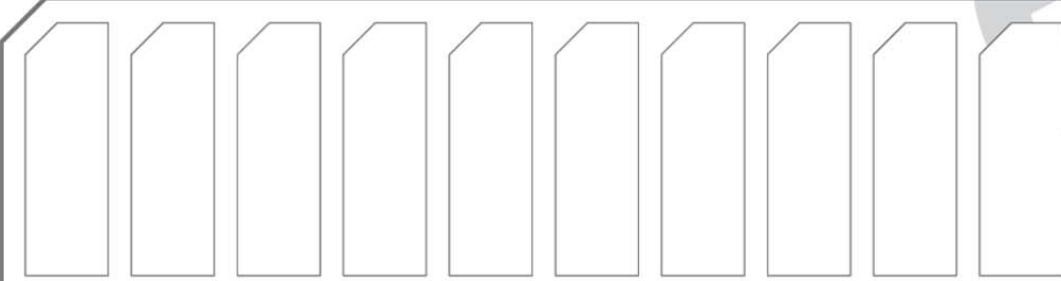


Cloud 300 Essentials Participant Guide



Contents

Facilities and Logistics	13
WebEx Logistics	14
Chat Window.....	17
Annotation Tools.....	18
Participant Window	19
Interactivity: Dream Vacation.....	20
Course Agenda	21
Cloud Essentials Introduction.....	23
Module Objectives	24
Thin Client/Mainframe.....	25
Client/Server	26
3-Tier Architecture	27
Interactivity: Matching Question.....	28
Cloud Native Application Architectures	29
Architecting and Re-architecting Applications for the Cloud	30
Automation Advantages.....	31
More Cloud Application Architecture Advantages	32
Cloud vs. Traditional Data Center Architecture	33
Cloud Comparison	35
Cloud Native Application Advantages	36
Module Summary.....	38
Twelve-Factor Methodology.....	39
Module Objectives	40
Challenges in Building and Deploying Software as a Service Applications (SaaS).....	41
Twelve-Factor App	42
Interactivity: Poll Question	43
Twelve-Factors	44
Codebase.....	45
1 – Codebase – Best Practices (git)	46
2 – Dependencies	48
3 – Config.....	50
4 – Backing Services	53
4 – Backing Service Usage.....	54
5 – Build, Release, Run	55
6 – Processes (Run time)	57
Interactivity: True or False?	59

7 – Port Binding	60
8 – Concurrency.....	61
9 – Disposability.....	63
10 – Dev/Prod Parity.....	64
11 – Logs	67
12 – Admin.....	69
Discussion: Twelve-Factor Methodology.....	70
Discussion: Twelve-Factor Deploy, Configure and Run.....	71
Module Summary.....	72
API.....	73
Module Objectives	74
What is an API?	75
Early History of APIs: Libraries	76
API Forms	77
Google Maps and Twitter.....	79
Protocols and Data Formats	80
Communication Requirements	80
Elements of Communication:	81
Interactivity: Matching Question	82
Communication Terms	83
Protocols	84
SOAP, REST, and GRPC	84
SOAP Web Services	85
SOAP and State and Disadvantages	86
REST	87
Interactivity: Poll Question	88
Serialization	89
Serialization Options.....	89
Platform-Specific Serialization.....	90
XML and Example	91
JSON and Example	92
BSON	93
Protocol Buffers (protobuf)	94
Thrift	95
Bond	96
Avro	97
Parquet.....	98

Comparison	99
Interactivity: Matching Question	100
Microservices	101
Monolithic Services vs. Microservices.....	101
UNIX Philosophy	102
Distributed Systems.....	103
Microservice Advantages and Architecture Advantages.....	104
The 5 Principles of Microservices Design and Implementation	105
JAX-WS and JAX-RS	106
Microservices Patterns	107
What is Spring Framework?.....	107
Interactivity: Poll Question.....	108
Inversion of Control Container.....	109
Advantages of IoC	110
Microservices and Containers	111
REST	112
What is REST?	112
REST Principles	113
A Case for REST	114
Jeff Bezos's Famous Memo @ Amazon	116
Interactivity: Poll Question.....	118
Evolution of Web Services	119
REST Success Story: AccuWeather on Azure.....	120
Success Stories of REST Web Services – Salesforce.....	121
Success Stories of REST Web Services – Amazon Web Services (AWS)	122
Success Stories of REST Web Services – Expedia.....	123
Components of REST	124
Interactivity: Poll Question.....	125
REST Basics	126
REST Webservice Examples	127
Interactivity: Poll Question.....	128
REST Design Principles	129
1: Client Server Architecture	130
2: Stateless	131
3: Cache-ability.....	132
4: Layered System.....	133
5: Code on Demand	135

6: Uniform Interface	136
What is HATEOAS?	137
Discussion: Serialization Protocols	138
Discussion: Authentication Standards.....	139
Lab: Implement a Protocol with Spring Boot	140
Lab: Open Source – REST Setup	141
Lab: Open Source – RESTful Time Tracker Web Service	142
Module Summary.....	143
Open Source	145
Module Objectives	146
Open Source and Optum.....	147
Software Licenses	148
Open Source Licenses Overview	148
Licenses, Free Licenses, and GNU GPL	149
Interactivity: Matching Question	150
GPL Versions	151
GNU LGPL	152
Permissive Licenses.....	153
Apache Licenses	154
Dual Licensing	155
License Summary.....	156
Interactivity: Poll Question	157
License Summary (Cont.).....	158
Cost Savings in Open Source.....	159
Discussion: Open Source Software Licenses.....	160
Discussion: Open Source Software License Considerations	161
Module Summary.....	162
Reliability, Resiliency, and High Availability.....	163
Module Objectives	164
Building Resilient Applications.....	165
Resiliency	165
High Availability	166
Measuring Uptime in Nines	167
High Availability	168
Interactivity: Matching Question	169
Redundancy	170
Examples of Redundancy	171

Discussion: Definitions and Examples	172
Discussion: Methods	173
Discussion: Analyzing an Application for Resiliency	174
Module Summary.....	175
Cloud Development Framework	177
Module Objectives	178
Microservices Introduction	179
Definition.....	179
Components of Microservices	180
Challenges of Cloud Enabling Microservices	181
Advantages.....	182
Interactivity: True or False?.....	183
Spring Cloud Introduction	184
Introduction.....	184
Scaling.....	185
Components Detail	186
Service Development with Spring Cloud	187
Ports to be Used.....	187
URLs to be Used	188
Spring Cloud Config Server.....	189
Interactivity: Poll Question	190
Bootstrapping Simple Web Service.....	191
1. Running Generated Web Service.....	192
2. Developing REST Limits Service and Deploying on 8080	193
3. Setup Git Local Repo with Config File	194
4. Generate and Setup Spring Cloud Config Server on Port 8888	195
5. Spring Cloud Config Setup	196
6. Changes to Limits-service to Read from Spring Cloud Config Server.....	197
Lab: Limits Service	198
Spring Boot and Spring Framework	199
Spring Framework	199
Spring Boot.....	202
Cloud SDKs	203
Interactivity: Poll Question	204
Cloud Application Scaffolding	205
Ways to Scaffold Cloud Application	205
Spring Initializr	206

Steps to Scaffold Boot Application with Spring Initializr	207
Boot CLI Setup	209
Steps to Scaffold Boot Application with Spring Boot CLI	210
Interactivity: Poll Question	211
Steps to Scaffold Boot Application with Spring Boot CLI (Cont.)	212
Cloud Scaffolding	213
Cloud Portability Patterns	214
Portability Definition and Theory	214
Where to Look out for Portability?	215
Advantages	217
Interactivity: Poll Question	218
How to Configure and Setup with Code?	219
How to Provision Instances?	220
Cloud Instance Setup with Scalr	221
When to Provision Instances?	222
Discussion: Microservices Advantages and Disadvantages	223
Discussion: Spring Framework	224
Lab: Spring Cloud Config Server	225
Module Summary	226
CI/CD	227
Module Objectives	228
The Build Life Cycle	229
Agile	229
Agile Events Flow	230
Enabling Methodologies	231
Manifesto	232
Agile Development Practices	233
CI and CD	234
CI/CD Flow within Optum	235
Interactivity: Poll Question	236
Introduction to Git	237
What is Version Control?	237
Version Control Systems	238
Git vs. Subversion	239
Repository	240
Is it a Repository?	241
Working Copy	242

Cloning	243
Cloning vs. Checkout	244
Remotes	245
Local History vs. Public History	246
Basic Git Operations – Diagram	247
Interactivity: Poll Question	248
Basic Git Operations	249
Jenkins and Maven Defined	250
Build Lifecycle	251
More About Jenkins	252
How does Jenkins Fit in with CI?	253
Jenkins History	254
Continuous Integration is Elusive	255
Discussion: CI/CD Concepts	256
Installing Jenkins (Demo)	257
Running Jenkins with All Demos	258
Maven with Jenkins	259
Use Cases	260
Architecture	261
Discussion: CI/CD Scenarios	262
Module Summary	263
Container Fundamentals	265
Module Objectives	266
Container Introduction	267
The Problem	267
Deploying to the Server	268
Much Later... Finally Working	269
Redeploying to a Different Server	270
Redeploy to Cloud	271
But It Worked on MY Computer	272
Remember Windows DLL Hell?	273
Dependency Hell	274
Solutions to Dependency Hell	275
The Matrix from Hell	276
Interactivity: Matching Question	277
Wouldn't It Be Nice?	278
Virtual Machines (VMs)	279

There is a Better Way: Virtualization	279
The History of Virtualization	280
Virtualization Providers.....	282
The Cost of Virtualization	283
The Problem with Virtualization.....	284
Interactivity: True or False?.....	285
Introducing Containers.....	286
What are Containers	286
Containers Versus Virtual Machines	287
Containers Versus Package Managers.....	288
Solving the Matrix from Hell	289
When NOT to use Containers	290
Container History	292
History of Containers	292
Linux Containers.....	293
Docker	294
Interactivity: Poll Question	295
Docker Overview	296
What is Docker?.....	296
Docker uses the Host Kernel.....	297
Docker for Windows (Developer)	298
Docker and Windows Server.....	299
Docker on Windows Use Cases.....	300
Docker on Mac (Developer)	301
Interactivity: Poll Question	302
Docker Hub and Container Registries	303
What is Docker Hub?	303
Container Registries	304
Docker Trusted Registry.....	305
Docker Pull	306
Docker Commands	307
Docker LS (list).....	307
Docker Run.....	308
Running a Shell	309
Lab: Pulling a Container	310
Docker RM.....	311
Removing a Container.....	312

Lab: Manipulating a Container	313
Docker Operations	314
Port Mapping	314
Running a Web Server	315
Mapping nginx	316
Running in Background	317
Lab: Ports	318
Container Logging	319
Application Logs	320
Is Docker Secure?	321
Kernel Exploits	322
Resource Starvation	323
Docker CLI Tools	324
CLI	324
Interactivity: Poll Question	325
Permissions	326
What is a Dockerfile?	327
FROM Statement	328
Changes to Image	329
Networking	330
Building	331
Lab: Dockerfile	332
Health Checks	333
Container Health	333
Discussion: DTR	334
Health Check Example	335
Anatomy of a Dockerfile – FROM and ENV	336
Anatomy of a Dockerfile – RUN	337
Dockerfile Best Practices	338
Discussion: Docker	339
Module Summary	340
Further Readings	341

This page intentionally left blank.

Facilities and Logistics

Please Turn Off Any Other Electronic Devices

Close any open applications other than WebEx and turn off other electronic devices

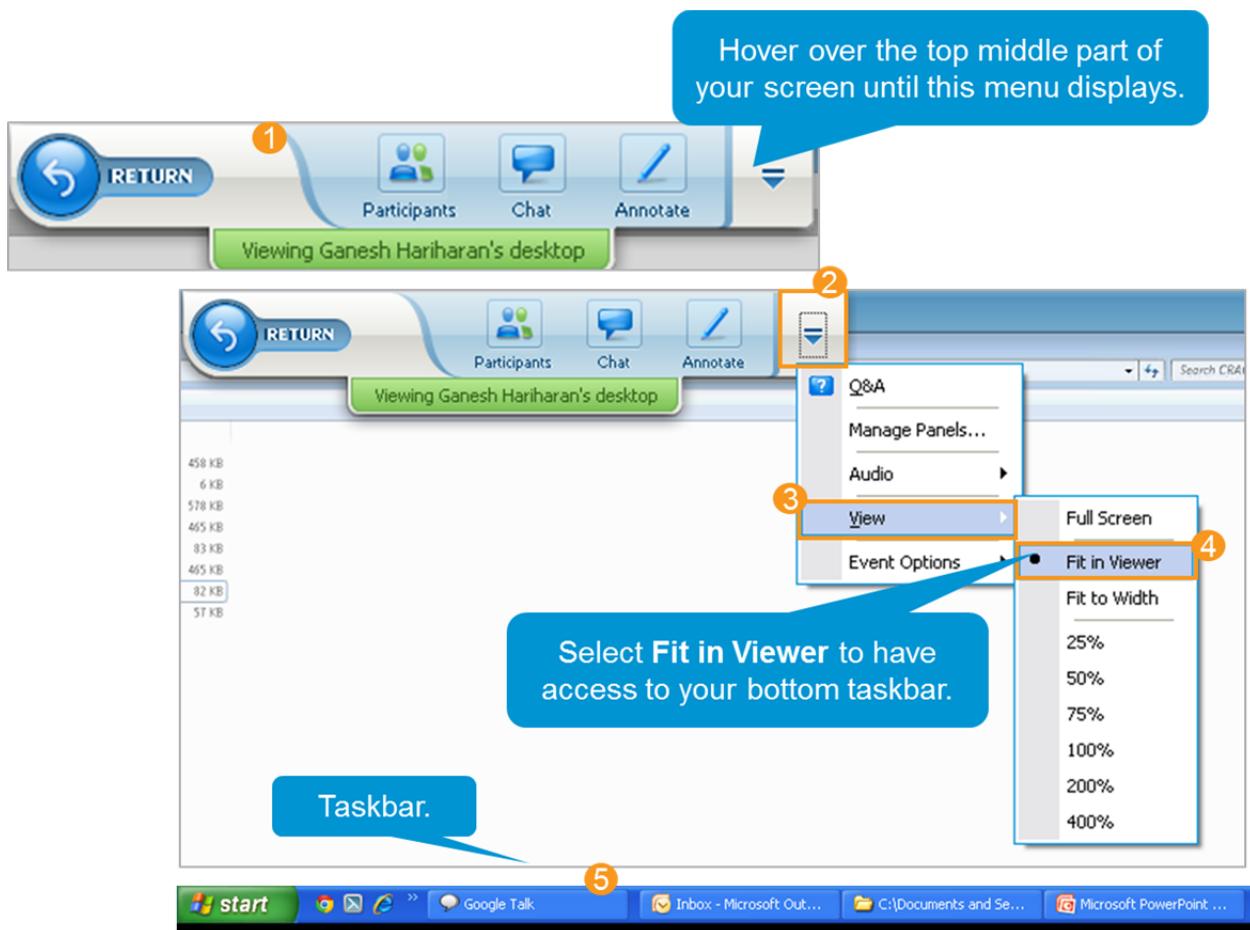
WebEx functionality will be reviewed prior to the class starting

Ensure that the instructor has you signed in for the class

Breaks and Lunch

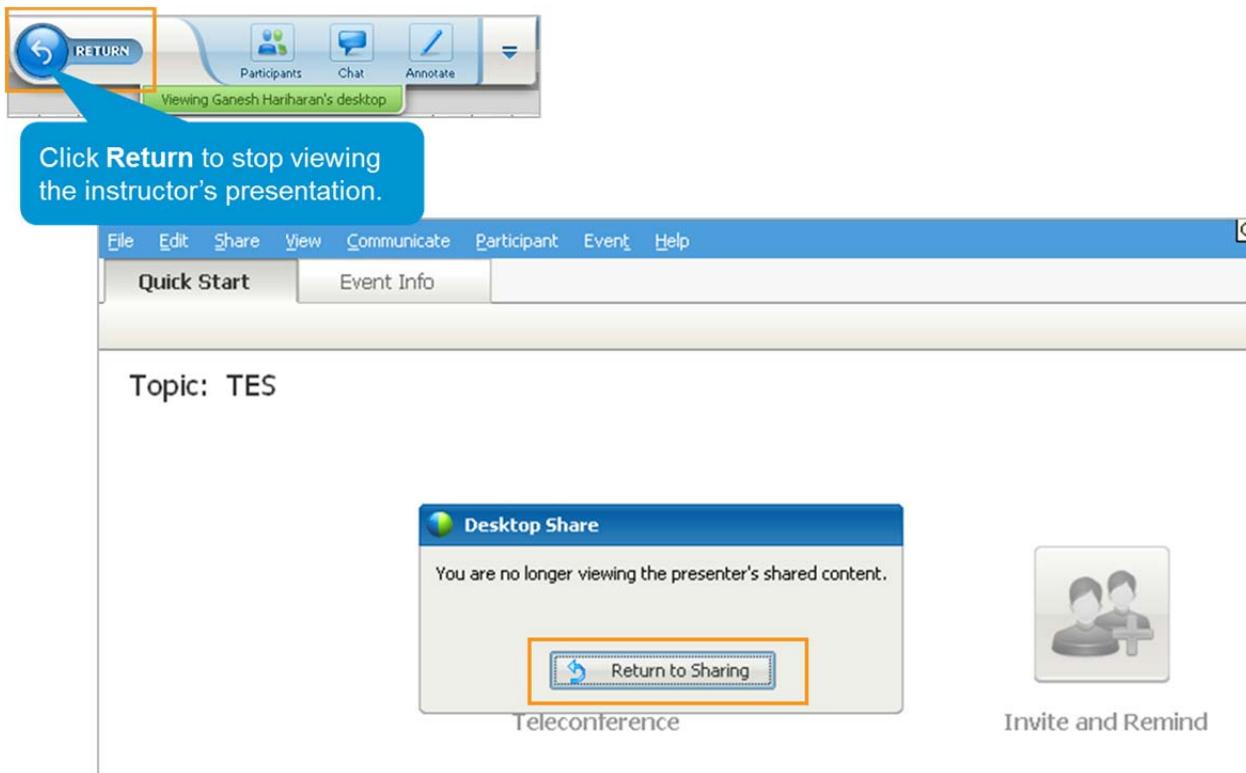
Q & A

WebEx Logistics



- When you attend a WebEx session and the presenter shares their screen, the session may take over your full computer screen. If this occurs, you may not be able to access the task bar at the bottom of your computer screen that shows all of your open applications and your **Start** menu.
- If this happens, you can adjust the screen fit to allow you access to the task bar while viewing the presentation:
 - To begin, hover over the top middle part of the screen until the drop-down menu shown appears.
 - Click the drop-down arrow that appears to the right of the pull-down menu.
 - Hover over the **View** option.
 - Select **Fit in Viewer**.
 - Once you make this selection, the task-bar will reappear at the bottom of your computer screen.

WebEx Logistics (Cont.)



- If you would like to stop viewing the presentation at any time, click **Return**.
- To return to the presentation, click **Return to Sharing**.

WebEx Logistics (Cont.)

Navigate between your Participant Guide and the WebEx session via the taskbar.



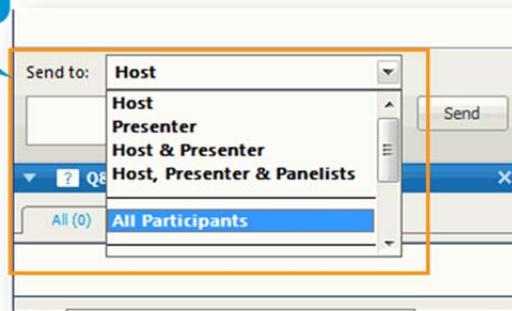
- During this class, if you are using an electronic (PDF) version of the Participant Guide, you may need to switch between viewing the WebEx presentation and your Participant Guide.
- Once you have the Participant Guide open, you can navigate between the two via the links that appear in your taskbar.

Chat Window



Select **Chat** to launch a separate pop-up Chat window.

Choose the people to whom you will send your chat.



- During this session, you will need to use the Chat panel to answer knowledge check questions, and to ask the instructor any questions you may have.
- To access the Chat panel while the session is launched, select **Chat** from the menu we introduced on the previous page—remember that this menu appears when you hover over the top middle portion of your screen.
- After you click **Chat**, a new Chat window will open. You can move this window anywhere on your screen so you can easily access it while you are watching the presentation.
- You can choose the person or group of people to whom you will send your chat by making a selection from the **Send to:** drop-down menu. All interactivities require responses to be sent to All Participants.

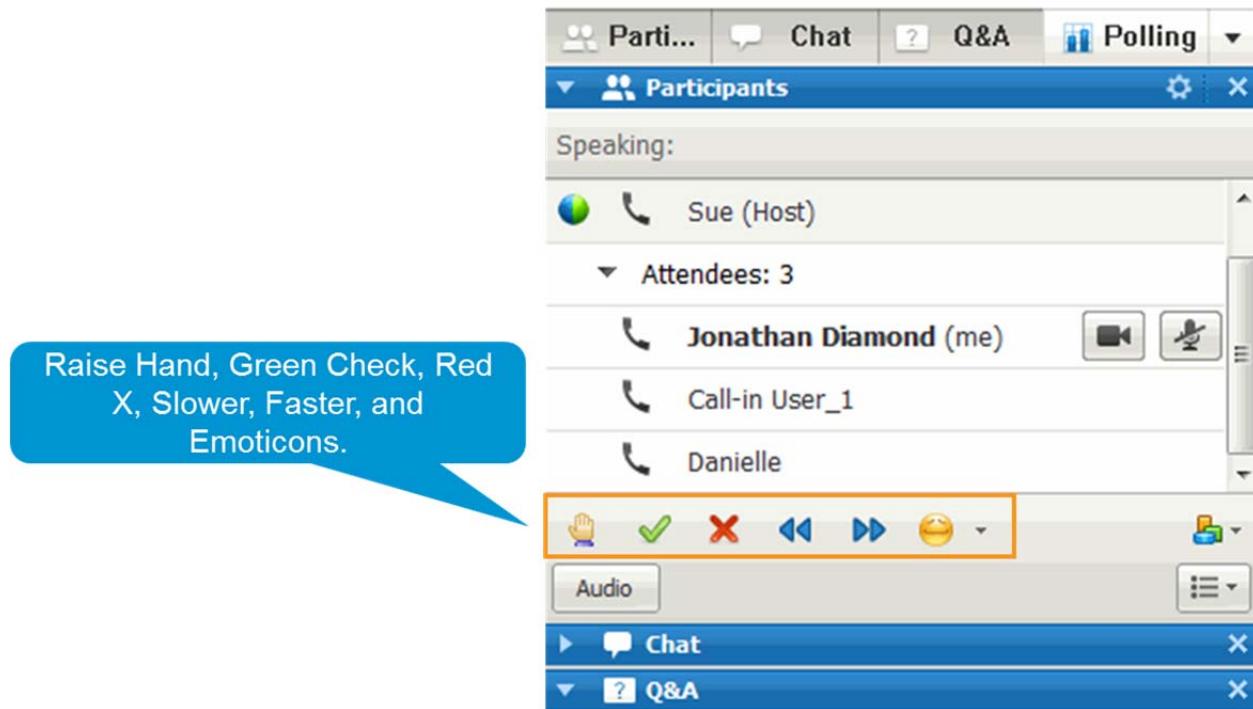
Annotation Tools



The **Annotation Tools** are used to interact with slide content. Each virtual interactivity will provide specific instructions for how you should participate using the tools available. The **Annotation Tools** include the following:

- **Pointer**: Lets you point out text and graphics on shared content. The pointer displays an arrow with your name and annotation color. Clicking this button again and then clicking the Close button turns off the text tool.
- **Text** Lets you type text on shared content. Attendees can view the text you have entered after you type it and click your mouse in the content viewer, outside the text box. To change the font, on the **Edit** menu, choose **Font**. Clicking this button again and then clicking the Close button turns off the text tool.
- **Line** Lets you draw lines and arrows on shared content. For more options, click the downward-pointing arrow. Clicking this button again and then clicking the Close button turns off the Line tool.
- **Rectangle** Lets you draw shapes, such as rectangles and ellipses on shared content. For more options, click the downward-pointing arrow. Clicking this button again and then clicking the Close button turns off the Rectangle tool.
- **Highlighter** Lets you highlight text and other elements in shared content. For more options, click the downward-pointing arrow. Clicking this button again and then clicking the Close button turns off the Highlighter tool.
- **Annotation Color** Displays the Annotation Color palette. Select a color to annotate shared content. The Annotation Color palette closes.
- **Eraser** Erases text and annotations or clears pointers on shared content. To erase a single annotation, click it in the viewer. For more options, click the downward-pointing arrow. Clicking this button again and clicking the Close button turns off the Eraser tool.

Participant Window



The **Participants Window** has various tools that can be used including:

- **Raise Hand:** Lets you get the instructors attention to ask a question or volunteer to respond.
- **Green Check/Red X:** Used to respond to Yes/No or True False questions from instructor.
- **Slower/Faster:** Used to ask the instructor to speed up or slow down their pace of instruction.
- **Emoticons:** Various emoticons can be used for communication purposes. The Break icon is commonly used in virtual training sessions.

Interactivity: Dream Vacation

Question: Where is your dream vacation location?

Instructions: Use Pointer tool.



Course Agenda

Topics:

- Module 1: Cloud Essentials Introduction
 - Module 2: Twelve-Factor Methodology
 - Module 3: API
 - Module 4: Open Source Software
 - Module 5: Reliability/Scalability/Resiliency
 - Module 6: Cloud Development Frameworks
 - Module 7: The CI/CD Process
 - Module 8: Container Fundamentals
-

There are 8 main sections to this course.

This page intentionally left blank.

Cloud Essentials Introduction

Module 1

Module Objectives

After completing this module, participants will be able to:

- Describe thin client/mainframe architecture.
 - Describe client/server architecture.
 - Describe 3-Tier Architecture.
 - Define Service-Oriented Architecture.
 - Describe the benefits of Cloud Native Applications.
-

Before we begin our look into the essentials of cloud computing, let's review a bit that we learned in the 200 level course. Specifically, we'll quickly examine a few different types of architecture that clearly influenced cloud computing.

After completing this module, participants will be able to:

- Describe thin client/mainframe architecture.
- Describe client/server architecture.
- Describe 3-Tier Architecture
- Define Service-Oriented Architecture
- Describe the benefits of Cloud Native Applications

Thin Client/Mainframe

Benefits of thin client/mainframe model:

- Hardware was optimized.
- Security was increased because there was less to monitor.
- TCO was lowered.
- The server could handle more sessions than a client could.

