback and touchpoints** with instructors and peers to ensure students meeting learning goals.

Learner Persona for Cloud-Native Accelerator

This product is specifically designed for the following audience:

- **Early- to mid-level Java engineers** with experience developing applications with the Spring framework (2+ years of experience preferred).

Anatomy of Cloud-Native Accelerator

Unit	What's Covered	Unit Project
Unit 1: **Cloud Infrastructure (8 hours)** Understand the benefits of adopting a cloud-based infrastructure and the different models of delivering cloud services.	• Cloud infrastructure concepts and benefits. • Distinguishing between cloud service models. • 12 factor application design. • Amazon Web Services tools. • Deploying an application with AWS.	Deploy a Java application on AWS EC2.
Unit 2: **Containerization (8 hours)** Use containers and understand how they fit into modern development workflows.	• How Docker works. • Dockerizing a Spring Boot application. • Running containers on Heroku. • How Kubernetes works. • Running Kubernetes with Minikube.	Deploy a Spring Boot application to Kubernetes.

Anatomy of Cloud-Native Accelerator

Unit	What's Covered	Unit Project
Unit 3: **Microservice Architecture (12 hours)** Explain the benefits of microservice architecture and build scalable microservice applications.	• Microservice vs. monolith architecture. • Domain-driven design for microservices. • API gateways with Eureka and Zuul. • Running microservices on Docker. • Splitting a monolith into microservices.	Build microservices for a music player application.
Unit 4: **Pivotal Cloud Foundry (12 hours)** Leverage PCF to enable rapid application development and deployment.	• How PCF works. • Deploying an application on PCF. • Software architecture principles and patterns. • Systems design thinking.	Whiteboard a systems architecture for a common application.

Tools Used in the Course

- JDK 8
- The course is written for MacOS; if students or instructors use a Windows or Linux system, some modifications to the curriculum may have to be made.
- Git and Github Enterprise
- IntelliJ IDEA Community Edition
- Zoom, Slack, and Google Chrome

Cloud-Native Accelerator

What You'll Learn

Module Overview

Cloud Infrastructure

8 hours

Why?

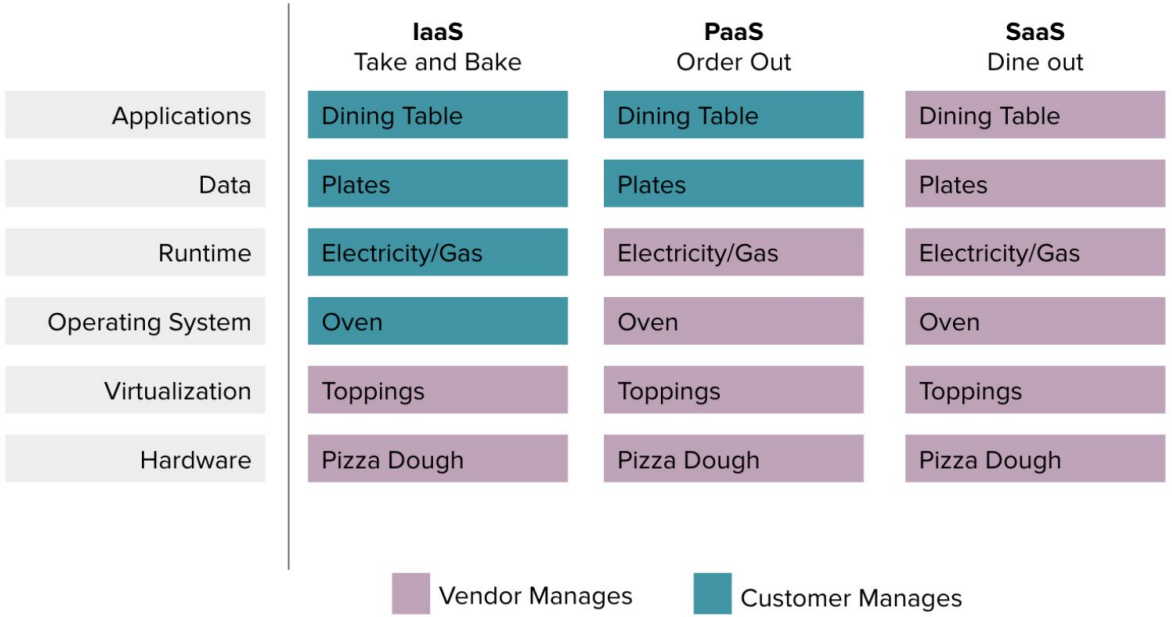
Almost all businesses today want to move all or part of their operations to the cloud, but it's not quite as simple as that. In this unit, students will explore the foundations of cloud infrastructure and the cloud delivery models available to businesses.

Learning Objectives

- Explain the benefits of cloud computing.
- Differentiate between the main cloud service models (SaaS, IaaS, PaaS).
- Recognize the difference between public and private PaaS.
- Define cloud native and the Twelve-Factor App Methodology.
- Explain how a hypervisor and virtualization work together.
- Launch an EC2 instance.
- Deploy code to a VM.