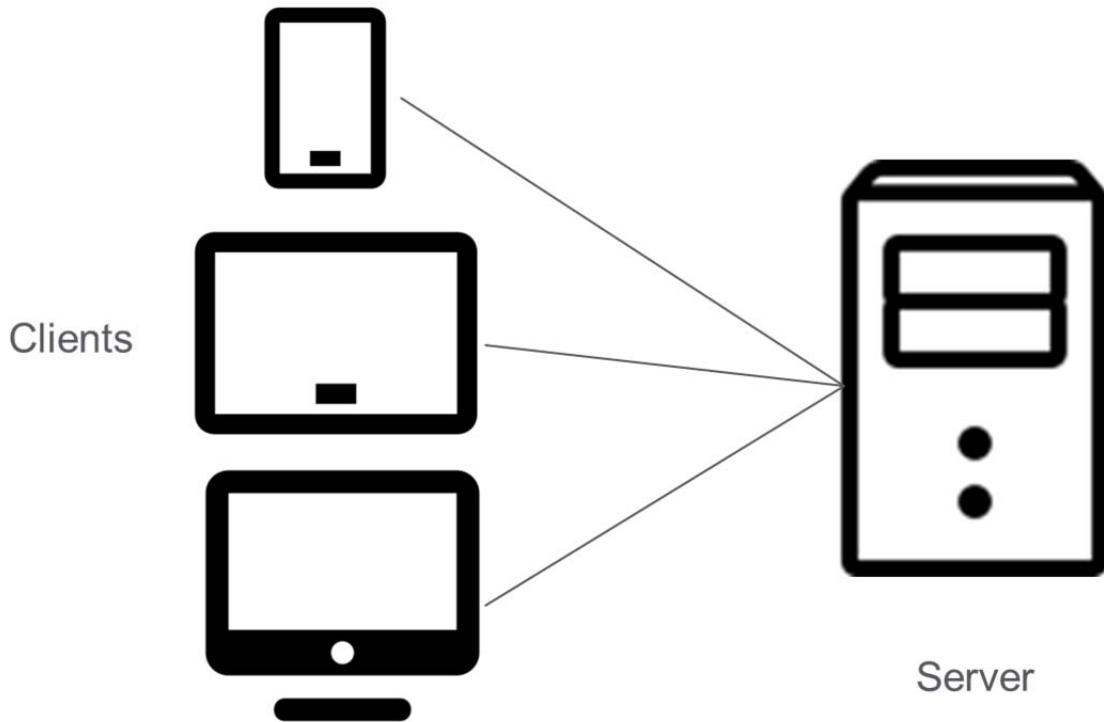


It used to be quite common for a central server to be used to do most of the computing work: launch and run applications; store data, and more. A number of “thin clients” would connect to this central server (a mainframe) as the thin clients had relatively little processing power of their own. This centralization helped in a number of ways:

- Hardware was optimized.
- Security was increased because there was less to monitor.
- TCO was lowered.
- The server could handle more sessions than a client could.

It's not hard to see how this was a precursor to the Cloud computing that we've seen develop in recent years.

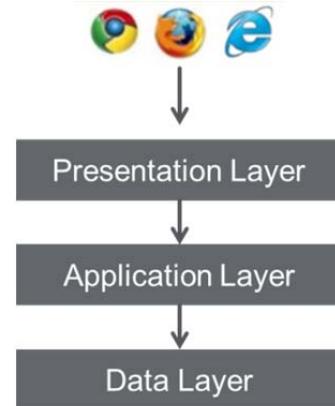
Client/Server



Next came the client/server model. A client could really be anything that was requesting a service, and a server could be anything that provided a service (this could even be another client that was running a server). But unlike the thin client/mainframe model, the client/server model meant that high-functionality devices could be clients; thin clients were little more than monitors in many cases.

3-Tier Architecture

- Presentation Layer: Interface that translates tasks and results
- Application Layer: Processes business logic
- Data Layer: Database & storage that provides access to applications



The architecture that makes up today's current cloud computing services environment can be summarized as a front end platform, a back-end system, and storage.

It's more commonly known as Presentation, Application, and Data layers, which we've seen called "3-tier Architecture," which is a type of client-server architecture.

Presentation Layer: The Presentation Layer is at the top level and displays information related to available services. The Presentation layer communicates with other layers by sending results to a web browser and other layers in the network. Presentation layers are often built on web technologies like HTML5, JavaScript, CSS, or other web development frameworks. It communicates with other layers through API calls.

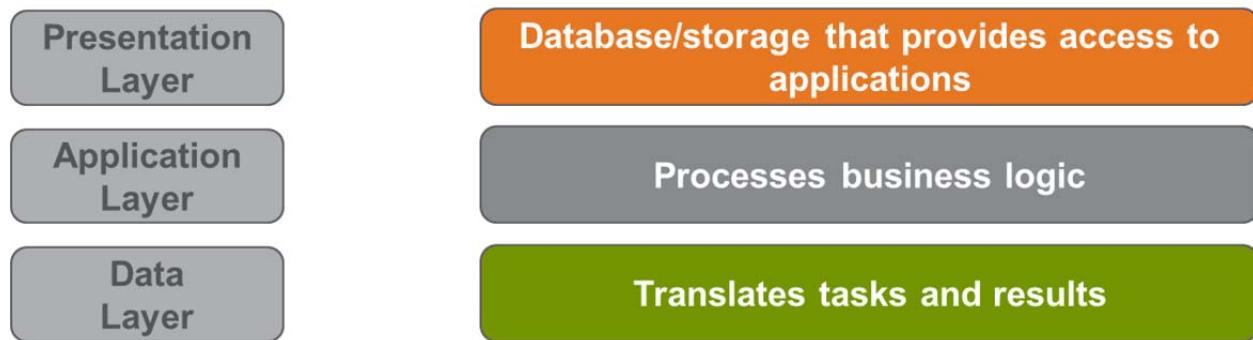
Application Layer: The application layer can also be called the middle layer, business logic, or logic layer. It controls application functionality by performing detailed processing. This might be written in C#, Java, C++, Python, and Ruby.

Data Layer: The Data Layer contains database servers and data storage systems where information is stored and accessed. The Data Layer provides access to application data. Data in this layer is kept independent of application servers or business logic. Examples are MSSQL, MySQL, Oracle, or PostgreSQL, MongoDB. Data is accessed by the application layer via API calls.

Interactivity: Matching Question

Question: Match each level of the three-tier architecture with its description.

Instructions: Raise Hand to volunteer and then use the Line tool.



Cloud Native Application Architectures

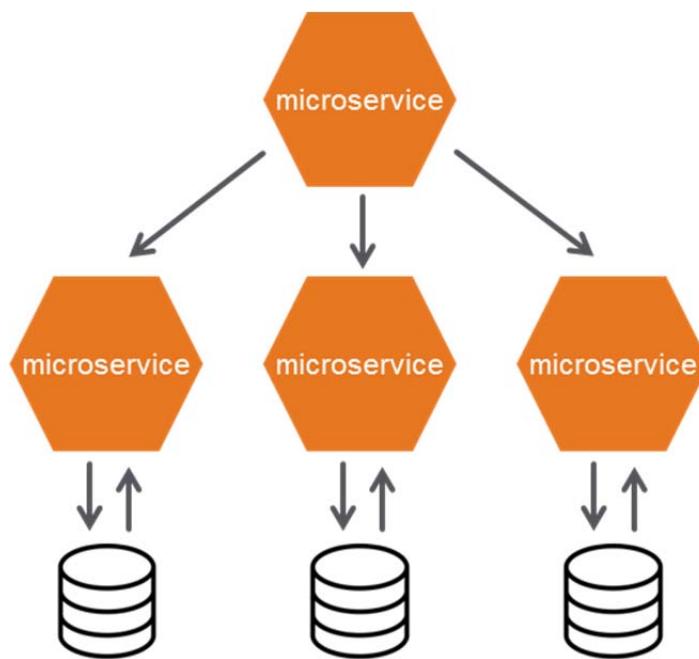
- Applications designed specifically for virtualization environments and cloud computing architectures that are “born in the cloud”
- **Designed to be used with cloud-computing frameworks:**

Elastic and scalable

Use of APIs

Open-sourced

Loosely coupled



Cloud native refers to applications that are born in the cloud, and this is in contrast to applications that are born and raised on company premises. Cloud native apps are intended and designed to use cloud computing frameworks, which are easier to build, deploy and update. Cloud Native architecture is microservice-based, and are automated when possible: CI/CD, APIs, automated configuration management — everything that can be automated is automated. DevOps drives Cloud Native Application, and this helps to ensure that the builders of an application also run that application.

Architecting and Re-architecting Applications for the Cloud



Cloud-native Applications:

- Performance
- Resilience
- Manageability



Microservices:

- Loosely coupled collection of services
- Built as a distributed collection of services – pairs well with cloud
- Team structure is realigned with the architecture



Monolithic:

- Codebase encompasses all features
- One big giant “lump” of code
- Is by definition a “single point of failure”

We're discovering that applications designed specifically for the cloud they run on will best offer the benefits of performance, resilience, and manageability.

Why is that?

A microservice-based application architecture provides a loosely coupled system and is easier to build, deploy, and update. Applications that are hosted on microservices are built as a collection of services and pair very well with the distributed nature of the cloud.

Applications that were hosted on hardware servers were at one time built as monoliths, where the codebase included every feature and service that made up the app as a single, giant lump of code. Today, with a microservices architecture, apps are being built as a distributed collection of smaller services.

With microservices, team structures can be realigned to reflect the change in architecture. Today software teams are small and cross-functional. This keeps teams agile, improves collaboration and quickens decision-making. It also eliminates a host of accountability issues, as teams own the product together.

Automation Advantages

Automation

- Auto-provisioning environments as the code is portable.
- Auto-scaling to add resources when needed.
- Auto-redundant applications are naturally resilient to failure.



Automation is another key benefit to applications architected for the cloud, especially as Automation through IAC is testable.

- Auto-provisioning environments as the code is portable.
- Applications can auto-scale (depending on traffic spikes, for example) and add resources only when needed.
- Auto-redundant applications are naturally resilient to failure. Processing moves to another server or zone automatically and seamlessly; the users never know.

More Cloud Application Architecture Advantages

Break Down the Silos

- Project management, Operations, Development, QA, Test...
- Each have their own goals, priorities, and management.
- With microservices, the team structure is realigned with the architecture.
- Small cross-functional teams = agile.
- Focus on including application developers in the architecture instead of just placing a “business requirement” on them.

Faster Software Releases

- Right infrastructure + right tools + right team = efficient & frequent releases.
- Errors and bugs are easy to find, and fix.
- Rollback to previous releases are easy with the right tools:
 - Kubernetes
 - Fluentd
 - Prometheus

More advantages to architecting applications for the cloud include breaking down the silos, which shifts things from a blame culture to a blameless culture.

Often, project management and other departments each have their own goals and priorities. These agendas often clash, and the applications suffer as a result. With Cloud Application Architecture, it naturally lends itself more to including application developers in the architecture instead of just placing a “business requirement” on them.

An additional advantage would be faster software releases. This means implementing the right infrastructure, tools, priorities, and management, and that leads to the benefit of efficient and frequent software releases. Bugs are much easier to find with this method, and its easier to rollback using some of the newer tools.

Let's take a quick look at some of these tools. Kubernetes is the leading container orchestration platform that lets you deploy and manage containers at scale. Fluentd unifies logging by collecting and sharing log data almost everywhere you need it via 500+ plugins. Prometheus: is monitoring tool that records time-series data for distributed service-oriented applications. Its strength is that it can function even if the rest of the system has failed.

Cloud vs. Traditional Data Center Architecture

Scalability

- Data centers have limited capacity, provisioned resources
- Cloud has unlimited storage due to scaling

Automation

- Data centers have heavy administration (upgrades, configuration, security)
- Cloud is managed by the provider

Cost

- Data center requires purchasing equipment upfront for business planning
- Cloud is more cost effective – pay for what you use
- Security
- Data center infrastructure requires you to protect your data
- Cloud provides security mechanisms
- DevOps teams are responsible for security



- Cloud computing is more than just hosting applications in the cloud. It's an abstraction of services from physical hardware.
- Here are some differences between cloud hosting and traditional web hosting:
 - With traditional data center infrastructure, you have limited capacity, you only have the resources that are available to you. If you run out of storage space, you purchase or rent more servers.
 - Cloud offers more flexibility and scalability, since cloud computing offers more server resources and unlimited storage. Depending on the amount of traffic that is received, cloud servers can scale up or down based on demand.
 - Traditional data centers have heavy administration and are both costly and time-consuming.
 - Server maintenance, such as upgrades, configuration problems, threat protection and installations are required.
 - Cloud hosting is managed by the provider who takes care of all the necessary administration.
 - With a data centers infrastructure, you will need to purchase equipment and additional server space upfront to adapt to business growth. If business slows, you pay for resources you don't use.

- Cloud computing is more cost-effective than the data center. With cloud-based services, you only pay for what is used.
- With a data center infrastructure, you are responsible for the protection of your data.
- Cloud computing looks less secure because it is externally stored, hosted, and delivered, but public cloud physical facilities are usually locked-down with highly restricted on-site access. Therefore, choosing a cloud service provider also involves security considerations, not just storage and delivery considerations.
- Multiple teams within Optum secure networking and storage on your behalf within the data center. DevOps Teams are responsible for ensuring the security of their environment top to bottom.

Cloud Comparison

Many CSP choices, all offer multiple services and capabilities:

- High-availability
- Durability
- Security
- Performance
- Support
- Automation

The market is dominated with:

- MS Azure
- Amazon Web Services (AWS)
- Google Cloud Platform

Each CSP provides an array of products and all referred to by different names.



- There are many choices for public CSP's (cloud service providers) and all offer multiple capabilities and services.
- High-availability (will my data be available when I need it?), durability (will it be there in the future?), security, performance, customer support, and automation are all offered, and at different levels.
- The market for public CSP's is dominated by a top three—Microsoft Azure, Amazon Web Services, and Google Cloud.
- Each CSP provides an array of products and each differs not only in pricing but also how each service is referred to or grouped.

Cloud Native Application Advantages

- **Decentralized:** On-premises, on the other hand, is centralized.
- **Microservices:** Apps hosted on hardware are seen as “monolithic.” Apps hosted on microservices are built as a collection of services, which naturally pairs with the distributed nature of the cloud.
- **Automation:** Auto-scaling and tracking applications and pulling in resources when needed. Auto-redundant applications are naturally resistant to failure. Processing moves to another server or zone automatically and seamlessly.
- **Silos broken down:** Dev, QA, IT, Admins, sysadmin, release, and project management all have their own goals, priorities, and management. With microservices, the team structure is realigned with the architecture for overall improvement.
- **Faster software releases:** Errors and bugs are easy to find and fix. Rollback to previous releases is easy with the right tools
Cloud-native tools include Kubernetes, Fluentd, and Prometheus.

- Let's cover just a few of the features and benefits of Cloud Native applications.
- To start with, on-premise infrastructure is a centralized system. In the cloud, however, servers and databases are dispersed. We've already seen the many benefits of this so far.
- Apps that were hosted on hardware servers were built as monoliths. The codebase included every feature and service that made up the app as a single, giant lump of code. Now, microservices architecture allows apps to be built as a distributed collection of services, which aligns with the distributed nature of the cloud.
- Automation is also at the forefront now, and is a major advantage that cloud native architecture provides. Cloud native architecture helps with auto-scaling and tracking applications, only pulling in resources when they are needed. Additionally, auto-redundant applications are naturally more resilient. Processing moves to another server or zone automatically and seamlessly; the user is never aware of the change.
- Cloud computing also breaks down traditional silos. Dev, QA, IT, database admins, sysadmins, release management, and project management all have their own goals and priorities when engaged in the development effort. They have competing agendas that frequently clash, and this can strain an application. Thanks to microservices, team structures can be realigned to reflect the changes in architecture. Modern software delivery teams can be much more nimble, much smaller, and they can be much more cross-functional. Each team is organized to align with the services that they support. This way, teams remain agile, with faster decision-making and collaboration. This team organization also prevents many accountability issues. Teams can give up responsibility for a feature once it moves on from their team and before it is released.
- Cloud-native releases become much more frequent, and there are no longer any major releases. Every release affects a single service, or a handful of services at most. Because of this limited

scope for every release, errors become much easier to spot and fix. Rollback is also made easier, with many tools capable of automatic rollbacks to the previous stable version when a new version fails.

Module Summary

Now that you have completed this module, you should be able to:

- Describe thin client/mainframe architecture.
 - Describe client/server architecture.
 - Describe 3-Tier Architecture.
 - Define Service-Oriented Architecture.
 - Describe the benefits of Cloud Native Applications.
-

Now that you have completed this module, you should be able to:

- Describe thin client/mainframe architecture.
- Describe client/server architecture.
- Describe 3-Tier Architecture.
- Define Service-Oriented Architecture.
- Describe the benefits of Cloud Native Applications.

Twelve-Factor Methodology

Module 2

Module Objectives

After this module, participants will be able to:

- Describe best practices outlined by Twelve-Factor App methodology for building highly available, highly scalable apps.
 - Recognize problems that the Twelve-Factor App methodology is designed to solve.
-

One thing we'll return to throughout this entire course is how Cloud development at Optum ties back to a development methodology called the Twelve-Factor App. This methodology can help guide all stages of development and are intended to be best practices that enable applications to be built with portability and resilience.

Challenges in Building and Deploying Software as a Service Applications (SaaS)

Nowadays users expect resilient / highly available apps.

- Apps like Facebook, Twitter, etc. have set the expectations high.

Developing highly available / resilience applications is not trivial.

The need to deploy applications rapidly and reliably.

The need to control application environments reliably.

Applications usually run on multiple servers for high availability.
And many more.

Twelve-Factor App development methodology can help.



Companies take their services being highly available very seriously. As downtime means, disruption of services, lost revenue, and users are potentially going to competitors.

In 2013 Amazon suffered a 13 minutes outage. Forbes magazine calculated the outage cost the retailer around \$66,000 a minute – totaling \$2.6 million lost during the outage.

Link to the case study: <https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer>

Twelve-Factor App

'Twelve-Factor App' is a collection of best practices and methodologies for building Software as a Service (SaaS) applications. These practices were gathered at [Heroku](#) (cloud platform as a service). Now they are generalized.

These principles are mainly focused on web apps. So adopt these for your environment. The practices are being adopted in Microservices deployment.



Heroku (<https://www.heroku.com/>) is a 'Platform as a Service (PaaS)'. It was created in 2007 and probably the 'first' cloud platform. Initially, it supported Ruby on Rails applications (now it supports many more languages). Heroku became very popular because it made deploying scalable applications very easy.

It was later acquired by Salesforce in 2010.

Interactivity: Poll Question

Question: What are the twelve-factor apps about?

- A. Architectural patterns for cloud systems
 - B. A system of microservices
 - C. Code that developers use to develop apps
 - D. A computer program that performs virtualization
-



Twelve-Factors

Factor	Description
1 - Codebase	There should be exactly one codebase for a deployed service with the codebase being used for many deployments.
2 - Dependencies	All dependencies should be declared, with no implicit reliance on system tools or libraries.
3 - Config	Configuration that varies between deployments should be stored in the environment.
4 - Backing services	All backing services are treated as attached resources and attached and detached by the execution environment.
5 - Build, release, run	The delivery pipeline should strictly consist of build, release, run.
6 - Processes	Applications should be deployed as one or more stateless processes with persisted data stored on a backing service.
7 - Port binding	Self-contained services should make themselves available to other services by specified ports.
8 - Concurrency	Concurrency is advocated by scaling individual processes.
9 - Disposability	Fast startup and shutdown are advocated for a more robust and resilient system.
10 - Dev/Prod parity	All environments should be as similar as possible.
11 - Logs	Applications should produce logs as event streams and leave the execution environment to aggregate.
12 - Admin Processes	Any needed admin tasks should be kept in source control and packaged with the application.

The twelve-factor is a collection of best practices and design patterns that address the needs of the modern software development, often written for delivery through the browser and often supplied to users as a SAAS (Software as a Service).

The principles described here cover all phases of software production, from creating and maintaining the code to testing and deployment to configuration management.

Codebase

Code and resources (icons etc.) are tracked in version control system.

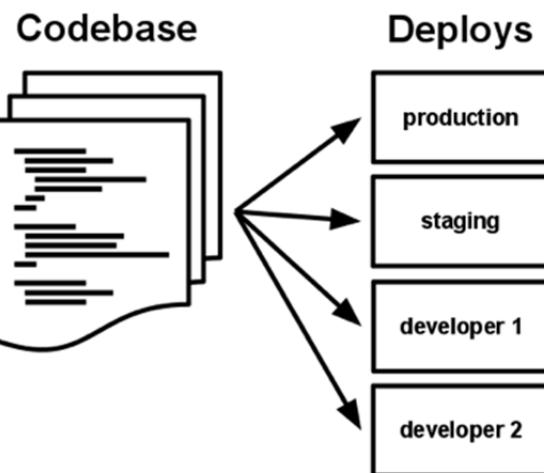
Version control systems: Git (very popular), Subversion, Mercurial.

One app ↔ one codebase.

- Do not share multiple apps within a single codebase – it complicates deploys.
- Each app evolves at its own pace.

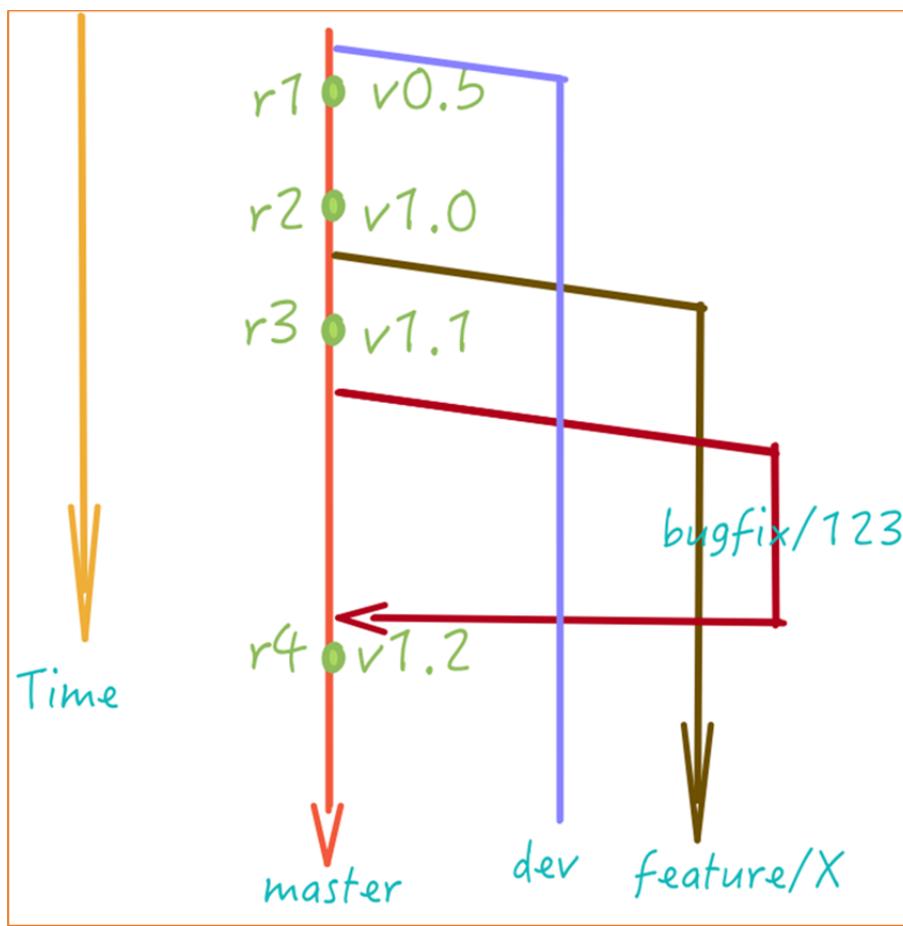
Multiple deployments are done from a single codebase.

Here we have production, staging and dev.



The same code goes through multiple environments before becoming live. Here we are showing two developers working on a feature. They are both testing it in their own setups. Probably QA (Quality Assurance) would run the same code on their environment looking for bugs. And before going to the final production, the same code might be tested on a staging area one last time.

1 – Codebase – Best Practices (git)



Here is a sample workflow for the git version control system.

Branching

It is recommended that the 'master' branch is always in a 'good' state and is ready to deploy. The 'dev' branch is used for current development work and usually considered 'unstable'. It's recommended that features be developed in their own branch. We also see bug fixes in their own branch, but quickly merged back to master branch.

Deploying

It's recommended that each deployment is tagged. So we can always 'roll back' to the last known good version in the event of some unforeseen failures.

References

- <https://www.atlassian.com/git/tutorials/comparing-workflows>

1 – Codebase – Best Practices (Cont.)

(Remember these are not rules, just guidelines. Adopt them to your needs accordingly.)

Master is always 'clean' – as in ready to deploy.

All deploys are done from the master branch.

Each deployment has a unique tag (r1, r2, r3).

- So if anything goes wrong with the latest deploy, we can switch back to 'last known good release'.

Each release can optionally have a version tag as well (e.g., v1.1).

Features / bug fixes should be done in a separate branch.

- Then merged to master when ready to deploy.

<https://sethrobertson.github.io/GitBestPractices/>

<https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices>

<https://www.atlassian.com/git/tutorials/comparing-workflows>

Adopting best practices in version control can ensure smooth deployments. Here are some best practices – from git version control system.

2 – Dependencies

Most apps depend on system libraries.

- E.g., python apps needing some libraries (compression, image processing, etc.)

Twelve-Factor apps never rely on the implicit existence of system-wide packages.

- It declares all dependencies explicitly and exactly (versions).
- Even common apps (like curl) are not assumed.
- Because they may or may not exist, and the installed versions may not be compatible.

Dependencies are packaged with the application.

- We use a 'bundler' tool completely package all files together.
- Usually bundled within app directory.

Examples of bundlers:

- Ruby has *Gem bundler*.
 - Python has *pip* and virtual environments.
-



Languages like python have a pretty sophisticated packaging / modules system. So it is easy to define all the dependencies.

However, if the app depends on system libraries (like ImageMagic, or libc++) we need to make sure those are installed as well. This is usually done by containers like Docker that ensures we have exact libraries present in the system.

2 – Dependencies (Cont.)

Here is a sample dependency listing for a python program.

requirements.txt

```
BeautifulSoup==3.2.0
Django==1.3
Fabric==1.2.0
PyYAML==3.09
Pygments==1.4
SQLAlchemy==0.7.1
```



Note how the requirements list packages and versions.

To install this using python package manager (pip)

```
$ pip install -r requirements.txt
```

When using Python it is recommended to create a virtual environment for the application. Do not install / update system modules – as it may break other scripts that depend on the system lib.

3 – Config

Configurations vary based on deployment environment.

- E.g., database host in development might be 'localhost'.
- Db host in production might be 'db1.company.com'.

Do not hard-code configs in code.

Also, it is recommended not to check-in config files into version control either.

- Sensitive information like passwords will leak.

Sometimes 'sample config files' are checked into version control.

- Developers make a copy of this sample file and customize settings in their own config file.
- This customized copy is NOT checked into version control.



There are multiple formats for specifying config.

XML, JSON, YAML, PLAIN TEXT

XML has been popular in Java environments. However, other simpler formats like YAML and PLAIN-TEXT are getting more adoption. It is very important to choose a format that is supported by your environment.

3 – Config (Cont.)

Checked into version control:	Developer 1:	Staging 1:
App/config /config.template <code>db.host=""</code>	App/config/config.template (in version control - not modified) App/config/config.local (customized, NOT in version control) <code>db.host = "localhost"</code>	App/config/config.template (in version control - not modified) App/config/config.local (customized, NOT in version control) <code>db.host = "staging1"</code>

Each developer / environment will have a customized file. This file is not checked into version control.

3 – Config (Cont.)

Twelve-Factor apps can specify config dynamically using an environment variable.

```
# here I am deploying to 'staging'  
# assume there is a file : config/staging.yaml  
  
$ ENV=staging ./deploy.sh  
  
# the app / scripts will use all settings in 'config/staging.yaml'
```

When specifying the environment before running the commands, we make sure those env settings are in effect. This allows a developer to work efficiently with many environments (dev, qa, staging).

4 – Backing Services

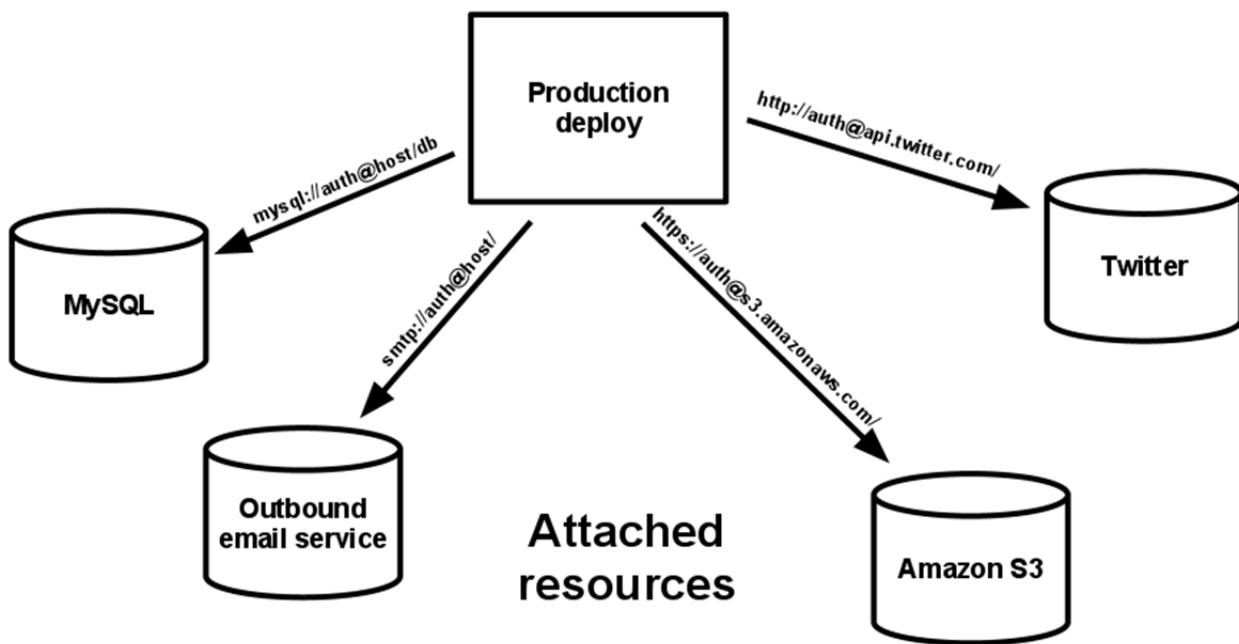
A **Backing service** is any service the app uses for operation.

- E.g., databases, email service, twitter

A Twelve-Factor app doesn't 'hard code' any backing service URLs in the code.

The same code should work in multiple environments without any code change.

- It picks up backing service URLs from appropriate config files (dev, staging, prod).



Most web applications depend on some sort of data storage / database for storing data. MySQL is a very popular relational database. It is open source and very easy to use. PostgreSQL is also very popular due to its scalability.

Now, cloud vendors are providing 'database as service' that simplifies operations (backups, restore, failover). Amazon's RDS (Relational Database Service) is one such service.

4 – Backing Service Usage

File : config/staging.yaml

```
db.host = "staging.example.com"  
db.user = "staging_user"  
db.pass = "secret123"
```

File : config/prod.yaml

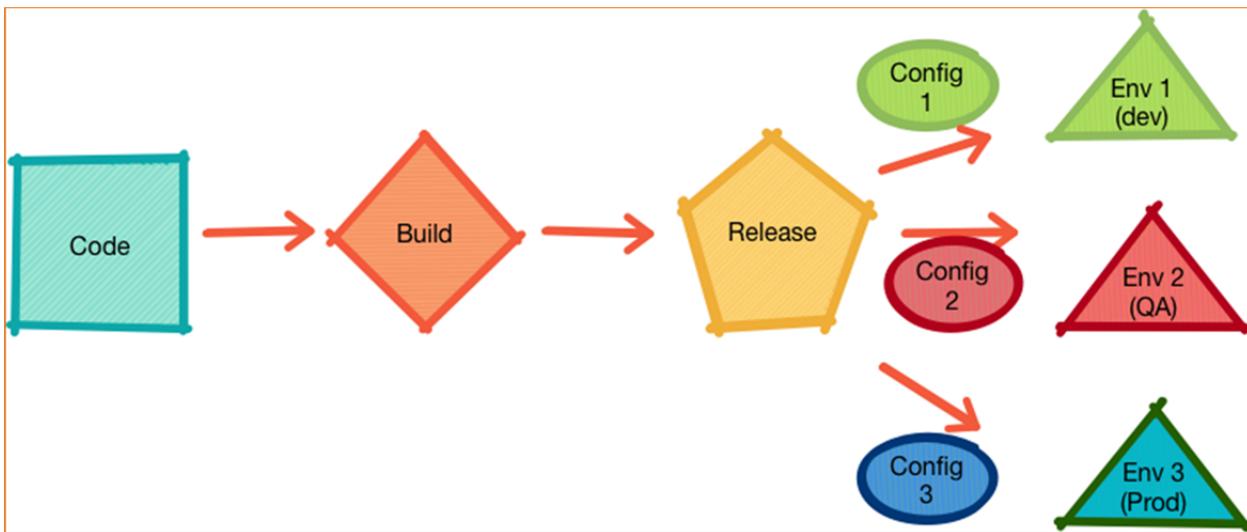
```
db.host = "production.example.com"  
db.user = "prod_user"  
db.pass = "!secret123!"
```

application code

```
# config is pointing to appropriate env config file  
db_host = config.get("db_host")  
db_user = config.get("db_user")  
db_pass = config.get("db_pass")  
Db_connection = Db.open(db_host, db_user, db_pass)
```

When we load properties from code, the right env properties get loaded (stage, prod, etc.).

5 – Build, Release, Run



This shows a typical development and deployment process. Build turns code into a release. Release is then deployed into various environments (dev, QA and prod). Usually we are referring to automation here.

5 – Build, Release, Run (Cont.)

Build: Converts code into executable bundles (called build).

Release: Combines build and configuration and deploys to environment.

Run: The app execution.

Twelve-Factor app makes a clear distinction among build/release/run steps.

No code changes in run time, because you can not propagate the changes to build stage.

Each release has a unique release-id.

- Timestamp (2018-06-01_13-04-01) or r23

Release tools should allow reverting to previous builds quickly.

- Deploy a new build.
 - Something goes wrong.
 - Should be able to revert back to last known good build.
-

Build tools for Java: Maven, Gradle.

Build tools for C: Make, Automake.

Build tools for Ruby: Ruby build-tool.

6 – Processes (Run time)

The app is executed in the execution environment as one or more processes.

The process can be a simple python script, or a complicated web application that depends on multiple resources (database, email, twitter, etc.). Twelve-Factor processes are stateless and share nothing.

Stateless

- All state and data are stored in 'backing store' (see 4).
- So if an app is re-started, it can read data from the backing store and keep continuing.

Share-nothing

- Apps don't depend on other apps.



State-less and Share-Nothing architecture allows an app to be scaled easily. When demand goes up, more processes can be spun up. When demand subsides, processes can be shut down.

<https://12factor.net/processes>

6 – Processes (Run time) (Cont.)

How about local stores like memory cache / local disk?

- Ok to use transient, intermediate data.
- For example, store uploaded file on local drive before uploading a durable storage like S3.
- A process restart can wipe local cache memory.

Sticky sessions?

- Sticky sessions mean if one process served a user request, it would also serve subsequent requests from the same user.
 - Session data is usually stored in local storage: memory / disk.
 - Sticky sessions are in violation of Twelve-Factor design.
 - We want user request to be served by any process (not just one).
 - Recommended storing session data in stores like Memcache / Redis.
-

<https://12factor.net/processes>

Memcached: Free & open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load.

Redis: Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.

Interactivity: True or False?

Question: True or False. Share-nothing apps depend on other apps.

Instructions: Use ✓ (True) or X (False)



7 – Port Binding

Twelve-Factor apps (mostly web apps) listen on a port to serve requests.

Popular ports:

- http: 80, 8000, 8080

An app can be run within another container.

- Java web apps run within Tomcat container.
- Python web apps run within Tornado container.
- PHP web apps run within Apache or Nginx web server.

Listening on a port enable one app to service another:

- For example, an email app can be a service used by another web app.
-

Apache is a popular web server that supports multiple language stacks: PHP, Python, Java, etc.

<https://httpd.apache.org/>

Nginx is another very popular web server, focused on performance.

<https://www.nginx.com/>

Tomcat is a popular as a Java web container.

<http://tomcat.apache.org/>

8 – Concurrency

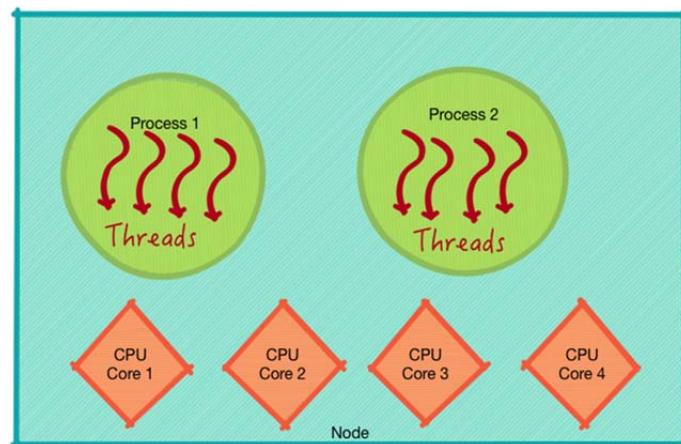
Apps run as multiple processes / threads can utilize hardware better.

Process

- 'heavy weight'.
- Multiple processes can run on same machine utilizing multiple CPU cores and other resources.
- E.g., PHP web application process.

Threads

- Threads run within a process.
- Many threads in a process.
- Threads are light weight.
- E.g., Java web application thread within Tomcat.



Unix popularized the process model. The Unix/Linux operating system can run multiple processes. Each process will have a unique id. A moderate machine can run thousands or processes.

Some processes are short-lived (like doing an 'ls' command). Some are long-lived (mailer daemon).

Threads are much lighter weight than processes. There can be hundreds or thousands of threads per process.

8 – Concurrency (Cont.)

In the twelve-factor app, processes are a first class citizen.

The process concept is closely modeled after Unix/Linux processes.

- Traditional web requests are handled by web processes.
- Long-running tasks (like sending emails, etc.) can be handled by daemon process.

A single process can run multiple threads inside it, parallelizing the workload:

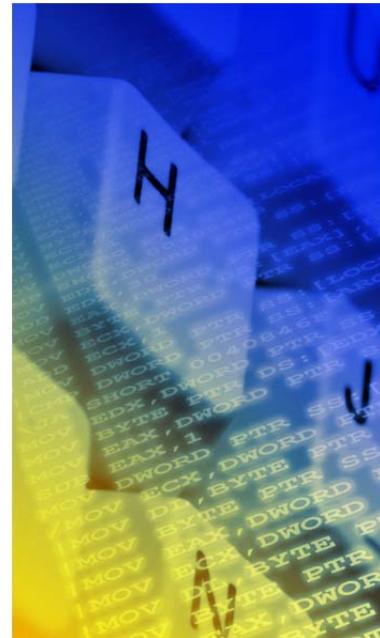
- E.g., Node.js

However, there is only so much vertical scaling, a single process can achieve.

Scaling the app:

- Run multiple processes within the same machine.
- And scale out to multiple machines.

Twelve-Factor apps should not manage daemon processes. It should use operating systems process / daemon manager.



A Linux/Unix system can run thousands of concurrent processes.

9 – Disposability

The twelve-factor app's processes are **disposable**, meaning they can be started or stopped at a moment's notice.

This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

The process should startup fairly quickly (few seconds at most).

- This allows load balancers to scale up processes as the demand increases.
- Also, processes can be moved to another machine.

The process should shutdown quickly and gracefully as well.

- Used to restart / redeploy apps quickly.

Twelve-Factor apps are designed to handle unexpected / sudden terminations.

- To prevent data loss due to sudden crashes, save the data in backup store.
-

We use Unix signal SIGTERM to tell processes to shutdown. When a process gets the signal, it should finish serving current requests and shutdown quickly.

10 – Dev/Prod Parity

Traditionally there is a substantial gap between development and production.

There are three main areas where we see this gap.

The Time Gap	The Personnel Gap	The Tools Gap
<ul style="list-style-type: none">Developer can work on features that can take days / weeks / months before it goes into production.	<ul style="list-style-type: none">Developers write code, ops-engineers deploy it.	<ul style="list-style-type: none">Developer stack: light-weight httpd, SQLite on Mac OSX.Production stack: Nginx, MySQL on Linux.

Dev/Prod parity is a serious obstacle to doing frequent releases (a practice followed by many companies). If a developer has a very different environment than in production, this will lead to failed applications upon deployment. For example, we have all heard the 'but it worked on my machine' explanation, right?

10 – Dev/Prod Parity (Cont.)

The twelve-factor app is designed for **continuous deployment** by keeping the **gap between development and production small**.

Make the time gap small: A developer may write the code and have it deployed hours or even just minutes later.

Make the personnel gap small: Developers who wrote the code are closely involved in deploying it and watching its behavior in production.

Make the tools gap small: Keep development and production as similar as possible.

	Traditional app	Twelve-factor app
Time between deploys	Weeks	Hours
Code authors vs. code deployers	Different people	Same people
Dev vs. production environments	Divergent	As similar as possible

As you can see, Twelve-factor app methodology encourages frequent deploys. This minimizes the friction of deploying new features and bugfixes to production systems. We have good software development practices to minimize the gap. For example, Python developers use 'virtual environments' that match production setups.

10 – Dev/Prod Parity (Cont.)

Twelve-Factor app principal urges developers to use same backing store in development and production.

- E.g., if MySQL is used in production, developers should also use MySQL on their development machines.

Because sometimes tiny discrepancies in data storage can lead to problems that are hard to trace.

- E.g., how integers are represented (max / min values).

Running stores used in production (MySQL, PostgreSQL) on dev machines is easier now.

- Most of them can be installed and run natively.
- Or use light-weight containers like Docker and Vagrant to run the app.

It is even recommended to use the same version of backing store in dev and production.

- There can be subtle version incompatibility bugs.
-



Traditionally VMs are used to run isolated environments on a dev machine. But VMs are heavy in resource utilization. Docker is a modern container format that is light-weight and works universally.

<https://www.docker.com/>

11 – Logs

Logs provide visibility into the behavior of a running app. Logs are commonly written to a file on disk (a “logfile”).

Logs are critical for:

- Diagnosing bugs / exceptions happening in the program.
- Understanding app behavior – INFO level logs, metrics.

Twelve-Factor app is highly encouraged to log:

- Exceptions and errors.
- INFO, DEBUG level messages to log the working of the app.

However, the app should not manage log files:

- Use a logging tool.
 - Logging tool will save the logs to a destination (disk file, database, etc.) according to the configuration.
-

We recommend using loggers in your program from day one. Most developers start with PRINT() and before long there are too many PRINT statements littered in the code.

11 – Logs (Cont.)

Don't log using PRINT():

- It is hard to turn on / off.
- And selectively turn on logging for a component.

Loggers

- Java: Log4j.
- Python: PyLogger.
- Log Levels: DEBUG > INFO > WARN (default) > ERROR.

Loggers can send the output to any number of destinations (or a combination).

- Disk files, Database, Network service, a message queue.

Loggers can also 'rotate' files – based on timestamp or size. And loggers allow us to turn logging for separate components on / off. E.g., Component A = LogLevel.DEBUG, Component B = LogLevel.INFO



There are usually more than one logger per language. We recommend using one that has wide adoption and an easy to use API.

12 – Admin

There are steps needed to set up the application so it is ready to run.

For example:

- Setting up database schema.
- Seeding some initial data (like zip code lookup data, etc.).

Setup scripts are part of app. Setup scripts use environment config to setup. Include 'one time' scripts too.

For example:

- `php scripts/fix-bad-records.php`



Most admin tasks can be done as 'user'. We don't need 'admin' privileges. It is recommended to run web apps NOT as admin user. Usually a special user is created to run web apps.

Discussion: Twelve-Factor Methodology

Questions:

- What are the major principles of code writing, building, and testing, as formulated in the twelve-factor app methodology?
- Can you describe where you apply these in your projects?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Discussion: Twelve-Factor Deploy, Configure and Run

Questions:

- Questions:
- What are the major principles of code writing, building, and testing, as formulated in the twelve-factor app methodology?
- Can you describe where you apply these in your projects?
- Instructions: Enter your answers into Chat (All Participants).



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Module Summary

Now that you have completed this module, you should be able to:

- Describe best practices outlined by Twelve-Factor app methodology for building highly available, highly scalable apps.
 - Recognize problems that the Twelve-Factor app methodology is designed to solve.
-

Now that you have completed this module, you should be able to:

- Describe best practices outlined by Twelve-Factor app methodology for building highly available, highly scalable apps.
- Recognize problems that the Twelve-Factor app methodology is designed to solve.

API

Module 3

Module Objectives

After completing this module, participants will be able to:

- Identify the elements of data communication.
 - Compare and contrast popular serialization formats.
 - Differentiate between protocols.
 - Understand API service frameworks.
 - Describe how open standards and RESTful web interfaces have transformed enterprise applications.
-

What is an API?

API stands for Application Programming Interface.

- It is a means for allowing developers (not users) to access functionality.
- A good API should shield the user from the app internals.



APIs have become very popular as a best practice for software development, and they are a means for allowing developers (not users) to access functionality; developers can call them right from their code. They implement the idea of “decoupling” which results in many benefits. We’ll cover many of these benefits over the rest of this module.

Early History of APIs: Libraries



Early libraries consisted of a list of procedures

- Well-defined parameters
- Return types



Early libraries linked statically

- Included object file at link process



Dynamic linking or late binding

- Windows DLL/Unix .so
- Allows code to be linked at runtime

Libraries are the first form of API. It was a way for a vendor or programmer to open the functionality of the application. Without the benefit of a permanent internet connection, it is easy to see how things needed to be this way.

API Forms

Object Oriented APIs	Remote Procedure Calls: RPC	Message Queuing	Web Services	Web APIs
<p>Classes rather than Procedures</p> <ul style="list-style-type: none"> • Data + Methods • Remote objects / Remoting <p>Use IDL to provide class metadata</p> <ul style="list-style-type: none"> • Examples: CORBA, DCOM <p>Tended to be heavyweight, and fragile</p> <p>Not really cross-platform</p>	<p>Distributed Systems: Need to remotely execute code</p> <p>Interface Definition Language (IDL)</p> <ul style="list-style-type: none"> • Facilitates communication from client to server <p>Early RPC has no failure protocol</p> <ul style="list-style-type: none"> • No guaranteed delivery 	<p>IBM MQSeries: Saw communication as series of messages</p> <ul style="list-style-type: none"> • Allowed reliable delivery • More failure tolerant <p>Decouple Client from Server</p> <ul style="list-style-type: none"> • More reliable • Loose Coupled <p>Message Infrastructure has steep requirements.</p>	<p>Web technologies like HTTP, HTML, XML used for APIs</p> <ul style="list-style-type: none"> • Open Standards, Cross Platform <p>Implement RPC and Object Remoting via HTTP/WWW.</p> <p>Service-Oriented Architecture</p> <ul style="list-style-type: none"> • Sees distributed application as services • Communicate via XML (SOAP) • XML-based IDL (WSDL) <p>Problems:</p> <ul style="list-style-type: none"> • Complexity • Fragility 	<p>The Web is built on open standards</p> <ul style="list-style-type: none"> • HTTP Protocol • HTML/CSS/JavaScript client-side <p>APIs can use the open protocols</p> <ul style="list-style-type: none"> • So ANY user can use it <p>Easier to use than TCP/UDP Programming</p> <ul style="list-style-type: none"> • HTTP ports usually open on network firewalls • Able to communicate over public internet (if desired)

Class Libraries: Class Libraries are the OOP version of earlier libraries. Instead of a list of procedures, class libraries store classes of objects: Data and methods. With OOP's principles of inheritance, polymorphism, and encapsulation, these objects allow for better code-reuse and more modular design than traditional libraries.

The problem with class libraries as an API is that extending OOP principles to distributed systems are difficult, heavyweight, and fragile. Security is another major concern, as remote users can maliciously use application objects.

Another issue is distributing across platforms. Traditional class libraries in Java (to choose one example) are of no use to a developer in .NET. Communication between systems requires a different sort of API.

Remote Procedure Calls (RPC): Remote Procedure Calls (RPC) are a procedural (rather than OOP) form of API. Database stored procedures are similar. The idea behind it is that it allows a network API to be published that allows users to use the service.

The RPC is still the basic paradigm around distributed APIs. Ironically, this old, seemingly outdated mechanism is making a comeback, because many frameworks now exist to provide this useful functionality.

The problem with the RPC is how do we ensure failure. If the server is "down" the RPC fails. If the client is down, again, it fails.

The RPC also is limited because it is essentially a one-way request-response mechanism. There's no direct way for the server to push messages to the client, and even if there was a way, there's no way to guarantee that clients would receive such messages.

Message Queuing: Messaging fundamentally rethinks how an API works. Rather than seeing API as a series of calls, it is a series of messages.

The Actor Model is a good way of viewing this. The Actor model sees the interaction as a conversation rather than as an imperative framework. To make the actor model work, we need some kind of message queuing.

Distributed systems are inherently lossy and unreliable. Message queuing is a necessary part of making this a reliable connection.

Web Services: Web Services first arose out of necessity. Object remoting systems couldn't get through web firewalls. Objects themselves weren't cross-platform. We needed a way to communicate over the web, in a platform-independent way. So, we have Web Services. Web services tunnel through the web, so they are web-friendly. They use XML, which for all its many faults is an open standard and supported by all systems. Web services are the most popular today.

Web APIs: Web-oriented products needed a way to communicate via APIs. For these products, the web WAS the medium of delivery of the application, and everything had to be accessible over the web.

This is different than the XML Web services that merely used the web as a convenient transport mechanism and serialization format. Web APIs are a way to programmatically interface with the Web itself.

Google Maps and Twitter

Google Maps

Google Maps is an early Web API (2006)

- Early users "hacked" Google Maps to create location applications.
- "Where's the closest XYZ store?"
- Google Maps Mash-ups.

Google Maps API

- Google released a public Google Maps API.
- Anyone could call the API and integrate Google Maps.



Twitter

Twitter is the first modern Web Service Application

- The **primary** interface is the API, not the website.
- Users connect to the Twitter "firehose" via APIs.
- Uses JSON to communicate.

Example Use Case: Sentiment Analysis on Stock

- Users search for tweets mentioning a stock.
- Score sentiment (positive/negative) about that.
- Enable stock trading from results.



Google Maps: Google maps represented an early web UI. For the first time, a web application was able to be embedded into other products. At first, this was more of a "hack" -- developers would use the Google Maps application itself but "mash up" in new ways -- for example, to show the locations of coffee shops, etc.

Google eventually recognized this and created a public API for its map service, which gave rise to Google's Cloud Web APIs.

Twitter: Twitter is the first modern web service application for several reasons:

- First, it was an application that clearly differentiated between a website and the application. Twitter has many "clients", including a web client, but that is quite separated from the core functionality of twitter. It's one of many windows into the twitter application.
- Second, it uses HTTP (REST) as its main API. While this is standard today, at the time most web services used antiquated protocols like SOAP. By using JSON instead of the cumbersome XML, it made it much easier to process results, particularly client side.

Protocols and Data Formats

Communication Requirements

Claude Shannon (Bell Labs):

- "Father of Information Theory"
- 1948: "Mathematical History of Communication"



Source: https://en.wikipedia.org/wiki/Claude_Shannon

Claude Shannon formalized notions that people use to take for granted. For example, what is information? Well, one bit of information is defined as the information that I get when I ask a "Yes or No" question and I get back the answer.

Elements of Communication:

-  **Source:** Generates message data.
 -  **Transmitter:** Produces signal.
 -  **Channel:** Medium for transmission.
 -  **Receiver:** Reconstructs message from signal.
 -  **Destination:** Entity for which message is intended.
 -  **Message:** Information transmitted.
-

These bullet points introduce the major definitions in cybernetics, which we then build upon. Bandwidth of the channel is the biggest limitation, and we get around this by making data as local as possible.

Interactivity: Matching Question

Question: Match each element of communication with its description.

Instructions: Raise Hand to volunteer and then use the Line tool.



Channel	Generates message data
Source	Reconstructs message from signal
Message	Medium for transmission
Receiver	Information transmitted

Communication Terms



Serialization (the transmitter):

- Process to transmit data structures or state into a format for storage or transmission.



Deserialization (the receiver):

- The reverse process.



Protocol (the communication process):

- A set of rules allowing:
 - Two or more entities to transmit a message.

Here we introduce the technical terms of how information exchange happens. These definitions will be later translation into specification and then into code.

Protocols

SOAP, REST, and GRPC



SOAP uses XML
Describes service using WSDL file
Not often used in modern era



REST is technically not a protocol
REST can use any serialization

- JSON
- Protobuf

REST does NOT use RPC



Protobuf is only a serialization format, not a communication protocol
The communication protocol Google uses for Protobuf
Brings Protobuf in line with Thrift, Bond, or Avro

SOAP: SOAP was the first data interchange format for web services. Then Amazon-designed REST replaced it.

REST: The RESTful protocol was designed by Amazon and is now the de facto standard. It wins due to its simplicity, and today, due to its widespread use. It is easy to use and there are multiple libraries for the implementers.

GRPC: One more differentiate between a serialization format (that is how you write the data) and a communication protocol (that is how you send that data to the client). Developers use Protobuf and do not work directly with GRPC, so we won't be describing the GRPC protocol in more detail.

SOAP Web Services

SOAP (Simple Object Access Protocol) was invented at Microsoft in 1998

- Submitted for standardization in 2000.

Relies on Open Standards:

- Uses HTTP (or UDP or SMTP, but not commonly)
- Uses XML for serializing Data

Provides a way to loosely couple systems

- Bypasses security problems with COM/DCOM
- Cross platform beyond Microsoft Technologies
- Clients can read the WSDL (Web Service Description Language)

Uses RPC (Remote Procedure Call) model

- Client makes a procedure call to the server.