IaaS, PaaS, SaaS: What's the Difference?

Let's dive into each of the three service models: IaaS, PaaS, and SaaS.



Deploying an Application on AWS

The Deployment Process

Sign Up for an AWS Account

You should already have created an AWS account; if not, sign up here.

Create Amazon EC2 Key Pairs

You'll need these to SSH into your EC2 instance.

Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. Typical applications include remote command line, login, and remote command execution, but any network service can be secured with SSH.

To create the SSH, follow the instructions here.

Launch an Instance

Step-by-step instructions are provided here.

A couple of things to look out for:

- Make sure that "type" is Amazon Linux AMI.
- The configuration should be marked as "free tier eligible."
- The instance type should be "micro."

Module Overview

Containerization

8 hours

Why?

Containers are the first step in being able to build highly scalable and efficient microservices. Because containers can be deployed and run in almost any environment — including in the cloud — developers can spend more time thinking about logic and less time debugging.

Learning Objectives

- Differentiate between VMs and containers.
- Identify when (and when not) to use Docker.
- Pull an image from Docker Hub and run a container on your machine.
- Run source code as a container volume.
- Push a Docker image to production in Heroku.
- Explain Kubernetes and key terms.
- Run Kubernetes locally with Minikube.

How Docker Works

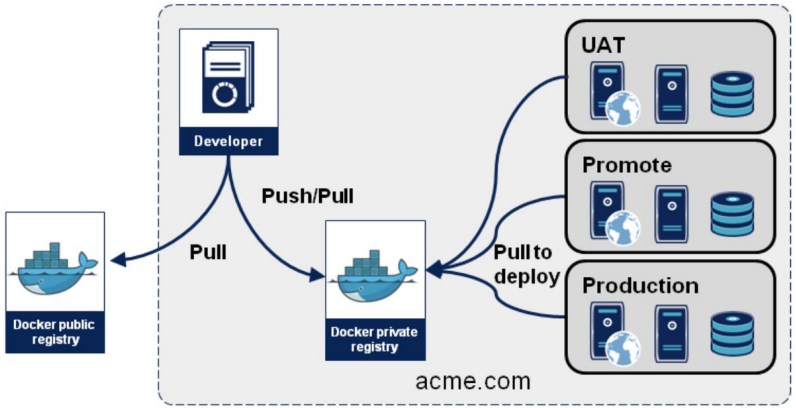
Docker is an open-source platform (the most popular one) for building applications using containers.

However, Docker isn't a completely new technology. Many of the components and principles existed previously. Docker is written in Go and takes advantage of several features of the Linux kernel to deliver its functionality, including namespaces and cgroups (more on those in the Additional Resources section).

The ultimate goal of Docker is to mirror our dev environment with our production environment. This is mostly useful for your back-end but can be applied to your front-end applications as well! It's cool to run Docker locally, but its real benefit comes into play while running it on some production machine.

Take a look at this *very* simplified version of how Docker works.

> **Knowledge Check:** Can someone explain what's going on here?



Running Kubernetes Locally With Minikube

In Docker world, you can quickly spin up a local Docker environment with Docker for Mac. Similarly, **Minikube** allows you to run a local Kubernetes environment.

When you spin up Minikube, it creates a local VM in the background. Inside this VM, it spins up a single-node Kubernetes cluster. It also set things up such that the Kubernetes API server and other services inside of the VM are also available outside in your local environment.

You can then use the `kubectl` binary on your local machine to manage the cluster inside the Minikube VM.

Let's Do It

The first step is to install Minikube. Click this link, select your operating system, and follow the installation instructions.

> **Note**: Minikube is NOT meant for production environments. It's ideal for local development environments on your desktop/laptop and lets you get familiar with Kubernetes in a short period of time.

Once you've completed the Minikube installation, follow this demo guide to learn how to:

1. Start Minikube.
2. Use `kubectl` to interact with the cluster.
3. Create a Kubernetes deployment.
4. Launch and access a pod.
5. Delete the service and deployment.
6. Stop and delete the cluster.
7. Manage a cluster.
8. Explore various configuration options.

See how far you can get in the next 45 minutes or so. Happy Kubernet-ing!

Module Overview

Microservices Architecture

12 hours 

Why?

After students have learned to manage and run containers, they'll apply these skills to building microservice applications. In a microservice architecture, individual services can be scaled up or down as needed, allowing businesses to fully capture the efficiencies of cloud computing.

Learning Objectives

- Discuss the pros and cons of microservices and monoliths.
- Describe a basic microservice architecture.
- Implement a API gateway and service registry using Eureka and Zuul.
- Explain how the API gateway, service registry, and services operate together in a microservice application.
- Identify seams and Bounded Context within a monolithic application.

Building a Microservice Application

Requirements

Use the `starter-code` directory to grab the starter code for the lab.

Set up a basic service that returns some songs at localhost:8080/songs.