SOAP isn't really used much anymore, though there are a huge number of legacy applications that use it. Mainly, SOAP is of interest as an iteration of modern web services, that are going to be using RESTful principles.

The method of using TCP sockets for "remoting" between systems was fundamentally broken. Corporate firewalls locked down various TCP ports and the concepts of making distributed applications based on TCP sockets weren't going to work. Somehow, we needed services that could talk on open standards using HTTP protocols and standard ports, which are guaranteed to be open.

By today's standards, the idea of using a technology like DCOM for distributed systems was never going to work well. The very definition of "tightly coupled" -- DCOM was single-platform, proprietary, unsupported by other vendors, and generally created brittle, fragile systems that didn't work well as distributed systems.

Web Services, even in their most basic form, solved all of these problems. They were based on open-standards, cross-platform, and allowed loose coupling between systems.

The RPC model works well for some systems, as we can see that it has found its way even into much more modern systems like Thrift.

SOAP and State and Disadvantages

Serialization Specific to One Platform

- SOAP web services can be stateful.
- State stored per-user session in DB.

Disadvantages

- XML is bulky, verbose, and inefficient.
- Roles fixed as client and server.
- Web Service APIs quickly grew out of control, with huge WSDLs.
- Nonstandard interaction model.

Statefulness, while discouraged in REST systems, is ubiquitous in SOAP. There are actually two kinds of state: a server-side state tied to the user's session, and a client-side state stored in hashed browser data, called the ViewState. The latter is almost never used by modern APIs, and the former has significant scaling limitations, which is why REST discourages statefulness.

SOAP was revolutionary in its time, but as a first generation web-service technology there were many ways in which it did not lead to its promise. XML now has a reputation for verbosity, unreadability, and clumsiness, but at the time it was the epitome of openness.

The real problem with SOAP web services is that there was no standard way to access one. Every SOAP web service would have its custom API, which although was available to the WSDL, meant that applications essentially had to be custom written to use the API.

REST

REST:

- REST is: Representational State Transfer.
- It is NOT a protocol, but an architectural pattern.



REST is one of the organizational paradigms of the modern web service movement. Although RPC models are still very popular, it is safe to say that most developers have a certain expectation that a modern API will correspond to REST. We will be covering REST in much more detail later in this module.

Interactivity: Poll Question

Question: Which protocol is only a serialization format, not a communication protocol?

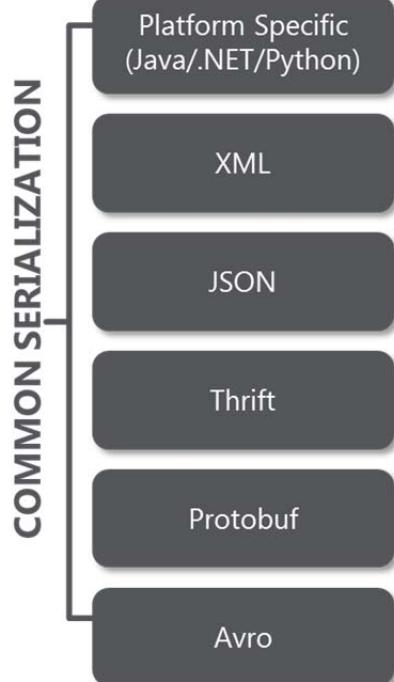
- A. REST
 - B. SOAP
 - C. GRPC
 - D. XML
-



Serialization

Serialization Options

There are a number of ways we can serialize our data.



Serialization is technical drill-down into the implementation of communication. All ‘information’ or ‘messages’ need to be serialized into bytes that will be communicated in the information exchange. Popular protocols use XML or JSON.

Platform-Specific Serialization

Serialization Specific to One Platform

- Java Serialization
- Python Pickle
- .NET Remoting

Disadvantages

- Typically inefficient and slow
- Not portable
- Not cross-platform

Depending on the platform where you are implementing serialization, there will be some protocols that are not portable between platforms.

XML and Example

XML was historically used, especially with SOAP.

Not frequently used:

- Complex
- Verbose
- Inefficient
- Not very readable by humans



Example:

```
1 <?xml version="1.0"?>
2 <soap:Envelope
3 xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
4 soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
5 <soap:Body>
6   <m:GetPrice xmlns:m="https://www.w3schools.com/prices">
7     <m:Item>Apples</m:Item>
8   </m:GetPrice>
9 </soap:Body>
10 </soap:Envelope>
```

SOAP XML Services encapsulate data in the xml wrapper. The XML wrapper is a document format rather than a record format so the entire XML document must be completely parsed before any of the data can be read.

In this example, we are specifying the XML namespace (soap), and the encoding style (soap) based on standard w3 descriptions which must be read by the recipient in order to correctly parse the data.

In the Body, we are making a request to the RPC method called GetPrice, with a parameter to the call as Apple. Being an RPC model, the requester will be expecting a response from the recipient.

While SOAP works fine for this simple example, it is cumbersome and verbose, and somewhat brittle given the fact that an error in reading the XML request renders the entire document unreadable.

However, it is an open-standard, and is well understood.

JSON and Example

JSON: JavaScript Object Notation

Advantages:

- Human readable
- Self-describing
- Easy to parse



Disadvantages:

- Verbose
- Slow to scan sequentially

JSON can be compressed (ZIP, gzip, etc.)

Example:

```
1 {  
2   "id":1,  
3   "name":"Apple",  
4   "price":1.0  
5 }
```

JSON is the new darling. Today, programmers will always choose it by default when they are looking for text-based interchange format, and you don't have to have every field present in every record.

This message is functionally identical to the previous example. However, being far more compact (relatively speaking), and simple to parse makes it far preferred. It's also quite easy to "eyeball" by humans.

Despite being relatively more compact and readable than XML, it is still not a particularly efficient means for storing or reading the code. Querying data from a JSON file will prove to be far inferior to many of the binary formats we use for storing data. These disadvantages mean that JSON is often compressed for efficiency's sake.

BSON

Binary JSON:

- Uses packed binary format
 - Uses a "length" field to facilitate faster scanning
 - Row-based
-

BSON { 01010100
11101011
10101110
01010101 }

BSON is one of the first binary protocols that were designed specifically for programmers. There are libraries available to convert Java objects to BSON, and the like.

Protocol Buffers (protobuf)

protobuf:

- Developed by Google
 - Both a format and a protocol
 - Binary
 - No RPC, but gRPC framework exists to do RPC with protobuf
-



Protobuf was created by Google and immediately became extremely popular. It is the internal preferred format for Amazon Machine Learning, due to its speed. There are many libraries for the protobuf protocol available today.

Thrift

Thrift:

- Developed by Facebook
 - Both a format and a protocol
 - Binary
 - Supports versioning
-

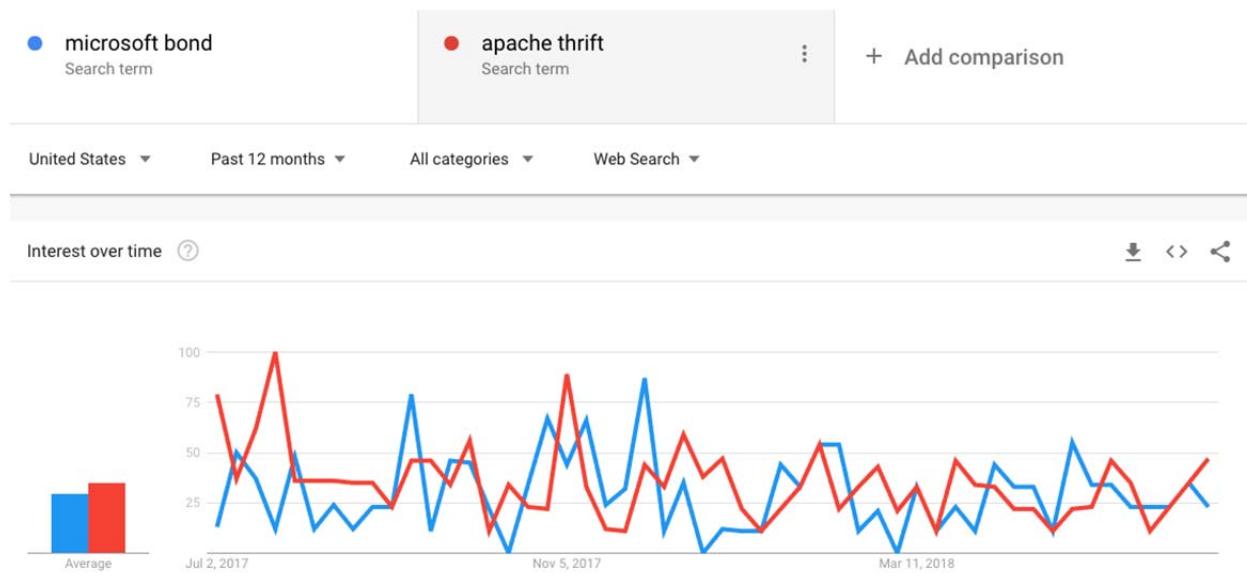
Apache Thrift™

Thrift is the first very popular format for exchanging data between different programming languages and between different systems. For example, many NoSQL implementations support Thrift as the first data exchange protocol. Thrift was developed by Facebook and is the default for a lot of Big Data as a result.

Bond

Bond:

- Developed by Microsoft
- Both a format and a protocol/RPC
- Binary

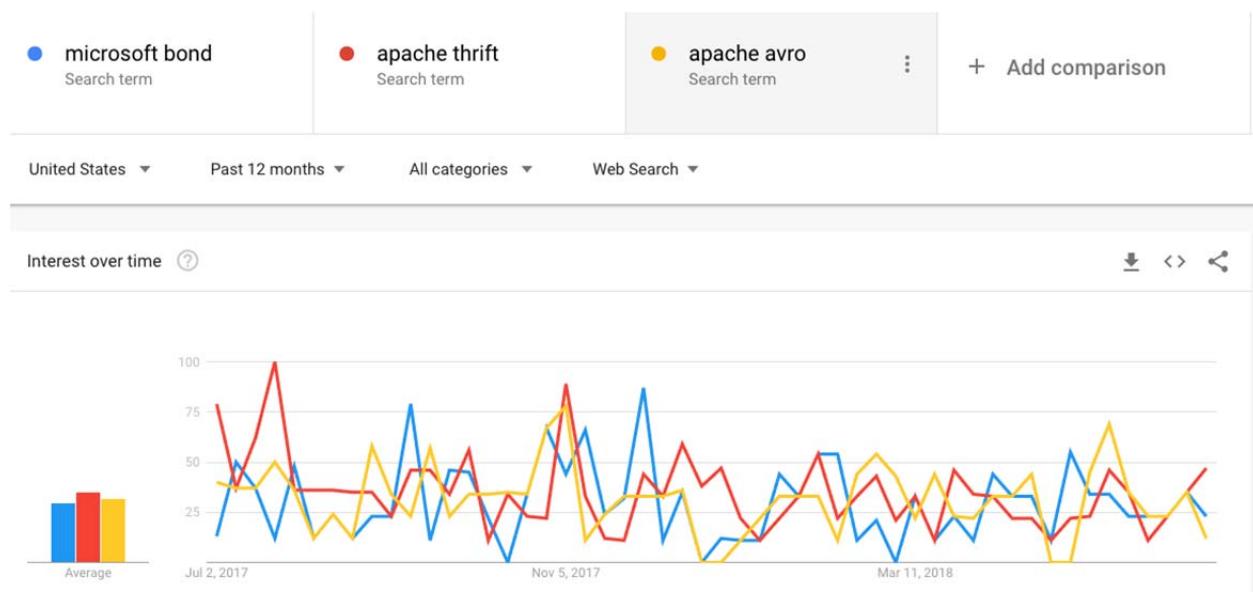


Bond was developed by Microsoft. It is interesting to note that Bond and Thrift are about equally popular, which is not usually the case.

Avro

Avro:

- Mainly used in Hadoop/Big Data
- Supports RPC
- Self-describing
- Advanced schema evolution features
- Row-based



As you can see, Avro format competes with both Bond and Thrift. It is self-describing, so you can read the description of the data from the data itself.

Parquet

Parquet:

- File and Serialization format
 - Uses Thrift for RPC/protocol
 - Columnar
 - Limited schema evolution
 - Self-describing
-



Parquet is both a file format and a serialization format. It was designed by Cloudera and it is their preferred format. However, they will read and write in their competing format, ORC.

Comparison

	Protocol	RPC	Human Readable?	Self Describing	Binary	Compressed	Schema Evolution	Storage
XML	No	No	Yes	Optional	No	No	No	Row
JSON	No	No	Yes	Optional	No	No	No	Row
BSON	No	No	No	Optional	Yes	No	No	Row
Thrift	Yes	Yes	No	No	Yes	No	No	Row
Bond	Yes	Yes	No	Optional	Yes	No	Yes	Row
Protobuf	Yes	No	No	Optional	Yes	No	Yes	Row
Avro	Yes	Yes	No	Yes	Yes	No	Yes	Row
Parquet	No	No	No	Yes	Yes	Yes	Limited	Column

In this summary table, you can see the relative pluses and minuses of all formats. More correctly, you can see advantages and disadvantages, all relating to specific needs.

Interactivity: Matching Question

Question: Match each serialization option with its description.

Instructions: Raise Hand to volunteer and then use the Line tool.



JSON

Uses a packed binary format, is row-based, and uses a “length” field

Parquet

Developed by Microsoft and is both a format and protocol/RPC

Bond

Human readable, self-describing, and easy to parse

BSON

File and serialization format that uses Thrift for RPC/protocol

Microservices

Monolithic Services vs. Microservices

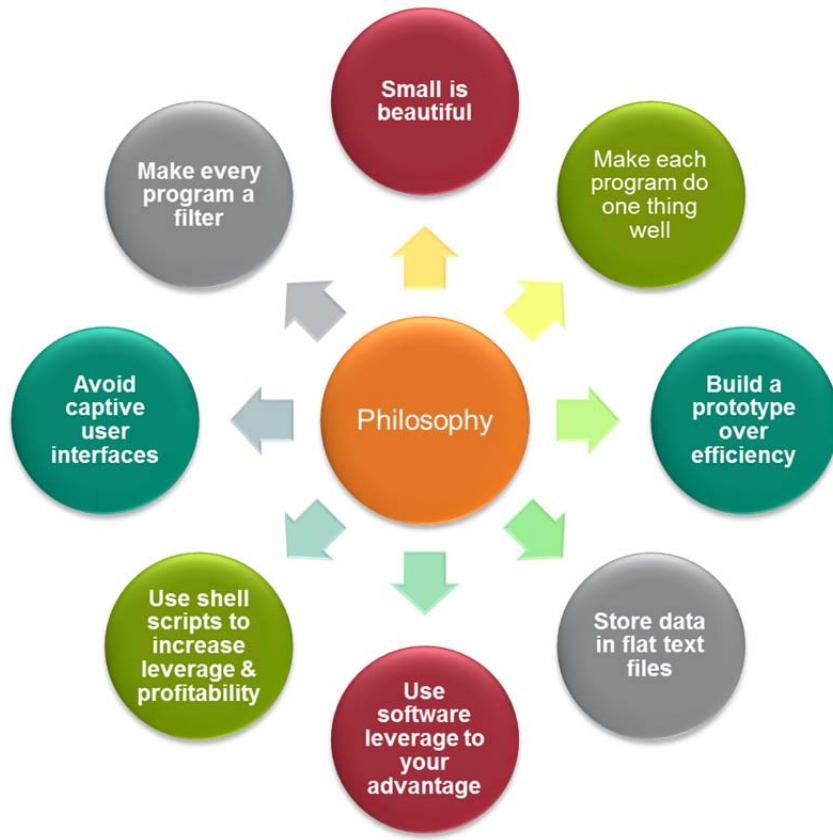
Monolithic Services	Microservices
Traditional services are defined as monolithic	"Do one thing and do it well" <ul style="list-style-type: none">▪ Each microservice does minimal level of useful functionality
Monolithic service does all of the following: <ul style="list-style-type: none">▪ Presentation▪ Business logic▪ DB access▪ Application integration	Application composed of interconnecting services: <ul style="list-style-type: none">▪ Services communicate via RPC or Messaging▪ Services do their own persistence
Disadvantages of monolithic services: <ul style="list-style-type: none">▪ Hard to fully understand▪ Difficult to scale▪ Reliability	

Monolithic services are what we're trying to move away from. Monolithic services are built from the "do everything" model, which has all of the drawbacks mentioned above.

Microservices is a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight.

Microservices have become so popular in the recent years that one can safely state that it is now a basic principle of building modern software.

UNIX Philosophy



Why does the UNIX Philosophy matter? It is because this philosophy led to the creation of the most successful operating system today: Linux.

Not to diminish the importance of Windows, but today, the vast majority of systems today are built on top of UNIX (the desktop PC being the primary exception). Linux, Mac OS X, Android, iOS, and most embedded devices all use a UNIX-like OS. There is a reason for its popularity in building modular systems.

Distributed Systems

Microservices are the “UNIX philosophy”

- They are the philosophy in distributed systems and in services.
 - Does this mean that I can only do this on Linux?
 - NO!
 - “UNIX philosophy” can be used on Windows, too.
 - It’s just an idea.
-

Microservices are a new way of practically building distributed systems. But many of the distributed systems ideas come from operating system design. One of the most successful OS's has been Linux. It is not surprising then that in building microservices architecture, much has been borrowed from Linux.

Microservice Advantages and Architecture Advantages

Microservice Advantages

- Scalability:
 - Functionality is already distributed.
 - Easy to break it up during deployment.
- Testability:
 - Each portion is minimal, so easy to test.
 - TDD

Microservice Architecture Advantages

- Each microservice is relatively small.
- Easier for a developer to understand.
- The IDE is faster, enhancing developer productivity.
- The web container starts faster, making developers more productive, and speeding up deployments.
- High rate of change.
- Low cost of change.

TDD stands for Test Driven Development. TDD and microservices are an especially good fit.

Test Driven Development is, quite simply, a test-first approach to development. Tests should be done concurrently with requirements, so as requirements are defined, then tests plans are designed to test those requirements. As functionality is implemented, then tests that start out having failed will pass.

Why is this a good thing, and what does this have to do with Microservices?

- Smaller functionality (Microservices) is easier to test
- Having robust testing of all requirements means that services can be developed in isolation.
- Test-First functionality ensures that service components are correctly talking to one another.

Test-driven development or TDD is a development philosophy that emphasizes very short development cycles. With

A good reference: <https://www.ness.com/microservices-architecture-and-design-principles-2/?cn-reloaded=1>

The 5 Principles of Microservices Design and Implementation

Anyone implementing microservices will do well to be mindful of the following principles

Elastic	Resilient	Composable	Minimal	Complete
<ul style="list-style-type: none"> Must be able to scale up or down Multiple stateless instances of service Routing and load balancing Registration, naming, and discovery 	<ul style="list-style-type: none"> Service will have multiple Instances High availability <ul style="list-style-type: none"> Redundancy Fault Tolerance No single point of failure <ul style="list-style-type: none"> No one service is indispensable Load/Risk distribution Service instance dynamic 	<ul style="list-style-type: none"> Common, uniform interface REST principles Composition patterns: <ul style="list-style-type: none"> Aggregation Linking Caching Proxies Gateways 	<ul style="list-style-type: none"> The "Micro" in Microservices! Entities should be cohesive SRP (Single Responsibility Principle) <ul style="list-style-type: none"> One business function Not necessarily small in size <ul style="list-style-type: none"> (See Complete) But as small as possible 	<ul style="list-style-type: none"> Minimize coupling with other services <ul style="list-style-type: none"> Tight Coupling limits reusability. Should fully accomplish business function <ul style="list-style-type: none"> Don't "split" services for the sake of it Each module is as big as it needs to be "Micro" doesn't always mean small

Elastic: These are the building blocks which are not obvious and which help build good microservices. Include these into your design.

Resilient: These principles explain how you can make your microservices resilient. This means building reliable systems out of non-reliable components. That is how modern software systems are built. Amazon mantra: plan for failure and then nothing fails.

Composable: REST is a good example of an enabling technology. Take its principles and apply them to your microservices.

Minimal: This is the basic lesson of microservices. SRP (Single Responsibility Principle) will do well in many other areas of software, but in microservices, it is essential in defining the scope of each of the microservice. It simplifies writing tests as well.

Complete: This is the last and final principle of microservices construction. “Decouple” is the key principles of AWS services construction. However, there are some ‘gotchas’ on the way, and the bullet points here explain when not to be a stickler for “micro” and when to make a microservice reasonably big.

JAX-WS and JAX-RS

JAX-WS

- Java API for XML Web Services (JAX-WS)
- Java API for SOAP
- Part of Java EE platform
- Metro: Reference Implementation for JAX-WS
 - Demonstration

JAX-RS

- JAX-RS is an API, not an implementation
- Jersey is a reference implementation of JAX-RS

JAX-WS: We mentioned the SOAP protocol before. However, when you are programming in Java, you need a library that will form the SOAP request for you. One such library, called JAX-WS, is included into the standard Java EE. Both Java EE and SOAP are on their way out, even though you will still find jobs using these technologies.

JAX-RS: As we mentioned, JAX-RS is an API, the main subject of this class. However, it is not a working service that would implement it. Therefore, there was a need for a reference implementation. This is Jersey.

Jersey was introduced in order to simplify development of RESTful Web services and their clients in Java, a standard and portable JAX-RS API has been designed. Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.

Microservices Patterns

What is Spring Framework?

- Spring Framework is a suite of Java Frameworks
- Based on Inversion of Control Container
- Dependency Injection



Spring was a reaction to J2EE being too bulky and too corporate. The company behind it, SpringSource, was acquired by VMWare for \$420 million in 2009.

Over the years, original Spring added many frameworks, one of them being Spring for Hadoop.

Inversion of control means that instead of specifying the dependencies in the source code, we set them in configuration files. Thus, one can create many versions of the application by changing the configurations and keeping the same code. This leads to less bugs.

Dependency injection is a technique for one object to supply the dependencies of another object. This fundamental requirement means that using values (services) produced within the class from new or static methods is prohibited. The client should accept values passed in from outside. This allows the client to make acquiring dependencies someone else's problem.

Interactivity: Poll Question

Question: What is dependency injection?

- A. Dependencies are set in configuration files.
 - B. Services have multiple instances.
 - C. One object supplies the dependencies of another object.
 - D. A reference implementation is required.
-



Inversion of Control Container

Spring is the most popular IoC in Java

- Allows software to automatically modify itself
- Hollywood Principle:
"Don't call us; we'll call you."



IoC stands for inversion of control, which was introduced on the previous slide.

Advantages of IoC

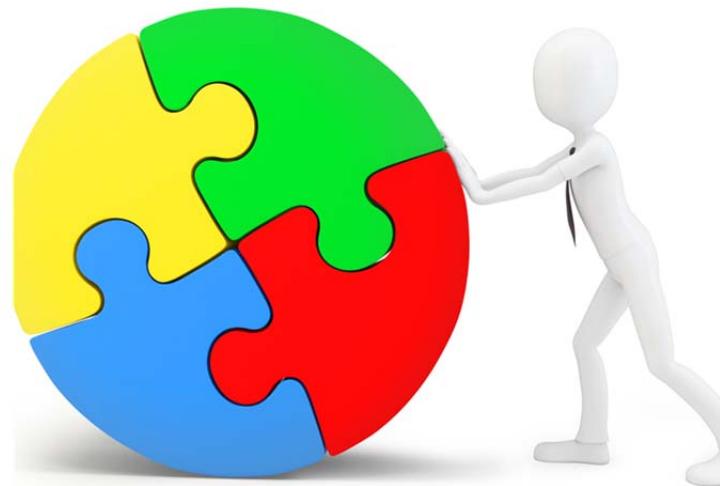
Decouple execution from implementation

Focus on module or task

Rely on contracts

Prevent side effects

Much easier testing



'Coupling' is a term that describes the relationship between two entities in a software system (usually classes).

When a class uses another class, or communicates with it, it's said to 'depend' on that other class, and so these classes are 'coupled'. At least one of them 'knows' about the other.

The idea is that we should try to keep the coupling between classes in our systems as 'loose' as possible: hence 'loose coupling' or sometimes 'decoupling' (although in English 'decoupling' would normally mean 'no coupling at all', people often use it in the microservices context to suggest 'loose coupling between entities').

Microservices and Containers

Container frameworks facilitate microservices.

Kubernetes PODs:

- Allows groups of containers to run together.
- Facilitates microservices.



Microservices may be a great design principle. But “implementation is left as an exercise to the reader.” Without good support, microservices would be just a dream. Kubernetes provides a good platform for microservices implementation. This will be discussed in later modules, and will also be covered when we move to the class on kubernetes.

REST

What is REST?

- REST stands for REpresentational State Transfer.
 - Rest is an Architectural Pattern.
 - NOT just a protocol.
 - NOT just a specific implementation.
-

Representational State Transfer (REST) is an architectural style that defines a set of constraints and properties based on HTTP. Web Services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations.

For more information, see https://en.wikipedia.org/wiki/Representational_state_transfer

REST Principles

Performance: The communication style proposed by REST is meant to be efficient and simple.

Scalability: The simple interaction proposed by REST greatly allows for this.

Simplicity of interface: A simple interface allows for simpler interactions between systems.

Modifiability of components: The distributed nature of the system, and the separation of concerns proposed by REST, allows for components to be modified independently of each other at a minimum cost and risk.

Portability: REST is technology- and language-agnostic, meaning that it can be implemented and consumed by any type of technology.

Reliability: The stateless constraint proposed by REST allows for the easier recovery of a system after failure.

As we already mentioned, REST is a collection of design principles, not even an architectural pattern, and definitely not a library. On this slide, these principles are enumerated and explained.

A Case for REST

At this ecommerce company X, we have two systems



Order Processing System



Credit Card Processing System

The Order Processing System needs to communicate with the Credit Card Processing System to complete the order.

- How do they communicate?

Usually this is done via either a database or a message bus.

As a starting point of building the case for rest, this slide presents a real-world scenario. Two departments of a company have disparate systems of processing. They do not use a common communications method nor data storage. Both were developed ad-hoc, without a thought given to later integration. How should they communicate?

A Case for REST (Cont.)

- Applications that directly interact with each other are called 'tightly coupled'.
 - 'Tightly coupled' apps are usually hard to maintain.
 - They can be fragile. If one app's library or database schema changes, the other app might break.
 - Evolving applications independently can be challenging.
 - Updating new versions of applications involves a lot of testing among applications.
 - Also, we may want to expose our services to the outside world.
 - Called **Web Services**
 - Examples of Web Services:
 - Expedia exposes its data via public APIs
 - Salesforce exposes many of its services as Web Services
-

Hardly any company worth its salt does not have an API that it will expose to the developers. In fact, publishing an API means that "you got it made." It is a sign that there is enough developer interest in your product.

Jeff Bezos's Famous Memo @ Amazon

- Around year 2002, Amazon was growing at a rapid rate. Lots of systems were developed by multiple teams.
- Amazon CEO Jeff Bezos issues a mandate on how different products should interact with each other.
- Some say this transformed Amazon from a bookseller to the biggest 'Infrastructure as a Service' company.
- (See next slide for the full memo.)



Around 2002, the most common practice to share data between systems was to use a common datastore / database. Even though this facilitates access to up to date data, it makes both systems highly intertwined and hard to evolve at their own pace.

Exposing data as (web) service solves both problems:

- It allows access to the most up to date data.
- And, systems are loosely coupled and evolve at their own rate.

Jeff Bezos's Famous Memo @ Amazon (Cont.)

All teams will henceforth expose their data and functionality through service interfaces.

Teams must communicate with each other through these interfaces.

There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they use.

All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

Anyone who doesn't do this will be fired. Thank you; have a nice day!

This memo expressly forbids a repeat of the situation that had existed previously: applications either used private, non-standard protocols, or did not expose their data altogether. After Bezos sent this memo, only public, standard protocols were to be used, and all application data had to be exposed to be used by other departments.

References:

<https://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/>

<http://homepages.dcc.ufmg.br/~mtov/pmcc/modularization.pdf>

Interactivity: Poll Question

Question: In the early 2000s, what was the most common practice for sharing data between systems?

- A. Web services
 - B. A common datastore/database
 - C. Distributed systems
 - D. Docker
-



Evolution of Web Services

SOAP was created as a mechanism for implementing Web Services.

SOAP suffered from 'designed by committee' syndrome:

- It was heavy weight (depended on lots of libraries).
- It was clunky to use.
- It wasn't very well received by web/mobile developers.

Roy Fielding presented his paper "Architectural Styles and the Design of Network-based Software Architecture". (year 2000)[1]

He coined the term "REST," an architectural style for distributed systems.

REST has achieved wide spread adoption by developer community.

For example, Amazon Web Services (AWS) makes their APIs available in SOAP and REST. Their REST API is used 80% of time! [2]

SOAP was designed as an object-access protocol in 1998 by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein for Microsoft.

After SOAP was first introduced, it became the underlying layer of a more complex set of web services, based on Web Services Description Language (WSDL), XML schema and Universal Description Discovery and Integration (UDDI).

References:

- <https://en.wikipedia.org/wiki/SOAP>
- [1] Roy Fielding's original paper:
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- [1] https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- [2] https://aws.amazon.com/blogs/aws/rest_and_soap/

REST Success Story: AccuWeather on Azure

AccuWeather provides weather forecast data:

- 2.3 million locations covered.
- Two billion people around the world use AccuWeather one form or another.

AccuWeather's users include 180,000 websites, 200+ television station, 900+ radio stations and 600+ newspaper sites.

AccuWeather provides data through a variety of APIs:

- The API service is hosted by Microsoft Azure.

Azure API Service provides scalable, secure way of hosting AccuWeather APIs.



AccuWeather gathers hundreds of thousands of real-time weather observations from land, ships, aircraft, satellites, and radar. All this data (hundreds of Terabytes) is hosted at Azure blob store.

The raw data is processed and the results exported via APIs.

References

- <https://customers.microsoft.com/en-us/story/accuweather-partner-professional-services-azure>
- Azure App Service : <https://azure.microsoft.com/en-us/services/app-service/>

Success Stories of REST Web Services – Salesforce

- Salesforce is Cloud-based Customer Relation Management (CRM) service.
- It makes its CRM services available via APIs.
- Since it is all web based, there is no software to install at the customer site.
- This API approach enables Salesforce to integrate easily with third-party apps like Oracle, Google Docs and Amazon Web Services.



Enterprises use Customer Relation Management (CRM) software to keep track of their interaction with their customers. Some activities tracked by CRM include: finding new leads, following up with them, and eventually turning them into customers.

References

- https://www.slideshare.net/faberNovel/6-reasons-why-apis-are-reshaping-your-business/24-Case_Study_1_Salesforce_API

Success Stories of REST Web Services – Amazon Web Services (AWS)

Amazon Web Services (AWS) is a 'Infrastructure as a Service' (IaaS) offering from Amazon.

AWS is a hugely successful cloud platform:

- Nearly 50% market share
- Next competitor Microsoft Azure has about 10% market share
- Revenue of \$17 Billion in 2017

AWS has extensive API to access all their services

This has allowed a lot of applications in the [AWS Marketplace](#)

A few notables examples are:

- Cost Optimizers: Programs that schedule and run compute clusters in the most cost efficient manner on the AWS platform
- Database tools: Tools to provision and monitor databases
- Operational Monitoring: Services that send alerts when services go down



AWS was created to solve problems Amazon.com had with its internal systems. Once internal Amazon systems were developed with API as their focus (after Jeff Bezos's famous memo), they started focusing outward. AWS Compute Cluster was made available to the public around 2006. Since then AWS has steadily added more cloud services, like storage and databases to its platform.

References

- <https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/>

Success Stories of REST Web Services – Expedia

- Expedia is a travel booking company.
- It makes their content available via APIs to partners.
- The partner ecosystem has grown to 10,000 +.
- Now the majority of revenue (90%) comes from their API.



A user can search for travel deals directly at Expedia.com. The same content is available at partner sites – for example travel blogs, hotel review sites. This is achieved by Expedia API.

References

- https://www.slideshare.net/faberNovel/6-reasons-why-apis-are-reshaping-your-business/26-Case_Study_2_Expedia_marketing

Components of REST

Element	Description
Resource	Conceptual target of a hypertext reference, e.g., customer/order
Resource Identifier	A uniform resource locator (URL) or uniform resource name (URN) identifying a specific resource. e.g., http://myrest.com/customer/123
Resource Metadata	Information describing the resource, e.g., tag, author, source link
Representation	The resource content—JSON Message, HTML Document, JPEG Image
Representation Metadata	Information describing how to process the representation. e.g., media type, last-modified time
Control Data	Information describing how to optimize response processing. e.g., if-modified-since, cache-control-expir

REST consists of the following basic components:

- Who we are talking to, what we call it, and what additional information is needed to describe a request (that is, all “Resource” concepts)
- What we are sending (Representation and Representation Metadata)
- Additional instructions for processing the request (Control Data)

Interactivity: Poll Question

Question: Why should applications expose their data?

- A. To ensure that it is secure.
 - B. For the purposes of auditing the data.
 - C. To avoid any copyright issues.
 - D. It is an asset that should be available to others as a way to build the company's software together.
-



REST Basics

- REST doesn't need a new message format.
 - It uses HTTP protocol.
 - HTTP allows the following verbs:
CRUD = Create, Retrieve, Update, Delete
 - GET (Retrieve) = Give me some info
 - POST (Update) = Here is some update info
 - PUT (Create) = Here is some new info
 - DELETE = delete some info
-

REST has "verbs" like GET, POST< PUT, and DELETE. These verbs give REST its expressive power.

REST Webservice Examples

Get a list of orders.

<http://myapi.com/orders/>

Now, get details of one order.

<http://myapi.com/order/1>

The response could be a JSON document:

```
{ "id": 1,  
  "order_date" : "2018-01-01 14:04:01",  
  "amount" : "42.54" }
```

This code explains the main elements of REST.

1. Get the complete list of orders.
2. Select a specific order from the result of the previous query.
3. JSON is the language used by REST.

Interactivity: Poll Question

Question: What does the REST verb PUT mean?

- A. Give me some info
 - B. Here is some new info
 - C. Here is some updated info
 - D. Delete some info
-



REST Design Principles

1: Client-Server Architecture

2: Statelessness

3: Cache-ability

4: Layered System

5: Code on Demand

6: Uniform Interface

These are some basic REST design principles. We are going to look at these in detail over the next few slides.

1: Client Server Architecture

REST was modeled on 'web architecture' – which is Client-Server architecture.

This allows for '*Separation of Concerns*':

Both client and server have distinct roles to play.

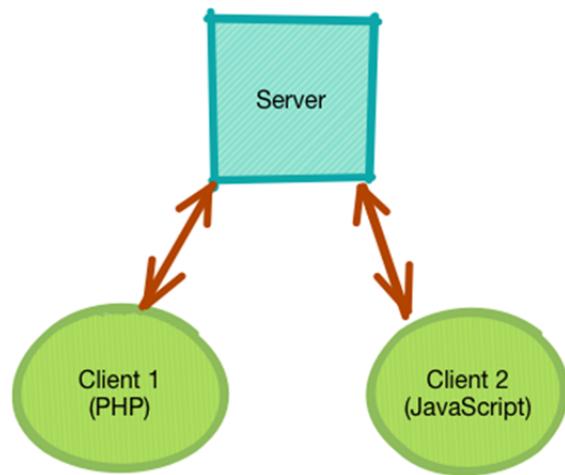
Server provides data.

Client can render the data (say a Browser displaying a webpage in a UI).

Both client and server can be implemented using different technologies and technologies:

For example, the server can be implemented using Python.

The client can be done using PHP or JavaScript.



We had Client Server applications before. But Web really popularized this architecture.

Web servers provide data to clients (usually browsers).

When a web server provides HTML, the browser renders it on user's computer/phone/tablet.

2: Statelessness

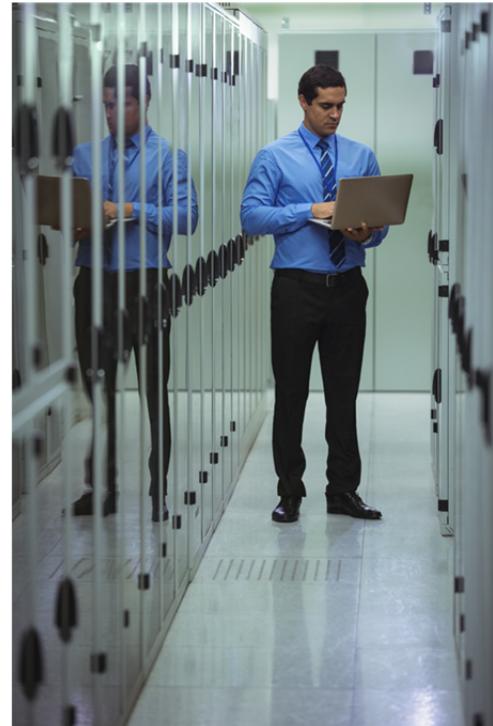
Server is not required to 'remember' client's state to serve request.

Clients must include all contextual information in each request to fulfill the request.

Client may store the state (session ID) and transmit it with the request.

This design allows the server to serve a large number of clients:

- Helps with scalability.



Web browsers store information in 'cookies'. Cookies were designed to be a reliable mechanism for websites to remember stateful information.

Examples are:

- User's login state
- Shopping cart id

https://en.wikipedia.org/wiki/HTTP_cookie

3: Cache-ability

When a web server sends out response, it specifies caching controls.

Stateless nature of web requests facilitates caching.

Caching greatly improves performance:

- Large assets (like images) can be cached for faster loading.

Caching also reduced the load on web servers and allows greater scalability.

Caching specifies how long a cached asset is valid:

- For example, a web page might be cached for up to an hour.
- But an image can be cached for a much longer period (few days).



A web page can have multiple assets.

- HTML page content
- CSS: for styling the page
- JavaScript scripts
- Images embedded in a page
- Etc.

Each asset typically will have a cache directive. Different assets may have different cache times.

When a browser is requesting a page, it may load certain items (images, css) from local cache on the user's computer. Caching helps reduce the load on web servers.

4: Layered System

In REST design, a network-based intermediary may intercept client-server communication for a specific purpose.

Some applications include:

- Load balancing
- Security
- Caching

Client cannot tell if whoever is intercepting is the server or intermediary.

This property allows us to deploy proxies or gateways.

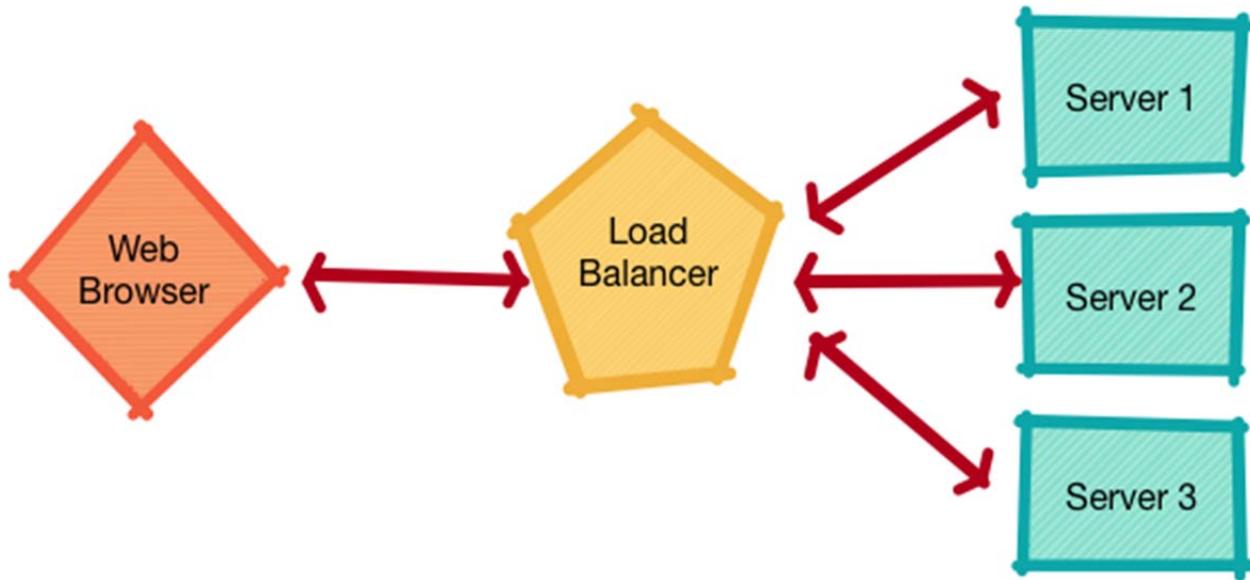
(See diagram in the next slide)



Proxies allow an organization to allow internet traffic in a controlled manner. Proxies can also cache data.

Gateways are designed to route traffic according to some rules.

4: Layered System (Cont.)



Here we see web page requests first land on the 'Load Balancer'. Load Balancer then sends the request to appropriate server. It considers the current load on the servers and will avoid heavily loaded servers. This allows the servers to share load evenly.

5: Code on Demand

The Web makes heavy use of code-on-demand.

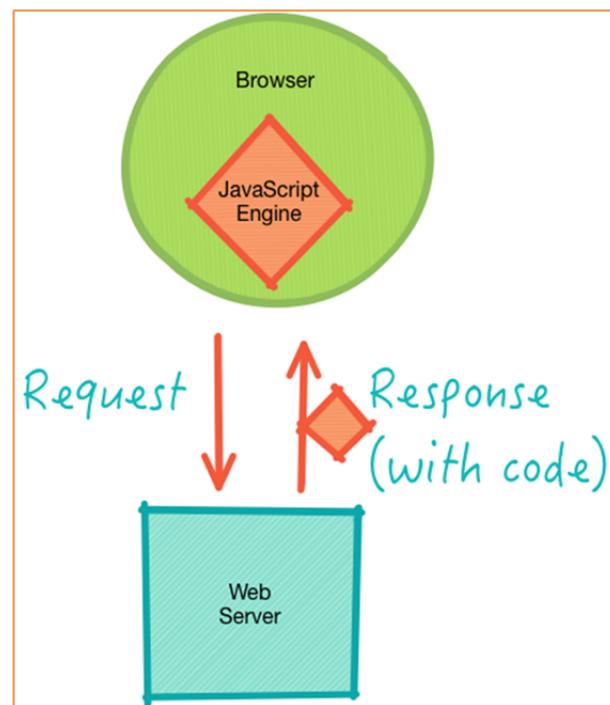
Servers can "push" executable code:

Code runs on the client.

Clients must be able to execute code.

In case of the Web servers:

- Servers push JavaScript to the browser.
- Browsers execute the code.



Executing client-side code reduces the load on the server. For example, JavaScript can do animations and rendering in the browser. Rendering graphs in a browser is a pretty good example of this.

For more information, refer to the D3 library (<https://d3js.org/>)

6: Uniform Interface

The interactions between the Web's components—meaning its clients, servers, and network-based intermediaries—depend on the uniformity of their interfaces.

Web components interoperate consistently within the uniform interface's four constraints, which Fielding identified as:

- Resources Identified in requests (URI)
- Resource Manipulated through Representation
- Self-Describing Messages
- Hypermedia as the engine of Application State (HATEOAS)



The uniformity of the interface is what gives REST the simplicity in design and implementation.

What is HATEOAS?

- HATEOAS = Hypermedia As The Engine Of Application State.
- Hypermedia is a document-centric approach with the added support for embedding links to other services.
- One of the uses of hypermedia and hyperlinks is composing complex sets of information from disparate sources.

```
<podcast id="111">
  <customer>http://customers.myintranet.com/customers/1</customers>
  <link>http://podcast.com/myfirstpodcast.mp3</link>
  <description> This is my first podcast </description>
</podcast>
```

This is a sample podcast definition. We see the 'customer' entry refers to customer data in one domain (myintranet.com). The link to 'podcast' points to another domain (podcast.com).

So this entry puts data together from disparate sources.

Discussion: Serialization Protocols

Questions:

- What are the most important factors when it comes to serialization? For example, speed, disk consumption, compatibility with outside tools, human readability?
- What are the considerations that lead you to make that decision? Are there standards and best practices within your organization?



Instructions: Enter your answers into Chat (All Participants).

Note that there is no right or wrong answer, but this exercise hopefully will help you think through this for your use case.

Discussion: Authentication Standards

Questions:

- How important is the use of standard API in building software?
- What are the major protocols or exposing data and functionality?
- How is exposing data different from exposing functionality?



Instructions: Enter your answers into Chat (All Participants).

Note that there is no right or wrong answer, but this exercise hopefully will help you think through this for your use case.

Lab: Implement a Protocol with Spring Boot

Overview:

Time: 15-30 Minutes

In this lab, you will:

- Implement Protocol with Spring Boot.



Instructions:

- Open the Student Lab Manual to Implement a Protocol with Spring Boot.
 - Refer to the lab files: api-lab-1
-

Please find the lab instructions in the file student lab manual. The accompanying lab files, which you will need to complete the lab, are also available in the api-lab-1 folder.

Lab: Open Source – REST Setup

Overview:

Time: 30-45 Minutes

In this lab, you will:

- Setup a development environment for RESTful calls.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Lab: Open Source – RESTful Time Tracker Web Service

Overview:

Time: 15-30 Minutes

In this lab, you will:

- Use your setup to create and deploy web services.



Instructions:

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Module Summary

Now that you have completed this module, you should be able to:

- Identify the elements of data communication.
 - Compare and contrast popular serialization formats.
 - Differentiate between protocols.
 - Understand API service frameworks.
 - Describe how open standards and RESTful web interfaces have transformed enterprise applications.
-

Now that you have completed this module, you should be able to:

- Identify the elements of data communication.
- Compare and contrast popular serialization formats.
- Differentiate between protocols.
- Understand API service frameworks.
- Describe how open standards and RESTful web interfaces have transformed enterprise applications.

This page intentionally left blank.

Open Source

Module 4

Module Objectives

After this module, participants will be able to:

- Understand the benefits of Open Source Software for Business and Enterprise.
 - Describe the ways in which various open-source licenses can affect usability.
 - Identify cost-savings opportunities through open source use.
-

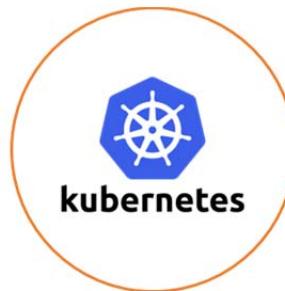
Open Source and Optum



Platform



OpenShift



kubernetes

There are three things that we will talk about regarding Open Source and how it is used at Optum. The first is what our Open Source platform is, which is available at this link:

<https://www.optumdeveloper.com/content/odv-optumdev/optum-developer/en/developer-centers/open-source-platform.html>

The second and third both involve the OSFI platform (Open Source Flexible Infrastructure) and how OpenShift and kubernetes are used on it. OSFI is Optum's next generation container hosting and cloud service solution. OSFI is built on open standards that promote commodity servers, commodity storage, and networking. OSFI also uses open source software, specifically: OpenShift Origin, kubernetes, and containerization technology.

For more on OpenShift, see the following link:

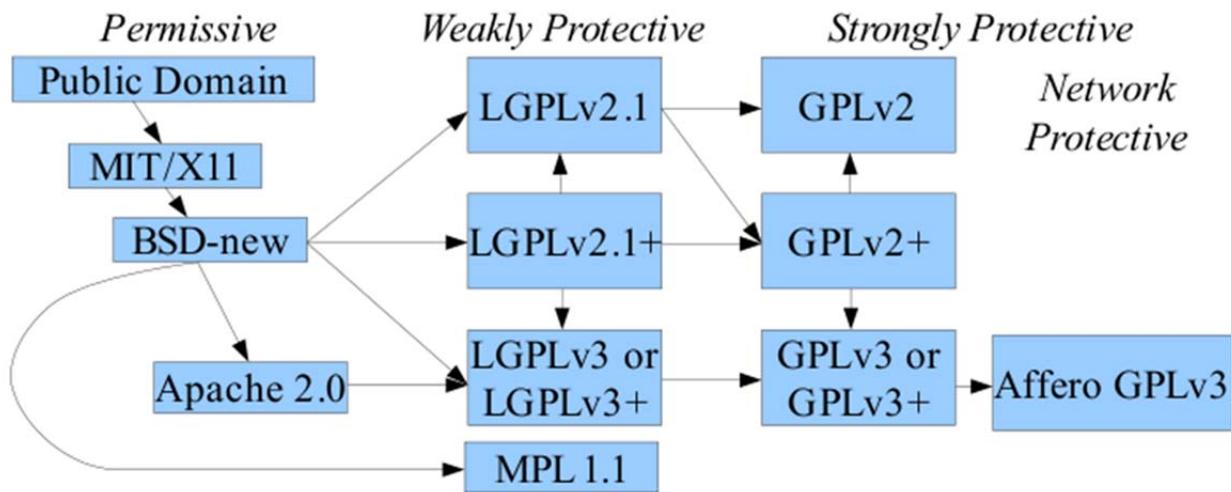
<https://www.optumdeveloper.com/content/odv-optumdev/optum-developer/en/developer-centers/hosting/hosting-openshift.html>

For more on kubernetes, see the following link:

<https://www.optumdeveloper.com/content/odv-optumdev/optum-developer/en/developer-centers/hosting/osfi-kubernetes.html>

Software Licenses

Open Source Licenses Overview



This is an overview of Open Source licenses. We will deep dive into each of these over the next few slides.

The world of Open Source is complex, but valuable for developers to get at least a basic familiarity with it.

Licenses, Free Licenses, and GNU GPL

Licenses	Free Licenses	GNU General Public License
<p>All Copyrighted software is owned by "someone"</p> <ul style="list-style-type: none"> • Creator • Company • Nonprofit <p>PD Software is not owned by anyone</p> <ul style="list-style-type: none"> • Have to "specifically" disclaim copyright • Otherwise copyright is implied, and license terms unknown <p>Copyrighted software must be licensed to the user</p> <ul style="list-style-type: none"> • Companies develop EULA (End User License Agreement) • Using software implies the user has accepted terms • Sometimes "Click-through" or "Shrinkwrap" acceptance 	<p>"Free Software" decided against PD status for GNU</p> <ul style="list-style-type: none"> • Retaining the copyright allows protection <p>"CopyLeft": Restricting distribution</p> <ul style="list-style-type: none"> • Means the end user cannot include code in end-product <p>Not Commercially Friendly?</p> <ul style="list-style-type: none"> • Companies can't use copylefted code in non-free products 	<p>GNU GPL is a "copyleft" license</p> <ul style="list-style-type: none"> • Means that any "derivative" work must also be GPL • "Viral" aspect of the license <p>GPL'ed code cannot be sublicensed</p> <ul style="list-style-type: none"> • GPL'ed code can only be used in a GPL'ed application <p>Steve Ballmer: "A Cancer", "A Virus", "Pac-Man Like" Considered Corporate Unfriendly</p> <ul style="list-style-type: none"> • Though most companies use some GPL software (Linux)

Licenses: There are companies that verify your use of free and open source software, like Black Duck, <https://www.blackducksoftware.com/>

Free licenses: We should mention that there are free code scanners and commercial companies, like Black Duck Software, that find out about potential licensing problems. This slide particularly illustrates the seriousness of this: you may have to use open source software that you have built on top of GNU software.

GNU General Public License: For further study, you can use the following guide:
<https://www.gnu.org/licenses/quick-guide-gplv3.en.html>

Interactivity: Matching Question

Question: Match each license type with its characteristic.

Instructions: Raise Hand to volunteer and then use the Line tool.



Free license

Companies cannot use copy-left code in non-free projects

GNU general public license

This type of code cannot be sub-licensed

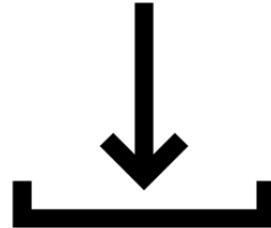
GPL Versions

GPLv2 vs GPLv3

GPLv3 has some additional requirements:

- Requires all owned patents required to run software be licensed
- Limits the Usage of DRM (Digital Rights Management)
- Allows more compatibility with other licenses (Apache)

Linux still licensed under GPL v2



The fight between GPLv2 and GPLv3 license was long and acrimonious. It happened because software patents came into the light. There are patent trolls today, and there are practicing entities. There are lawsuits (often in East Texas where the judges are famously inclined in favor of the plaintiffs), which can stop even Apple in its tracks. Thus, these questions are real and important.

GNU GPL

GPL requires "derivative works" to also be GPL.

But what about libraries?

Can you use a GPL'ed library in your code?

- Static linking is explicitly forbidden for GPL.
- Dynamic linking not entirely clear.

GNU LGPL (Lesser GPL) allows linking explicitly.

- Static and Dynamic linking.
 - "Copy and Paste" Code is not allowed.
-

LGPL is thus a combination of the open source ideals and business friendliness. However, many developers simply opt for Apache 2 license, which has all of the user friendliness and none of the GNU GPL baggage or associations.

Permissive Licenses

BSD Unix (Berkeley System Distribution)	MIT (Massachusetts Institute of Technology) License	Permissive Licenses allow "Copy and paste"
<ul style="list-style-type: none">BSD Unix was originally non-freeRelicensed under permissive license (BSD)Free but not "copylefted"Originally contained an "advertising clause" -- but later removed	<ul style="list-style-type: none">More popular today with OSS (Open Source Software) projects than BSDSimilarly permissive	<ul style="list-style-type: none">Companies can use code in proprietary productsExample: Apple uses BSD Linux (free) in Mac OS (Not free)May require attribution of code use in final productAllows companies to have free and non-free versions

Permissive libraries are much more popular with businesses, because they allow one to use the code and not contribute anything in return.