Remember:

- The microservice will be a basic Spring Boot REST application, with some extra dependencies and annotations that tell it to register with our service registry (Eureka).
- You'll need to allocate a new port for each service to run on. The API gateway is already set to run on `8080`.
- We'll want the app to register at the `/songs` URL through the Zuul API gateway.

Adding `Song` Functionality

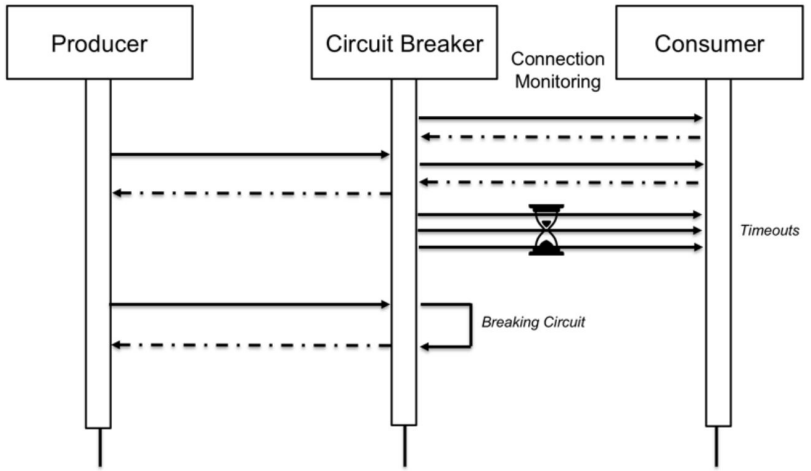
Once the service is wired up to our API gateway and service registry, we'll want to add some functionality to each. Both the `User` and `Song` microservices will share the same database.

2. Song service should have Flyway migrations that add `SONGS` table to the database.
    - The `SONGS` table should have at least `title` and `length` (in milliseconds) columns.
3. Song service should be able to perform CRUD (create, read, update, and delete data).

> The only catch to adding Flyway migration file is, as you know all the services by docker are run parallely so there is no way for Flyway to know which migration file to run first. That can really mess up it's head. There is no elegant way to combat this issue. So what you can do is put all migration files in any one service starting with different versions.

Avoiding Failure 101

Circuit Breaker



Patterns for Resilient

Architecture Circuit breaker with Timeout

What Is It?

- A timeout does not prevent our application from slowing down.
- With a circuit breaker, after a certain number of failed requests to the external service, the circuit is considered broken.
- After some time, a request is sent to an external service again; if successful, the circuit is reset.
- Care should be taken so that the circuit is not broken too readily nor takes an extended amount of time.

Module Overview

Pivotal Cloud Foundry

12 hours 

Why?

Being comfortable with multiple systems is an important skill for any developer. After starting with AWS, students will learn to prepare applications and deploy them on Pivotal Cloud Foundry (PCF).

Learning Objectives

- Explain how Pivotal Cloud Foundry differs from IaaS.
- Understand the key concepts of Pivotal Cloud Foundry.
- Deploy an application in Pivotal Cloud Foundry.
- Define systems architecture and common architectural patterns.
- Analyze the tradeoffs of flexibility, scalability, and security that come when designing a system architecture.

PCF Key Concepts

Cloud Foundry serves and scales applications using the following subsystems:

1. **Load balancing**: Loads are processed and distributed over multiple machines, optimizing for efficiency and resilience against a single point of failure.
2. **Apps run anywhere**: Component VMs constitute the platform's infrastructure, while host VMs host apps for the outside world. Cloud Foundry distributes app source code to VMs with everything the VMs need to compile and run the apps locally.
3. **Resource storage**: The Git system on GitHub is used to version-control source code, buildpacks, documentation, and other resources.
4. **Communication**: Messages are sent internally using HTTP and HTTPS protocols.
5. **Monitoring**: The component metrics and app logs are aggregated using the Loggregator system into a structured, usable form.

Discussion

These are just the high-level details, but there's a lot more to learn about the platform.

Take about five minutes to review the official documentation for Cloud Foundry. Then, with a partner, discuss what you've learned by answering the following questions:

- What makes PCF a powerful platform for developers?
- What sets it apart from other cloud providers?
- What questions do you still have?

We'll debrief your findings as a class.

Practicing Systems Design

In this exercise, you'll take about 30 minutes with a partner to sketch out on a whiteboard the system behind one of these services:

- Twitter
- Airbnb
- Uber
- Instagram

Here are a few tips to get you started:

- Divide the system into a few core components: front-end, back-end, database, and more.
- Identify some core functions on which you'll focus. Don't try to design for **every** functionality the company provides. (In a real interview, this is something you should always ask the interviewer.)
- Think through how the technology would interact well with the logic behind it. Take Twitter's "trending topics" feature, for example. Should you look at popular hashtags or search terms? What time frame should you look at? (These might be requirements you'll be given by an interviewer or decisions you'll have to make yourself.)
- Determine your data model and the APIs you'll need early on in your design.

Once you're done, each group will have a few minutes to present its design and the rationale behind the choices it made.


GA