Apache Licenses

Apache Web Server (Still the most popular!)	Apache Software Foundation	Allows "SubLicensing"
<ul style="list-style-type: none">Commercial friendly software license"Apache License" - Permissive License	<ul style="list-style-type: none">Nonprofit which holds copyrights to many popular productsExamples: Hadoop, Spark, Cassandra, etc.Corporate Friendly	<ul style="list-style-type: none">Means Apache code can be used in GPL'ed applicationOr in a commercial applicationBut the original code is still Apache licensedRequires Attribution

A great overview of all licenses and their practical significance can be found here:

<https://choosealicense.com/>

Dual Licensing

- Companies often have the same code with two licenses.
 - The free version is free to use in personal, non-commercial, academic purposes.
 - Commercial users will purchase licenses.
 - Community will develop the Open Source version.
 - This allows for “freemium” models.
-

For example, some database vendors will have a free version for personal use or evaluation purposes.

Once the software is to be used by an organization for commercial purpose, they would purchase the license.

For the software company, one of the benefits of the dual licensing model is plenty of users will download and practice the software. And when it comes time to choose software for commercial purpose, most users tend to choose the one they are already familiar with.

License Summary

	Public Domain	Permissive	CopyLeft	Freeware	Proprietary
Copyright	No	Yes	Yes	Yes	Yes
Right to Use	Yes	Yes	Yes	Yes	Yes
Right to Copy	Yes	Yes	Yes	Yes	No
Right to Modify	Yes	Yes	Yes	No	No
Right to Distribute	Yes	Yes, under same license	Yes, under same license	Yes	No
Right to Sublicense	Yes	Yes	No	No	No
Example	SQLite	Apache Web Server	Linux	Flash Plugin	Windows

Developers are not necessarily expected to memorize this, but the chart is included here for your reference. Developers are expected to know that there are different types of licenses, and that they have some restrictions in using OSS in their development efforts. Make sure to consult the legal department if you have any specific questions about whether an application or service can be used freely.

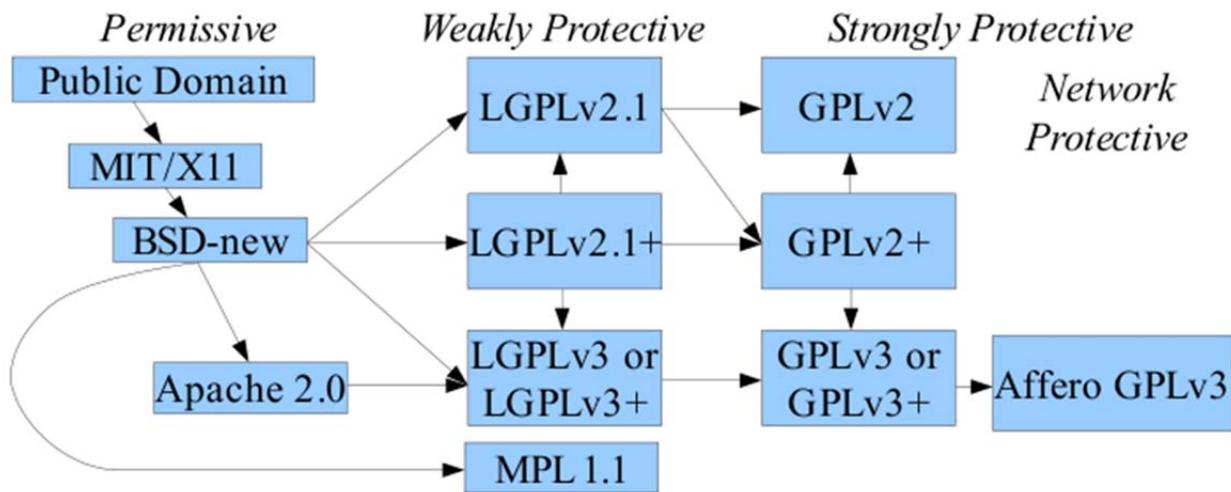
Interactivity: Poll Question

Question: What is the general procedure for analyzing licenses?

- A. Determine who created them.
 - B. Perform click-through acceptance.
 - C. Make sure they are weakly protective.
 - D. Consider the effect on your code.
-



License Summary (Cont.)



This is the final summary on licenses.

Cost Savings in Open Source



There are often very real cost savings opportunities that come in switching to Open Source software. Consider the example of switching from Oracle to MySQL. Such a switch has resulted in the past in 44% savings.

When investigating cost savings opportunities, remember that savings can come with either open source software (i.e., database) solutions, or even hardware solutions. Specifically, we see differences in commoditized open compute hardware vs. vendor-specific hardware.

Discussion: Open Source Software Licenses

Questions:

- Describe the major software licenses used with Open Source.
- Which licenses are considered business friendly?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Discussion: Open Source Software License Considerations

Questions:

- How does the software license of the open source affect your software development?
- What is the difference between distributing your software and running it as a service in your cloud, as far as licensing is concerned?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Module Summary

Now that you have completed this module, you should be able to:

- Understand the benefits of Open Source Software for Business and Enterprise.
 - Describe the ways in which various open-source licenses can affect usability.
 - Identify cost-savings opportunities through open source use.
-

Now that you have completed this module, you should be able to:

- Understand the benefits of Open Source Software for Business and Enterprise.
- Describe the ways in which various open-source licenses can affect usability.
- Identify cost-savings opportunities through open source use.

Reliability, Resiliency, and High Availability

Module 5

Module Objectives

After this module, participants will be able to:

- Understand industry terminologies to describe highly available applications.
 - Describe best practices of Application Resiliency.
 - Describe re-engineering patterns.
-

Building Resilient Applications

Resiliency

Resiliency: Recover, Self-Heal.

For example, if an app depends on a database, what would happen if the database goes down?

We would like to build apps that are resilient:

- That can withstand external service failures.
- And gracefully deal with failures.

Think about apps you use on a daily basis that are resilient:

- Facebook, Twitter, etc.



A resilient application not only handles bad input from a user but can also withstand the external systems it depends on.

For example, say an application is saving data to a database. What can the app do if the external database is not available? It should gracefully handle it without crashing. This is the concept of building reliable apps on unreliable components.

High Availability

High Availability: Refers to a system that is durable and operates exceeding an expected performance.

Usually, this means uptime.

- Expressed in percentage of uptime per year.
- These are measured in 9s.

Nines

- A system that stays up 99% of the time is called 2 nines.
This system can be down 14 minutes a day (24 hrs).
 - A 4 nine system (99.99%) can be down 8.6 seconds a day (24 hrs).
 - See next slide for more details.
-

High availability can refer to software and/or hardware. Advances in manufacturing processes have resulted in electronics components that last a lot longer. Modern software practices now make it easier to develop applications that are highly available.

- https://en.wikipedia.org/wiki/High_availability

Measuring Uptime in Nines

Nines	Availability %	Downtime per day	Downtime per month	Downtime per year
One nine	90%	2.4 hrs	72 hrs	36.5 days
Two nines	99%	14.4 mins	7.2 hrs	3.65 days
Three nines	99.9%	1.44 mins	43.8 mins	8.76 hrs
Four nines	99.99 %	8.66 secs	4.38 mins	52.56 mins
-Telco switches - Storage	Five nines	99.999%	864.3 ms	25.9 secs
High end flash arrays (HP 3PAR)	Six nines	99.9999%	86.4 ms	2.59 secs
	Seven nines	99.99999%	8.64 ms	262.97 ms
	Eight nines	99.999999%	0.864 ms	26.297 ms
				315.569 ms

Telco switches achieving 5 nines availability are only down for 5 minutes per year. This is pretty good. Note that during some system outages, there's nothing wrong with the data, it just isn't accessible.

One should distinguish between the guarantee of not losing the data and the guarantee of being able to access the data when you want. For example, in one Amazon outage, the data was there (they have nine nines reliability), but it was not available on request (availability does not have nine nines). Four thousand large sites (like Dropbox) were down for a while because of that.

The availabilities are usually defined in the Service Level Agreement (SLA).

High Availability

Let's say we have a database.

Data stored in the database is safe.

But if the database is down, we can't access the data!

- Not highly available.

Example: Bank ATM

- If the ATM is down, you can't access your account.
- Even though your money is safe in the account.



If a bank only had ONE ATM, then it is not very highly available.

So, the bank having many ATMs would be considered highly available.

Interactivity: Matching Question

Question: Match each characteristic with its description.

Instructions: Raise Hand to volunteer and then use the Line tool.



Resiliency

Using multiple components, some as standby

Reliability

Planning for various failure scenarios

Scalability

“Prepare for failure then nothing fails”

Redundancy

Redundancy is basically 'more than one'.

Examples:

- Two links between data centers.
- Two power supplies in a computer.

Redundancy allows 'service continuation'.

- If one power supply fails on a computer, the other one will keep providing power.
- If one link fails, traffic would be re-routed through the other link.



In this scenario, we have two data centers. They are connected by two links.

The goal is to have a backup system if one link goes down.

Can you think of ways when a link can be disrupted?

Someone digging a trench could dig into a cable.

These two links will be physically separate (following two separate paths).

And most likely these links will be serviced by two different data providers – this minimizes potential downtime.

Examples of Redundancy

1 + 1 Redundancy

- When one component fails, the other one takes over.
- Could be active / active setup.
- Switch over is automatic – usually.
- Example: active-active power supplies for a server.

N = Redundancy

- If only one active component is needed for operation.
- Tolerates (N-1) failures.
- For example, if we have 3 copies of the database, it can tolerate up to 2 (3-1) databases failing.

In 'active-active' systems both components are working. A classic example is two power supplies in a computer.' They will both supply power to the computer.

When one fails, the other power supply will provide all needed power.

In 'active-passive' setup, only one is active. When the active one fails, the passive one will take over.

There is usually a lag time before passive becomes active.

An example would be a database with a backup.

- https://en.wikipedia.org/wiki/N%2B1_redundancy

Discussion: Definitions and Examples

Questions:

- Give examples for:
 - System reliability
 - Scalability
 - Resiliency
- How do they differ?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Discussion: Methods

Questions:

- What are the usual methods of achieving reliability, scalability, resiliency?
- What is the idea of building reliable systems out of unreliable components?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Discussion: Analyzing an Application for Resiliency

Scenario:

- We have a web form: **Contact US**.
- Contact US collects some information and sends that information, in an email, to the company.



Instructions: Enter your answers into Chat (All Participants).



What are the failure scenarios you can think of?

Here are some things to take into consideration:

- What if the remote mail server connection is refused?
- What if the web server fails?
- What if the web server disk becomes full?

Module Summary

Now that you have completed this module, you should be able to:

- Understand industry terminologies to describe highly available applications.
 - Describe best practices of Application Resiliency.
 - Describe re-engineering patterns.
-

Now that you have completed this module, you should be able to:

- Understand industry terminologies to describe highly available applications.
- Describe best practices of Application Resiliency.
- Describe re-engineering patterns.

This page intentionally left blank.

Cloud Development Framework

Module 6

Module Objectives

After completing this module, you should be able to:

- Describe the basics of Microservices
 - Describe Spring Cloud basics
 - Explain Service Development with Spring Cloud
 - Describe Spring Boot and Spring Framework
 - Describe Cloud Application Scaffolding
 - Identify Cloud Portability Patterns
-

Microservices Introduction

Definition

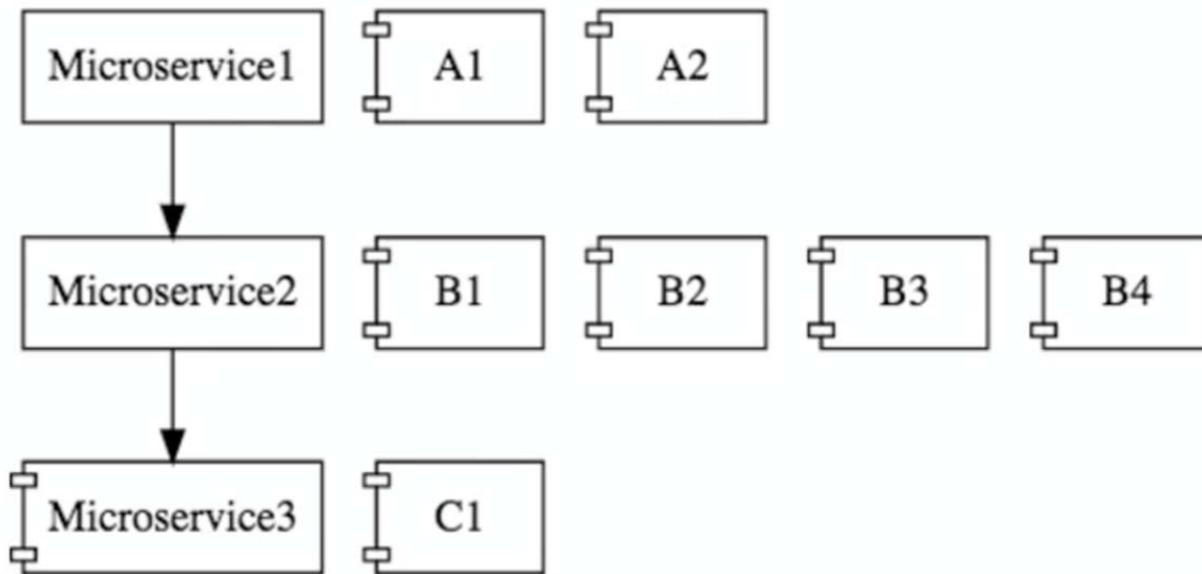
“Small autonomous services that work together.”

- Sam Newman

Microservices architectural style structures an application as a collection of loosely coupled services. It advocates modularity with parallel development of each service.

Components of Microservices

1. REST
2. Small well-chosen deployable units
3. Cloud enabled



In the above diagram, A1 and A2 are instances of microservice 1. Similarly, B1, B2, B3, and B4 are instances of microservice 2, c1 is an instance of microservice 3.

These services are deployable, all/any of them can scale independently.

Being cloud-enabled, we can simply bring up "more of the same" instance. This is much better than starting and configuring a new server from scratch.

Challenges of Cloud Enabling Microservices

1

Bounded Context: Deciding boundaries for each microservice, in previous slide. Microservice1 and 2 are interacting, but the next upcoming service may need to exchange data with service 2 again. Problematic to decide upfront, but doable gradually.

2

Configuration: Managing services configuration.

3

Dynamic: Scale up/down - Keep the load on active instances.

4

Visibility: Need understanding of the complete system and interactions to track down a bug.

5

Logging: Different services will have their own logging mechanism, resulting in GBs of distributed unstructured data.

6

Root Cause Analysis: RCA of any issue becomes difficult as logic spans across multiple machines.

There are more challenges in the cloud infrastructure, here is a list of some of them.

Communications between services: this is usually done with HTTP, if not, it will be an Enterprise Service Bus (ESB), or another protocol.

Health monitoring: you need a separate application, and maybe in a new language, like Nagios, to monitor health. Don't forget alarms and the like.

Logs will be created on multiple servers; you need a service to collect them together.

You may have to deal with multi-phase commit between different services.

If service A calls service B, and service B calls service A, you may have a cyclic dependency problem.

Finding root cause in distributed logging may not be easy.

Advantages

1. Freedom to choose a technology stack, microservice 1 may be written in Java and 2 maybe written in Node etc.
2. Dynamic scaling.
3. Faster release cycles.
4. Enables continuous delivery.
5. Fault isolation, even if one service fails, others continue.
6. Code is organized around business capabilities.
7. Works well with containers like Docker.
8. Complement cloud activities.
9. Follow the single responsibility principle.
10. Easy to change and test.



Evolutionary Design – No need to rewrite your whole application. Add new features like Microservices, and plug them into your existing application.

Small Codebase – Microservice deals in solving one smaller component or module of a business problem. Thus, it makes it easier to maintain a compact codebase

Auto Scale – Ability to load and scale the required service which can handle bigger loads.

Easy to Deploy – Modularity of microservices permits us to redeploy on the required codebase. Therefore, it is not required to redeploy the entire application.

System Resilience – If some of the services go down, only some features will be affected, not the entire application.

Interactivity: True or False?

Question: True or False. One of the advantages of microservices is that they are easy to deploy.

Instructions: Use ✓ (True) or X (False)



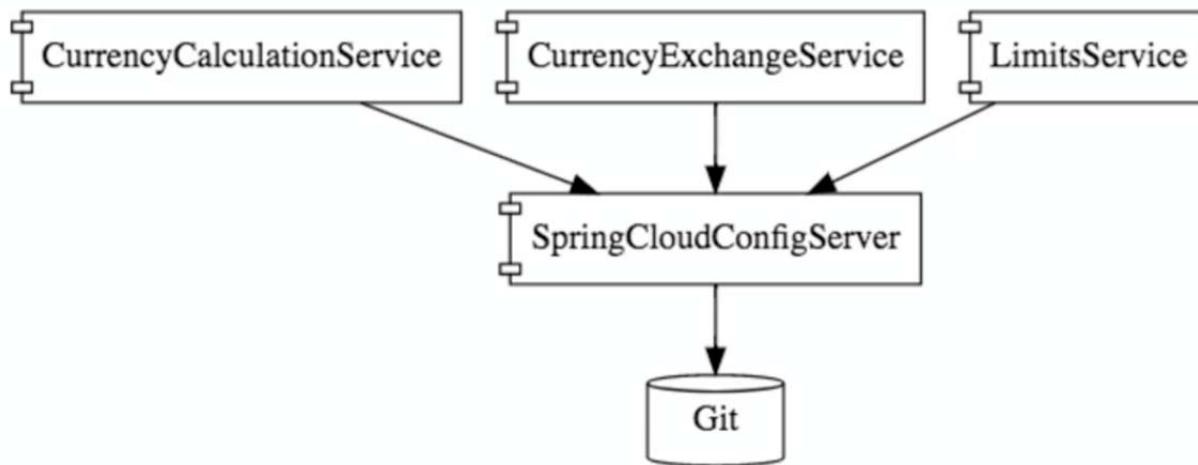
Spring Cloud Introduction

Introduction

Spring Cloud is not just one project, it has many projects under its umbrella. It provides tools to quickly build some common patterns in distributed systems.

How Spring Cloud solves microservices problems?

Configuration - Spring Cloud Config Server provides a central place to put configurations for all microservices. We will be connecting our rest service with config server in labs.



Some of the common patterns in distributed systems (e.g., configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state) have readily available tools in Spring Cloud for developers.

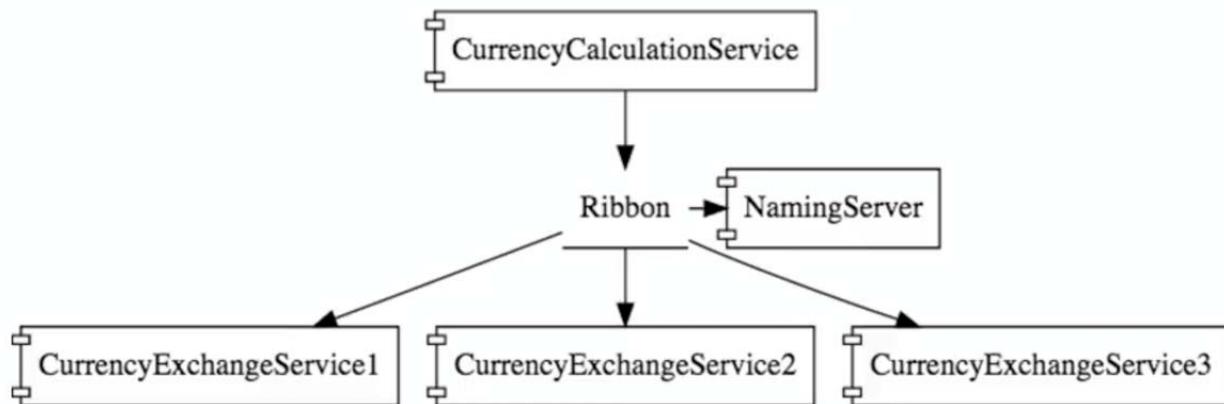
Spring cloud developers can quickly set up services and applications to implement boilerplate patterns that come from coordination of distributed systems.

These will work well in both local as well managed/cloud distributed environment.

Scaling

Dynamic Scale up/down - Addressed by below components:

- Naming Server (Eureka)
- Ribbon (Client Side Load Balancing)
- Feign (Easier REST Clients)



Components that form a part of the scale systems could be used while designing self-healing systems. Above is just an example, there are an array of options to choose for each component in microservices.

Components Detail

- In the previous slides example, currency calculation service needs an instance of the currency exchange service but does not know which service to call. Each currency exchange service will register itself with Eureka (naming server) and when needed currency calculation service will ask Eureka for instances of currency exchange service. Hence has two roles, service registry and service discovery.
- Ribbon is hosted by the currency calculation service, and after service discovery, it is used to balance the load on the currency exchange service.

Visibility - Addressed by:

- Zipkin Distributed Tracing
- Netflix API Gateway - common features like logging, security.
- Zipkin can trace a single request through multiple services.

Fault Tolerance - Managed by Hystrix, helps configure default response for services which are down.



OpenZipkin, full-fledged open-source version of Zipkin – a project that originated at Twitter in 2010 - is based on Google Dapper paper.

Zipkin provides REST API to which clients talk to directly.

Netflix's Hystrix library provides an implementation of the Circuit Breaker pattern – that is Hystrix watches for calls that fail, and if the number of failures cross a threshold, then Hystrix opens the circuit leading to automatic failure of all subsequent calls. While the circuit is open, calls are redirected from Hystrix to the method, and are passed on to the fallback method specified.

Service Development with Spring Cloud

Ports to be Used

Application	Port
Limits Service	8080.8081,...
Spring CloudConfig Server	888



We will see a use case for limit service which needs to read config values minimum and maximum.

Limit service is elastic in nature and can run on any of the ports like 8080, 8081, 8082 ...

There will be a central server that houses all the configurations and runs on 8888 and provides configuration when queried.

URLs to be Used

Application	URL
Limits Service	http://localhost:8080/limits
Spring CloudConfig Server	http://localhost:8888/limits-service/default



Mentioned here are URLs to refer to after deployment.

If there are multiple instances deployed of limit service, only the port needs to be changes for each deployed service.

Spring Cloud Config Server

1. Central configuration for all services.
2. It requires git repository as a central DB to store configuration and read it.



This service will read config from the git repository and keeps it ready for any services which may require it.

Local git is also fine while in production, a checked in git repo is recommended.

Interactivity: Poll Question

Question: What type of git repo is recommended in Spring Cloud?

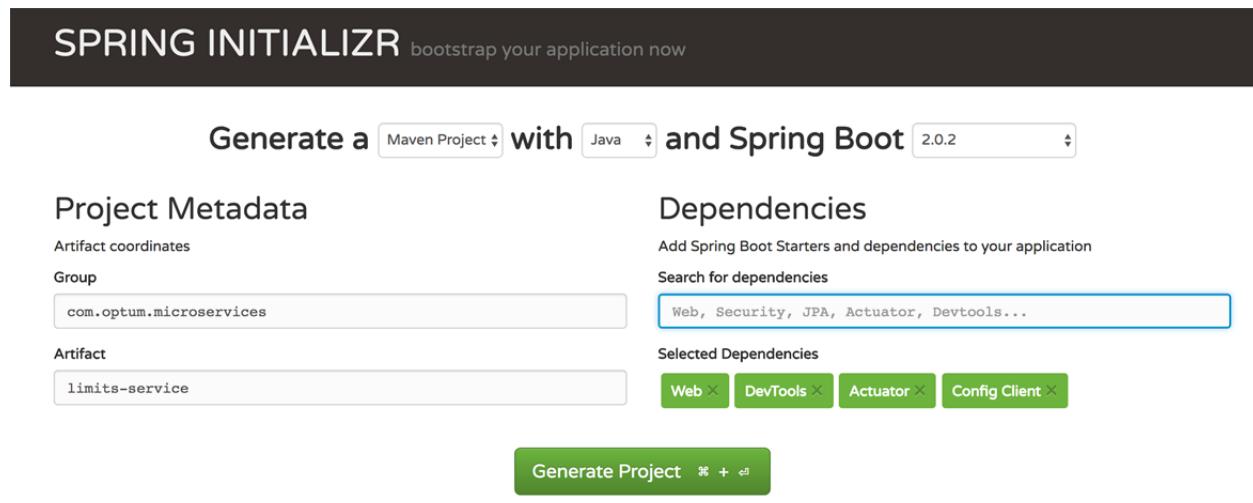
- A. Check in git repo
 - B. Stateless repo
 - C. Service-limiting repo
 - D. External repo
-



Bootstrapping Simple Web Service

Use either IDE support or Spring Initializr web project hosted at <https://start.spring.io>

Let's generate a service called limits service. All configurations are shown below.



The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, a search bar says "Generate a [Maven Project] with [Java] and Spring Boot [2.0.2]". The "Project Metadata" section has "Artifact coordinates" fields for "Group" (com.optum.microservices) and "Artifact" (limits-service). The "Dependencies" section has a search bar "Add Spring Boot Starters and dependencies to your application" with "Search for dependencies" (Web, Security, JPA, Actuator, Devtools...). A "Selected Dependencies" section shows four green buttons: Web, DevTools, Actuator, and Config Client. At the bottom is a green "Generate Project" button.

After clicking **Generate Project**, a zip file will be downloaded, unzip the file and keep it in a folder of your choosing.

Spring Initializr is very simple to use. We just need to enter our project details and required dependencies. The tool then generates and downloads a zip file that contains a standard spring boot project based on the details that have been entered by us.

Prototype that has been generated will have all the project directories, the main application class and the pom.xml or build.gradle file with all the dependencies. We need to unzip the file and import it in an IDE to start working.

The downloaded zip file is provided in the resources folder.

1. Running Generated Web Service

1

Open this project in your favorite IDE. First time setup takes time for Spring Boot.

2

Run com.optum.microservices.limitsservice.LimitsServiceApplication to assure service is working fine. Once run you should see the below statement.

Almost every IDE has Spring Boot support.

Instead of Initializr we could have used IDE.

2. Developing REST Limits Service and Deploying on 8080

1

Create a
classcom.optum.microservices.limitsservice.LimitsConfigurationController

2

Create a public method retrieveLimitsFromConfiguration and annotate this
method for GET mapping /limits, also class as Rest Controller.

```
1 @RestController
2 public class LimitsConfigurationController {
3
4     @GetMapping("/limits")
5     public LimitConfiguration retrieveLimitsFromConfiguration() {
6         return new LimitConfiguration(1000, 1);
7     }
8 }
```

1. `@RestController` is a specialized version of the controller and it includes the `@Controller` and `@ResponseBody` annotations. As a result, `@RestController` simplifies the controller implementation.
2. `@GetMapping` defines the access path for this method and supports HTTP GET method.

3. Setup Git Local Repo with Config File

1

Create a folder anywhere on your machine and perform git init.

2

Add this folder as an external resource, I have created it inside the resources folder.

3

Create a properties file inside this folder, this is the configuration server, however, file name has to be same as the application name for which we are creating the configuration, i.e., limits-service.properties.

4

Move the below entries from application.properties of limits-service to the new config project. Then for the limits-service you will get the configuration from spring cloud config.

5

Run git init inside the new folder and add the newly created file.

6

Commit this file locally, no need to push.

1

limits-service.minimum=1

2

limits-service.maximum=999

Create a new Git repository using git init.

It can also be used to convert an existing project to a Git repository or can be used to initialize a new repository with no contents.

We just need a folder with git initialization.

Its ok to have a checked in git repo.

4. Generate and Setup Spring Cloud Config Server on Port 8888

1

Create a new project from <https://start.spring.io> and select **Config Server** as a dependency as shown below.

2

Generate Project and open with IDE. In the diagram below, the downloaded project is placed in the resources folder.

Generate a with and Spring Boot

Project Metadata

Artifact coordinates

Group

`com.optum.microservices`

Artifact

`spring-cloud-config-server`

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

`Web, Security, JPA, Actuator, Devtools...`

Selected Dependencies

`DevTools` `Config Server`

Don't know what to look for? Want more options? [Switch to the full version.](#)

Spring Initializr is a web application used to generate Spring Boot applications. Currently Maven and Gradle are supported as build tools. It also manages compatible versions for every dependency.

5. Spring Cloud Config Setup

1

Make the below entries to application.properties.

1

```
spring.application.name=spring-cloud-config-server
server.port=8888
spring.cloud.config.server.git.uri=file:///{git-localconfig-repo-path}
```

2

Change git-localconfig-repo-path with the full path on your machine.

3

Add **@EnableConfigServer** to
com.optum.microservices.springcloudconfigserver.SpringCloudConfigServerApplication

4

Run **spring cloud config server** and click URL <http://localhost:8888/limits-service/default>

```
{"name": "limits-service", "profiles": ["default"], "label": null,
"version": "d7bbc8de7b6d61e4da2f8f20dcfea8b24954dc34", "state": null,
"propertySources": [ { "name": "file://resources/git-localconfig-repo/limits-service.properties", "source": { "limits-service.minimum": "8", "limits-service.maximum": "8888" } } ]}
```

All configurations for the boot application go into application.properties.

Spring Cloud Config Server reads all configurations centrally.

File URI in application.properties should be changed to checked in git repo for the production/test environment.

6. Changes to Limits-service to Read from Spring Cloud Config Server

1

Rename application.properties inside limits-service to **bootstrap.properties**.

2

Add **spring.cloud.config.uri=http://localhost:8888** to bootstrap.properties.

3

Run the application, and now limit-service will read the property from spring-cloud-config-server running on port 8888.

4

We now have a microservice running and talking to spring cloud through the rest URL.

5

Final contents of both services are placed under the resources folder. The minimum change required is the git file path.

6

Similarly, all services discussed before when developed will orchestrate to form a fault-tolerant and scalable system.

bootstrap.properties is loaded before application.properties.

bootstrap.properties is needed only for the spring cloud project where application's properties are stored on a remote server.

Spring cloud application operates by creating a bootstrap context which is parent context for the main application.

It also contains encryption/decryption information.

Lab: Limits Service

Overview:

Time: 30 Minutes

In this lab, you will:

- Develop limits service to read configuration from a properties file.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Spring Boot and Spring Framework

Spring Framework

Spring framework characteristics:

- Dependency injection framework
 - Allows you to build applications from “plain old Java objects” (POJOs) and to non-invasively apply enterprise services to POJOs
 - Glues things together
 - Provides flexibility of choices; helps in maintaining the architecture of the application
 - Features are organized into ~20 modules
 - Created as a response to the complexity of the early JEE specifications
-
-

Spring framework was developed to take care of infrastructure so that developers can focus on the application:

- Spring framework is a dependency injection framework, which manages the life-cycle of Java components (beans).
- Spring allows you to build applications from “plain old Java objects” (POJOs) and also to non-invasively apply enterprise services to POJOs. This capability also applies to the Java SE programming model and to full and partial Java EE.
- It is a framework that glues things together, a middleman to MVC frameworks (Struts 1, 2, JSF, etc.), ORM frameworks (Hibernate, iBatis, JOOQ etc.) and other necessary facilities (Quartz, Email, whatever we need, most likely, there's Spring support).
- Springs' framework nature is to provide flexibility of choices and helps in maintaining the architecture of the application.
- Spring Framework's features are organized into ~20 modules. These modules are broadly grouped into 7 buckets viz., Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the next slide.
- Spring came into being in 2003 as a response to the complexity of the early JEE specifications. Spring and Java EE are complementary to each other and not compete with each other.

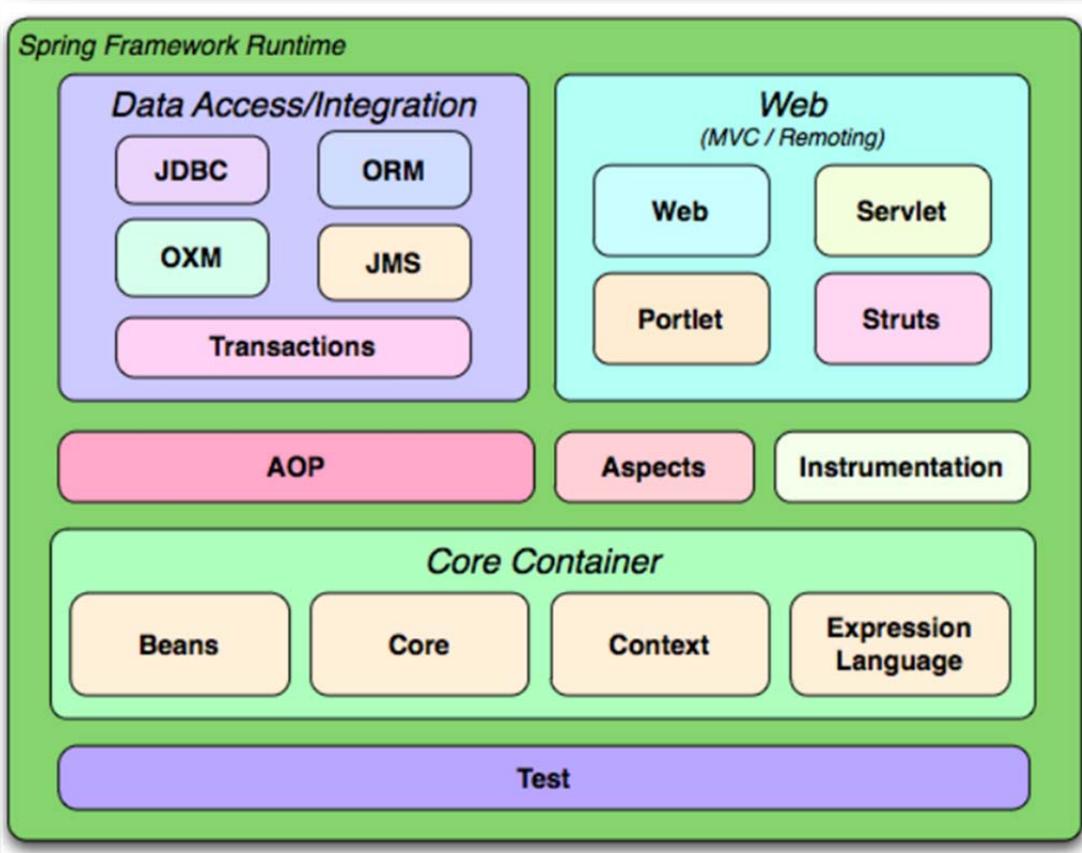
Examples of how an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.

- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

From the official Spring documentation (<https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/overview.html>).

Spring Framework (Cont.)



This diagram illustrates the major components of the Spring framework, Data Access and Web. They, in turn, rely on the internal services, AOP, Aspects, and Instrumentation. The Core Container services include Beans, Core, Context, and Expression Language.

Spring Boot

Spring Boot characteristics:

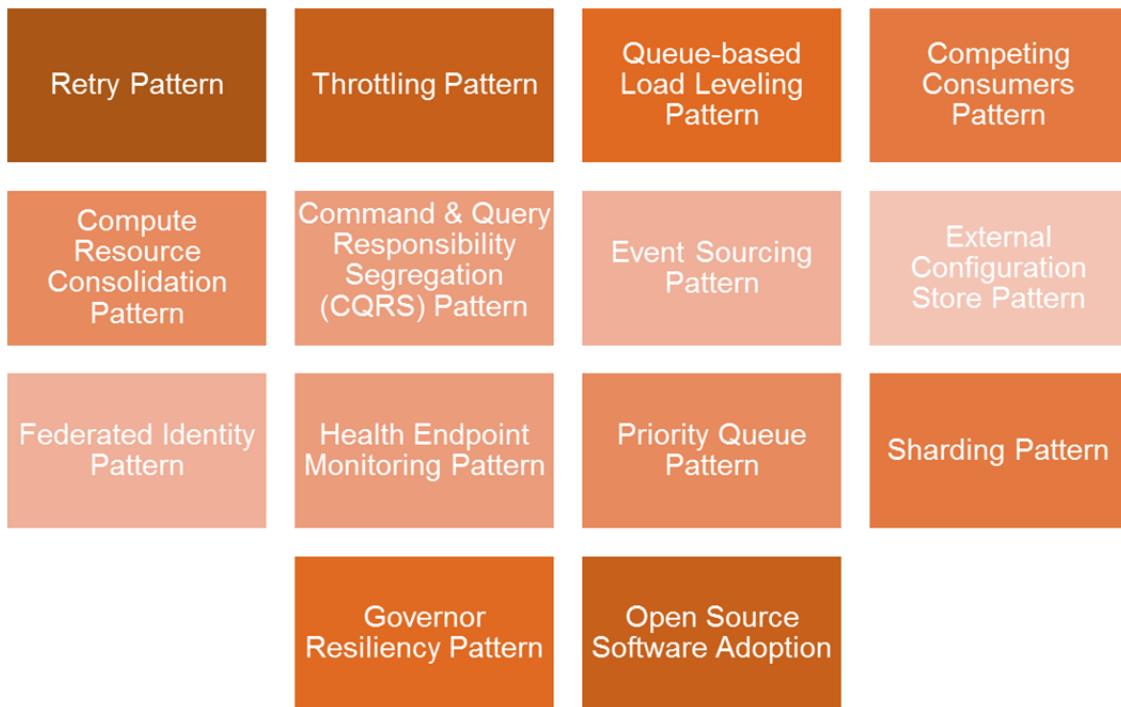
- Pre-configured, pre-sugared set of frameworks/technologies to reduce boilerplate configuration
- Takes less than 20 LOC to have a simple RESTful application up and running
- Many ways to configure applications to match your needs
- Most Spring Boot applications need very little Spring configuration
- Tomcat, Jetty or Undertow are embedded directly
- Provides production-ready features
- No xml configuration required

Spring Boot characteristics include:

- Spring boot is basically a suite, pre-configured, pre-sugared set of frameworks/technologies to reduce boilerplate configuration providing the shortest way to have a Spring web application up and running with minimal code/configuration out-of-the-box.
- It takes less than 20 LOC to have a simple RESTful application up and running with almost zero configuration.
- It definitely has a ton of ways to configure applications to match your needs.
- Most Spring Boot applications need very little Spring configuration.
- Tomcat, Jetty or Undertow are embedded directly (hence, not required to be deployed externally).
- It provides production-ready features like metrics and health checks.
- No xml configuration is required.

The official Spring Boot documentation is found here, <https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html>, please refer to this for more details.

Cloud SDKs



The capabilities of a Cloud platform won't benefit applications unless they are intentionally used. That's where design patterns come in. Design patterns help applications use the capabilities of the Cloud platform. Each pattern consists of a context and problem along with the solution, issues and considerations for incorporating the pattern into the application design. Cloud design patterns are associated with one or more of the following eight categories: availability, data management, design and implementation, messaging, management and monitoring, performance and scalability, resiliency and security.

Interactivity: Poll Question

Question: Which of the following is a characteristic of Spring Boot?

- A. XML configuration is required.
 - B. It takes more than 20 LOC to have a simple RESTful application up and running.
 - C. Most Spring Boot applications need very little Spring configuration.
 - D. Tomcat needs to be deployed externally.
-



Cloud Application Scaffolding

Ways to Scaffold Cloud Application

There are broadly two ways we can scaffold a cloud application.

1. Spring Initializr Project
2. Spring Boot CLI



-
- Spring Boot CLI also uses Spring Initializr behind the scenes.
 - There is a third way often discussed through the use of an IDE, but behind the scenes one of these two ways are in play.
 - Highly useful to get started with minimum effort and create stand-alone, production-grade applications.
 - Scaffolding improves developer productivity by reducing setup time.

Spring Initializr

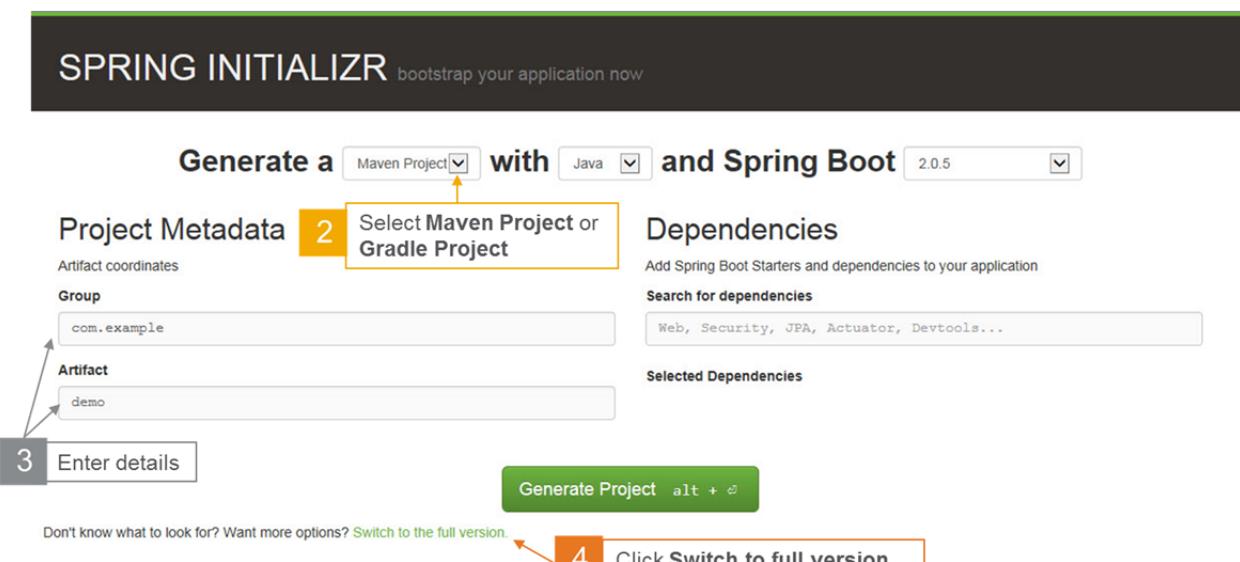
1. Spring Initializr is a web tool which helps in quickly bootstrapping Spring Boot projects.
2. It is provided by Spring on the official site.
3. Quickly generates specified components with matching module versions, source, test, and resources directories.
4. Generated code is downloaded as a zip file.
5. Maven and Gradle are supported build tools.



-
- The tool is very simple to use. We just need to enter our project details and required dependencies. The tool automatically generates and downloads a zip file with a standard Spring Boot project based on the details we have provided.
 - Generated prototype will contain all the project directories, the main application class and the pom.xml or build.gradle file with all the dependencies. The file can be unzipped and imported to an IDE of your choice and can start working.

Steps to Scaffold Boot Application with Spring Initializr

- 1 Go to <http://start.spring.io>.



Spring Initializr is a web application to generate Spring Boot applications. Currently on Maven and Gradle and is supported as build tools. It also manages compatible versions for every dependency.

The steps to Scaffold Boot Application using Spring Initializr are as followed:

- 1 Go to <http://start.spring.io>.
- 2 Select **Maven Project or Gradle Project** from the drop-down at the top. Maven Project is the default.
- 3 Enter group and artifact details.
- 4 Click **Switch to full version** for more options.

Steps to Scaffold Boot Application with Spring Initializr (Cont.)

Generate a Maven Project Java and Spring Boot 2.0.5

Project Metadata

Artifact coordinates
Group: com.example
Artifact: demo
Name: demo
Description: Demo project for Spring Boot
Package Name: com.example.demo
Packaging: Jar
Java Version: 8

Too many options? [Switch back to the simple version.](#)

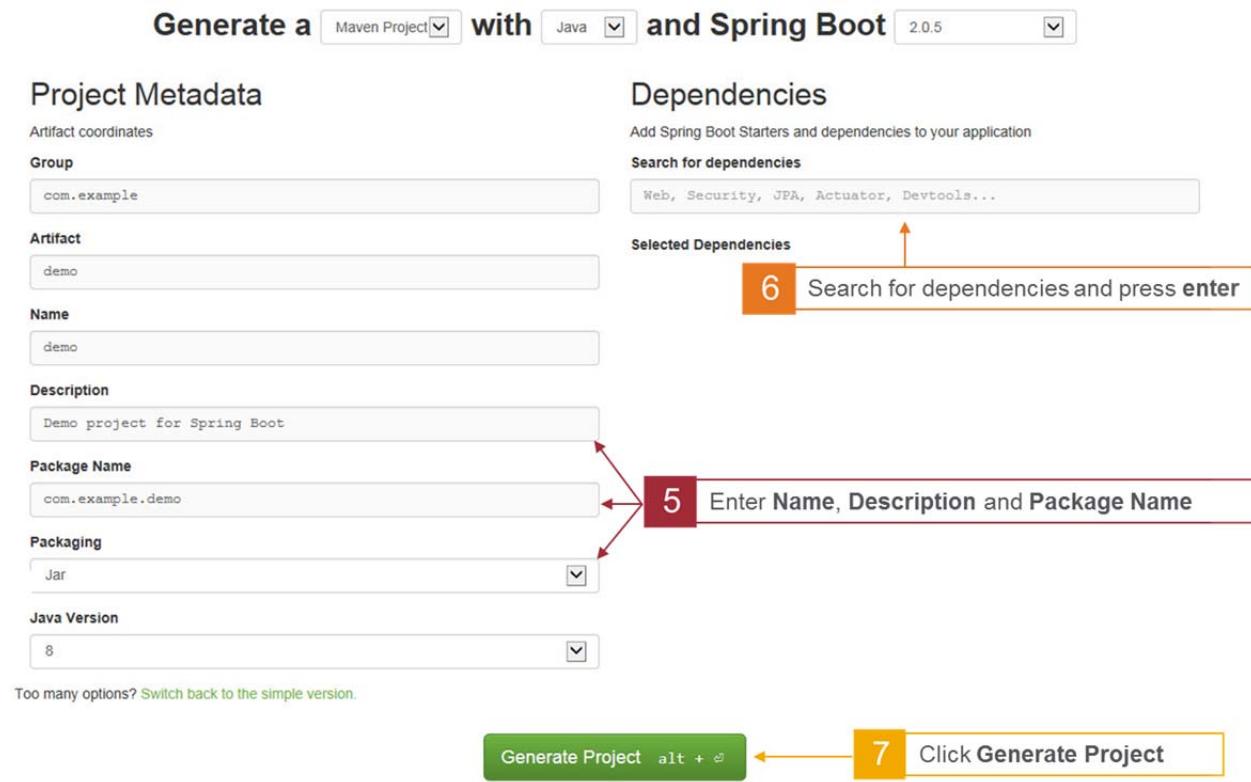
Dependencies

Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies

6 Search for dependencies and press enter

5 Enter Name, Description and Package Name

7 Click Generate Project



5. Enter Name, Description and Package Name.
6. Search for dependencies in the Dependencies section and press **enter** to add them to the project. By scrolling down, the list of all dependencies available can be viewed and required ones can be selected.
7. Once you have selected all the dependencies and entered the project details, click **Generate Project** to generate your project.

Boot CLI Setup

Spring Boot CLI is a command line tool that helps in quickly prototyping with Spring. We can also run Groovy scripts using this tool. Checkout the Official Spring Boot documentation for instructions on how to install Spring Boot CLI for operating system. <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-installing-the-cli>

Once installed, type the following command to see what can be done with Spring Boot CLI -

```
1 | $ spring --help
```



The Spring Boot CLI is a great way to develop Spring applications by just writing code. However, development on a more traditional Java project can be kick-started via the Spring Boot CLI as well.

Steps to Scaffold Boot Application with Spring Boot CLI

We'll look at the spring init command to generate a new Spring Boot project.

Spring init command uses the web-app. Spring Initializr, internally for generating and downloading the project.

Open your terminal and type:

```
1 | $ spring help init
```

This shows all the information about the init command.



The following are a few examples of how to use the command:

1. To list all the capabilities of the service:

```
1 | $ spring init --list
```

2. To create a default project:

```
1 | $ spring init
```

-
- The init command that acts as a client interface to the Initializr.
 - Most simple use of the init command is to create a baseline Spring Boot project.

Interactivity: Poll Question

Question: What command would you use in Spring Boot to create a default project?

- A. \$ spring init
 - B. \$ spring help init
 - C. \$ spring init --list
 - D. \$ spring --help
-



Steps to Scaffold Boot Application with Spring Boot CLI (Cont.)

3. To create a web my-app.zip:

```
1 |
```

```
$ spring init -d=web my-app.zip
```

4. To create a web/data-jpa gradle project unpacked:

```
1 |
```

```
$ spring init -d=web,jpa --build=gradle my-dir
```

5. Let's generate a maven web application with JPA and Mysql dependencies -

```
1 |
```

```
$ spring init --name=demo --dependencies=web,data-jpa,mysql,devtools --package-name=com.demo demo
```

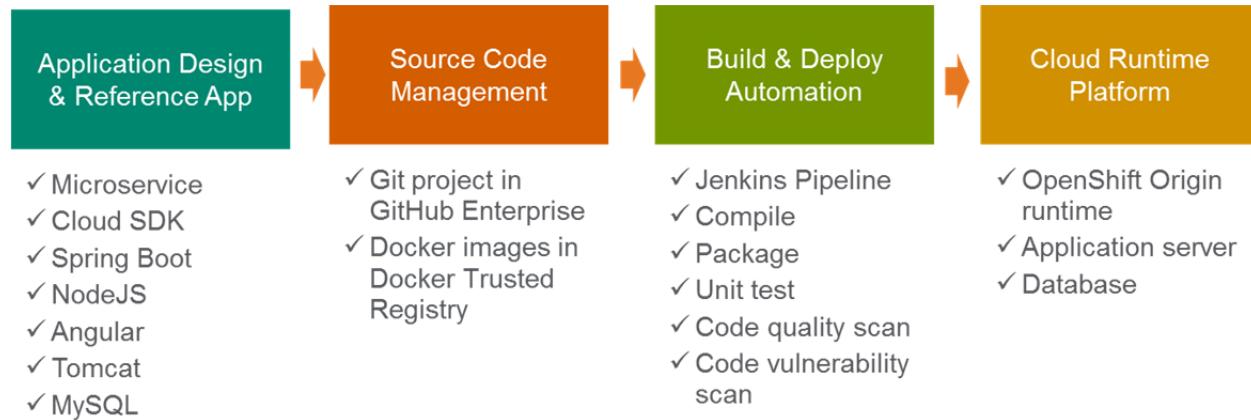
This command will create a new Spring Boot project with all the specified settings.

Almost every IDE has support for Spring Initializr which uses CLI and in turn its all spring.io.

- init command will conclude by downloading a demo.zip file after contacting the Initializr web application
- Project has a Maven project structure with pom.xml build specification.
- Maven build specification is minimal, with only baseline starter dependencies for Spring Boot and testing.
- By default, the build specification for both Maven and Gradle builds will produce an executable JAR file. If you'd rather produce a WAR file, you can specify this with the -- packaging -p or parameter.

Cloud Scaffolding

The Cloud Scaffolding Jumpstart API instantly enables software engineers to quickly start developing software using enterprise application build and deployment solutions on a proven technology stack.



www.optumdeveloper.com/scaffolding

The Cloud Scaffolding Jumpstart API instantly enables software engineers to quickly start developing software using enterprise application build and deployment solutions on a proven technology stack. A single API call creates a sample reference application, along with a GitHub Enterprise and DTR repo, a Jenkins pipeline that includes compilation, packaging, along with quality and vulnerability scanning. The pipeline also deploys your sample application to the OpenShift Origin platform.

Cloud Portability Patterns

Portability Definition and Theory

A solution can be called portable if the infrastructure is repeatable, automated and adding new servers requires nothing more than a DNS change.

1. Since a universal set of standards does not exist and most likely won't exist shortly, cloud ecosystems can encounter a significant risk of vendor lock-in.
 2. A typical scenario is an inability to migrate some components to the cloud due to data management or data sovereignty regulations.
 3. Cloud migration requires portability of all migrating components as well as interoperability of those components with systems that remain on-premise.
-

Interoperability between clouds presents its own challenges. Here are the issues:

- Trust and security.
- Prevention of vendor lock-in.
- Every cloud calls the same thing a new name.

Where to Look out for Portability?

Specific technology categories where portability and interoperability standards should be specified include the following:

Data: Enabling the reuse of data components across different applications. Since data interoperability interfaces do not currently exist, this may require the use of data virtualization techniques.

Applications: This focuses on interoperability between application components. These have SaaS deployed components, application modules leveraged in a PaaS, or infrastructure components consumed as IaaS.



- Similar issues arise in a hybrid environment when interfacing with a traditional enterprise IT environment or with client endpoint devices.
- Application portability enables the re-use of all application components across the entire hybrid IT environment.
- *cloud on-boarding* is of particular concern for cloud computing.

Where to Look out for Portability (Cont.)

Platforms:

This category addresses the re-use of service bundles that may contain infrastructure, middleware, or application components along with any associated data.

Infrastructure:

Interoperability and portability associated with various hardware virtualization technologies and architectures.

Management:

It is the interoperability capability that is required while interoperating between various cloud services and programs concerned with the on-demand self-service implementation

Publication and acquisition:

The self-service aspect of cloud computing gives end users the ability to acquire software, data, infrastructure and various other cloud services. Developers can also publish applications, data, and cloud services via online marketplaces. This category addresses interoperability between platforms and cloud service marketplaces, including app stores.

- Management may also include application programs concerned with the deployment, configuration, provisioning, and operation of cloud resources.
- The application, platform, and infrastructure components can be hosted in traditional enterprise computing platform – in-house, or they can be made available as cloud resources - software application programs (SaaS), software application platforms (PaaS), and virtual processors and data stores (IaaS).

Advantages

Advantages of portability:

1. Easy recovery from outages (DR)
2. Change hosting partner to save money

Next, we will see some tools which help to automate some tools with portability and operability of cloud enabled applications.



-
- Hardware and virtualization architectures provide portability and interoperability.
 - The interfaces are mostly internal to the IaaS and infrastructure components shown in [Data, Applications, Platforms, and Infrastructure](#).
 - Only physical communications interfaces are exposed similar to traditional computing.

Interactivity: Poll Question

Question: Which of the following is a benefit of portability?

- A. Unchangeable hosting partner
 - B. Easy recovery from outages
 - C. Users share data
 - D. Expensive to install
-



How to Configure and Setup with Code?

Below are some of the well-known configuration tools available which can automate server setup.



-
- Ansible is software that automates software provisioning, configuration management, and application deployment.
 - Chef - The Continuous Automation platform for delivering infrastructure, compliance, applications, and whatever comes next.
 - Allows us to build, deploy, and manage everything in the pipeline and in production. Automation ensures a single consistent, collaborative, and transparent mode to ship any application to any environment. Automation also delivers and operates all software across its entire lifecycle.

How to Provision Instances?

Once automation for server setup and configuration is in place, next should be provisioning automation. The below script will bring the `nginx` machine up and ready.

Configuration Management

Infrastructure as code

```
run_list['webserver']

webserver/recipe/default.rb
include_recipe 'nginx'
```

-
- Nginx is a web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.
 - The example shown above is for showing how infrastructure can be setup with code.
 - Automated provisioning helps with setting up infrastructure.

Cloud Instance Setup with Scalr

Scalr like cloud management platforms may help in managing these cloud setups.



- Scalr targets its cloud management platform at organizations with multi-cloud strategies and aims to give users a single interface to manage multiple cloud environments.
- Although cloud has its own machine management software, having a cloud agnostic helps in moving between them.

When to Provision Instances?

For this problem we have different monitoring tools which may decide scale and availability.

Automation is better!
Let machines do the work within rules you define.

1. Nagios is monitoring your servers
2. When a metric (CPU load, # connections, etc) reaches your threshold
3. Nagios sends alert
4. Jenkins responds to alert by spinning up new/more machines

-
- Nagios, now known as Nagios Core, is a free and open source computer-software application that monitors systems, networks, and infrastructure.
 - Used for network monitoring, server health monitoring and scaling by adding new machines.

Discussion: Microservices Advantages and Disadvantages

Questions:

- What are the pluses of using microservices?
- What challenges do they pose?
- What are the good practices for breaking a monolithic app into microservices?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Discussion: Spring Framework

Questions:

- What is the Spring framework?
- How does it implement port binding for twelve-factor apps?
- Which parts of Spring are especially useful for twelve-factor apps?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Lab: Spring Cloud Config Server

Overview:

Time: 15 Minutes

In this exercise, you will:

- Generate and setup Spring Cloud Config Server to hold maximum and minimum configuration properties and will run on port 8888. This service will serve limits-service with properties.



Instructions:

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Module Summary

Now that you have completed this module, you should be able to:

- Describe the basics of Microservices
 - Describe Spring Cloud basics
 - Explain Service Development with Spring Cloud
 - Describe Spring Boot and Spring Framework
 - Describe Cloud Application Scaffolding
 - Identify Cloud Portability Patterns
-

CI/CD

Module 7

Module Objectives

After this module, participants will be able to:

- Describe the Build Lifecycle at Optum.
 - Describe the goals of Continuous Integration and Continuous Delivery.
 - Explain the purpose of version control.
 - Understand Git as a version control system.
 - Explain the goals of Maven and Jenkins.
 - Describe common use cases.
-

The Build Life Cycle

Agile

Agile development:

Incremental software development

Empowers people to collaborate and make team decisions

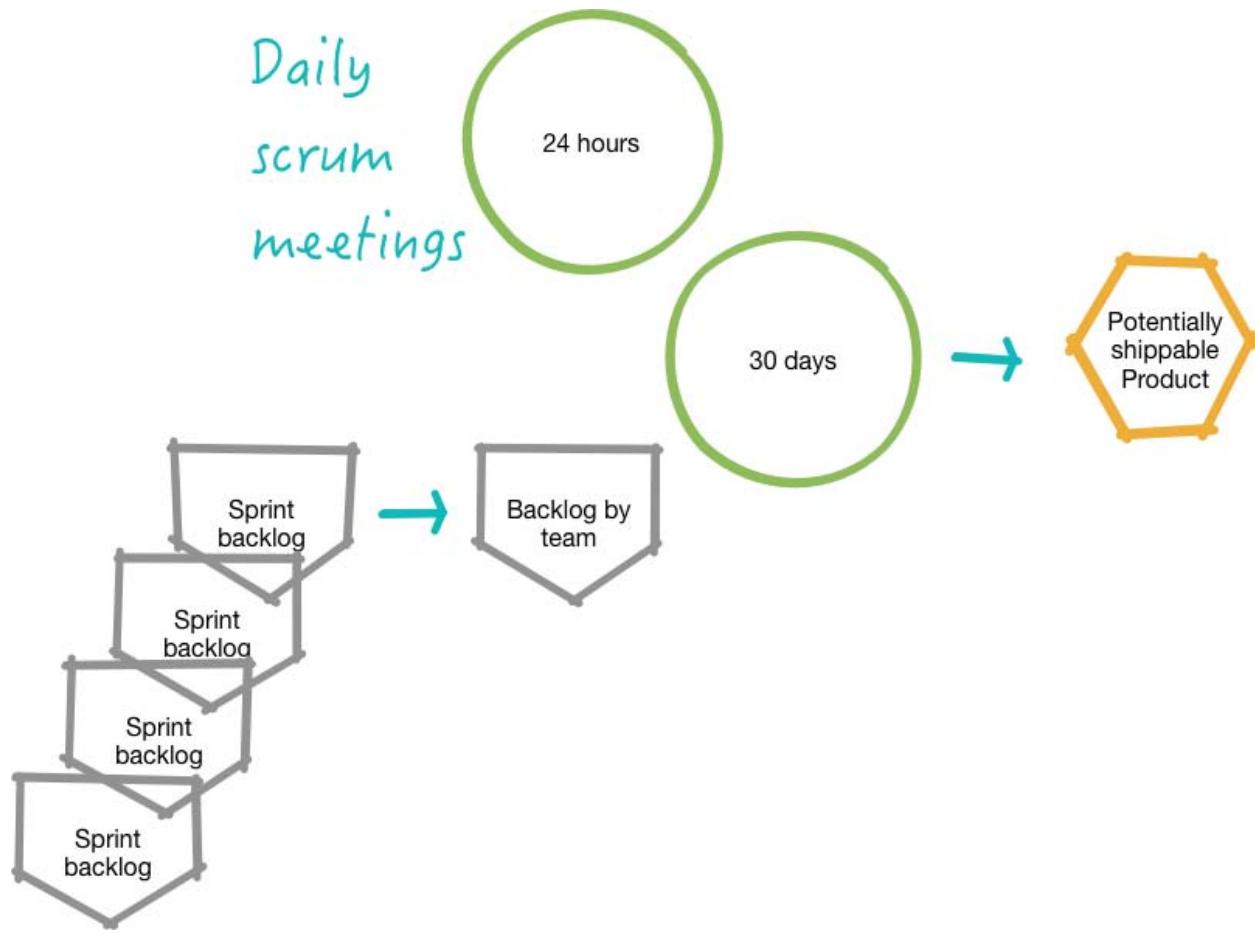
Continuous planning, testing, integration, and delivery



Agile is the style of development naturally supported by Jenkins. In Agile, the development happens continually, as explained in the following slides.

Continuous Integration (CI) and Continuous Delivery (CD) with Jenkins assure that the software remains coherent and as bug-free as possible, even with rapid changes in the development cycle.

Agile Events Flow



This diagram explains the time flow of Agile development.

It also introduces the key terminology for teams involved in the Agile lifecycles.

Enabling Methodologies



FDD stands for Feature-Driven Development. All these are different software development methodologies emphasizing different style and goals of development in various organizations.

Manifesto

Manifesto for Agile Software Development



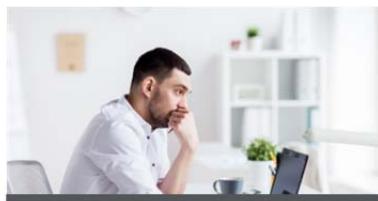
Individuals and interactions over processes and tools



Working software over comprehensive documentation



Customer collaboration over contract negotiation



Responding to change over following a plan

This manifest is self-explanatory. However, students may find it useful to discuss the possible use cases for them.

Especially, try to find situations where not adhering to these principles led to difficulties and had put projects in jeopardy.

Agile Development Practices

Test Driven Development

Automation is the Key

Continuous Integration (CI)

Continuous Delivery (CD)

Definition of Done

Common “War-room” Style Work Area

Product Backlog and Spring Backlog

Sprints and Daily Sync

Burn Down and Velocity Charts

Sprint Review and Retrospective

Simple Design and Regular Refactoring

Pair Programming

These, too, are well-known practices. Here, the students may find it beneficial discussing their justification for each. For example, they may say something like this.

Pair programming seems at first as a waste of resources: two people working on the same problem together. However, once you try it, you will appreciate the psychological benefits of it.

When one person is ‘driving’, he is tense and prone to make mistakes, overlook good solutions and best practices. The ‘rider’ in this situation is in a good position for making off-handed remarks which are all easy for him to make but prove invaluable to the driver.

Special attention should be given to CI and CD. Nevertheless, there are fairly recent articles on the statistics of CI and CD. They put the numbers of significantly implemented CI and CD in the 30% to 40% range at best. It may be instructive to check on the state of the CI and CD implementation, enterprise-wide, at the time of using this course for training.

CI and CD

Continuous Improvement and Continuous Delivery become essential ingredients for teams doing iterative and incremental software delivery in Agile Development.

- Developers share the common source code repository
 - Dedicated Continuous Integration environment
 - All code must pass unit tests
 - Integrate often
 - Regression tests run often
 - Code matrices are published
 - Every change to the system is releasable to production
 - Automation is the key
-

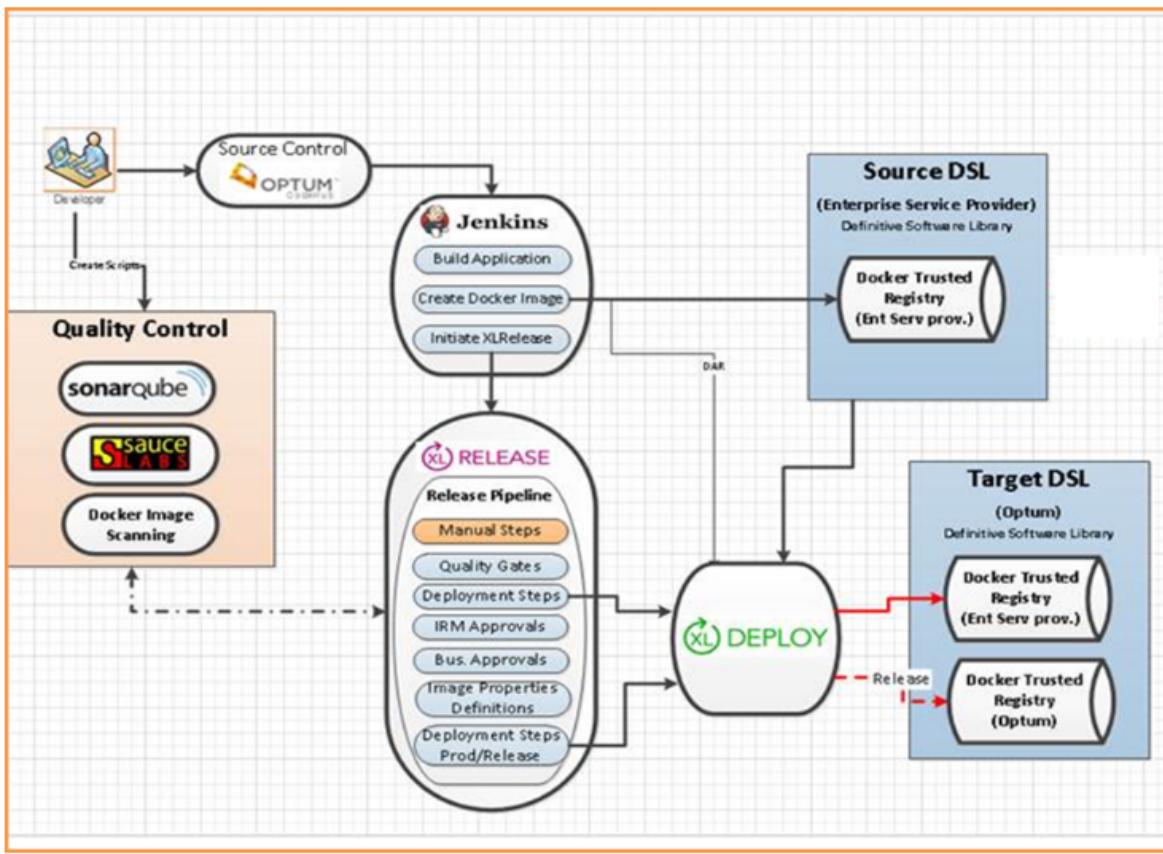
These are the major points of CI and CD. First, the students must make a clear distinction between what CI and CD are.

Second, they might talk about how much of CI and CD is implemented in their organization.

Next, they might talk about the potential benefits and challenges of this implementation.

CI/CD Flow within Optum

Sample CI/CD flow for building and using Docker Image



What we see here is a sample CI/CD flow within Optum. We will cover Jenkins' role a bit later in the module, and we will look more closely at Docker images in the final module of this course.

The two major technologies used for CI/CD at Optum are Jenkins and Docker. Jenkins is used for automating CI/CD, and Docker is the target platform. Accordingly, Jenkins gets the source code from the Source Control and instantiates the release. In doing so, it uses other components and plugins such as SonarQube, Source, and Docker Image Scanning. The release cycle, as it applies to Docker images, includes manual steps, quality gates, IRM approval, and so on. The resulting images are then published to various Docker registries.

Interactivity: Poll Question

Question: The two major technologies used for CI/CD at Optum are Jenkins and what?

- A. Kanban
 - B. FDD
 - C. XML
 - D. Docker
-



Introduction to Git

What is Version Control?

Version control, essentially, is a system for managing changes.

What one would like in version control:

- Revert code changes
- Never lose code
- Maintain multiple versions of a product
- See the difference between two (or more) versions of your code
- Prove that a particular change broke or fixed a piece of code
- Review the history of some code
- Submit a change to someone else's code
- Share your code, or let other people work on your code
- See how much work is being done, and where, when, and by whom
- Experiment with a new feature without interfering with the working code



These are all the advantages of a version control system, as an idea.

What other features of a version control would you want to have that are missing?

Version Control Systems

Server-based:

- CVS
- PVCS
- SourceSafe
- Subversion

Distributed:

- Git
- Mercurial



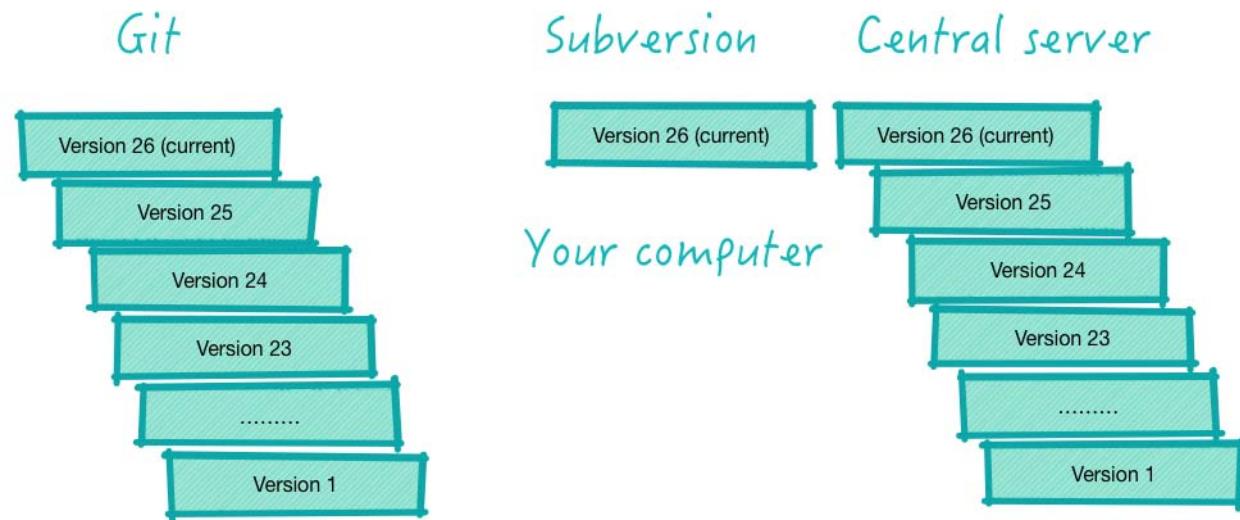
Version control systems went through an evolution. Initially, they all relied on the presence of a central server. This meant that without a central server, work was coming to a stop.

Another problem was merging the changes. Since server-based systems lacked the ease of merging from the central branch to the development one, you could not easily keep in touch with the main thread of development. It was not unheard of to spend weeks on merging the changes from a major development effort.

This problem is solved with Git: you simply merge the changes from your colleagues into your branch and thus keep in sync while developing new features.

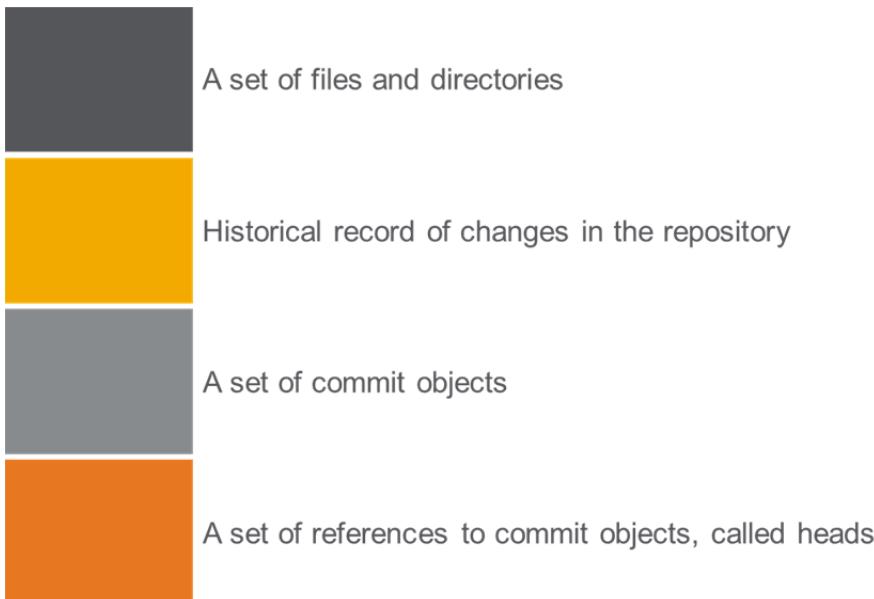
An early version of SourceSafe would lock the files! You had to walk to your colleague's office and knock on the door, asking to return the file he was working on and unlock it. Can you imagine doing that today?

Git vs. Subversion



This picture illustrates the concept of cloning vs. checkout. In cloning (Git), each repository stores a complete history, with all versions. By contrast, in checkout (Subversion), the central server stores all repositories, while the developer's local repository contains only one version at a time, in its own separate directory.

Repository



The purpose of Git is to manage a project, or a set of files, as they change over time.

Git stores this information in a data structure called a repository.

A repository is simply a place where the history of your work is stored.

It often lives in a .git subdirectory of your working copy - a copy of the most recent state of the files you're working on.

Is it a Repository?

- A copy of a project directory?
- CVS?
- Subversion?
- Git?



Git is a complete repository, whether it is local or remote.

Working Copy

Aka "working directory," is a single checkout of one version of the project.

- "Index" and "Staging area" are synonyms
- It is a simple file in the Git directory
- Stores information about the next commit
- It involved blobs and trees

Local Operations



Checkout the project



A "local repository" is a copy of your project on your computer, together with some files generated by Git. In a local repository, there are three areas. The working directory are your files.

"Staging area" is found in the .git directory. It is generated by Git and contains files ready to be put into Git.

The "Git repository" area is also under the .git directory. It contains all files that are tracked by Git.

A remote repository contains all of the same files, just that they are stored on a publicly-accessible server.

Cloning

Getting a copy of the existing git repository (quick, what is repository?)

How? git clone <url>

Example: \$ git clone git://github.com/schacon/grit.git



In cloning from a remote, you get all of the history of the project, with all of the versions, branches, and tags. All of that is copied from the remote to your local directory.

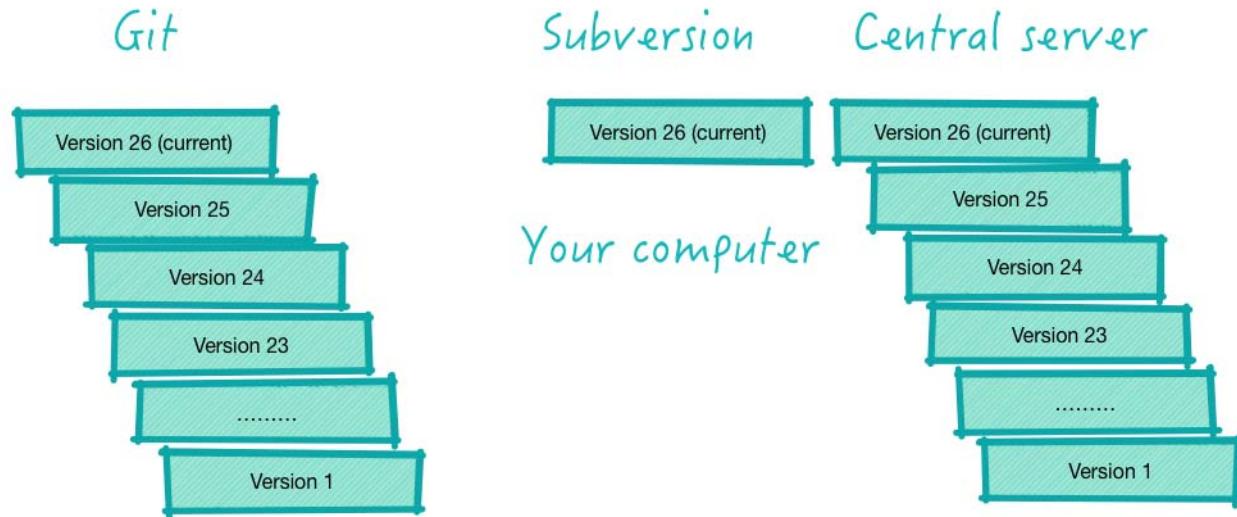
The purpose of Git is to manage a project, or a set of files, as they change over time.

Git stores this information in a data structure called a repository.

A git repository contains, among other things, the following: A set of commit objects.

Cloning vs. Checkout

In Subversion, this would be *checkout*. Differences?



Both your repo and public

Cloning copies the complete Git repository. This includes the history to the beginning of time, with all modifications.

This is different from a checkout, where you get only one slice of the development history, namely, one version related to a certain point in time.

git clone is to fetch your repositories from the remote git server.

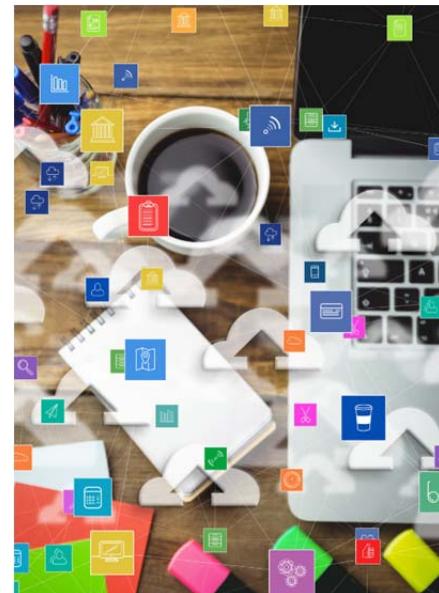
git checkout is to checkout the desired status of your repository (like branches or particular files).

Remotes

Versions of your project that are hosted on the Internet or network – that's how you collaborate.

Remotes can be:

- Multiple
- Read only
- Read-write



All that time that you develop with Git, the contents are stored on your computer only. This includes all the version, but still, that is local.

When you want to share this with the world, you use the remote repository. The remote, too, has all of the versions, so it is a complete copy of your local. The difference is that the remote is available publicly, albeit it may be protected with a password.

Local History vs. Public History

Local history is on your laptop.

You can:

- Change commits
- Change commit messages
- Reorder
- Squash

However, be careful pushing this to the public history because other developers may end up having to merge.

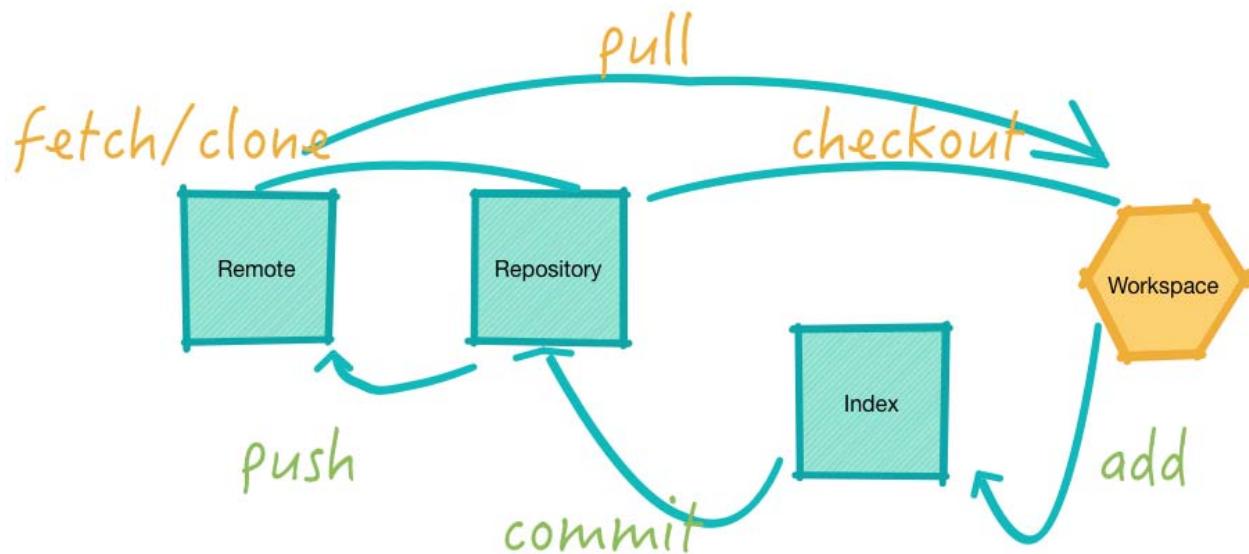


All of git history is stored on every participant's computer. Thus, all developers are code guardians and custodians. However, care needs to be taken not to ruin everyone's work.

This can happen when a developer messes up with the public history of the project.

Basic Git Operations – Diagram

Here is the basic cycle of events in Git



1. All the work is in the Workspace.
2. Commit pushes it to Index, that is, it is ready in the repository.
3. Push uploads it to the remote.
4. Others may pull the changes to the Workspace.
5. If you want to switch to another branch, you do a 'checkout'.

Interactivity: Poll Question

Question: How do you switch to another branch in Git?

- A. Make a working copy.
 - B. Clone the repository.
 - C. Run the `index` command.
 - D. Do a ‘checkout.’
-



Basic Git Operations

Viewing a commit in UI

Execute the commands below

In Git Bash

- gitk

Or

Tools-Git Shell

- gitk

To view a specific commit

- git show 5809 (first few letters of the SHA-1)

Switching branches

- git checkout <branch-name>



Basic git operations are done on the command line. However, many developers prefer using graphical tools with UI. Both approaches are popular. We will not be covering specific commands or functions in Git, as there are already courses within Optum University that cover these skills. Please see your manager if you are interested in taking these courses.

Jenkins and Maven Defined

What is Jenkins?

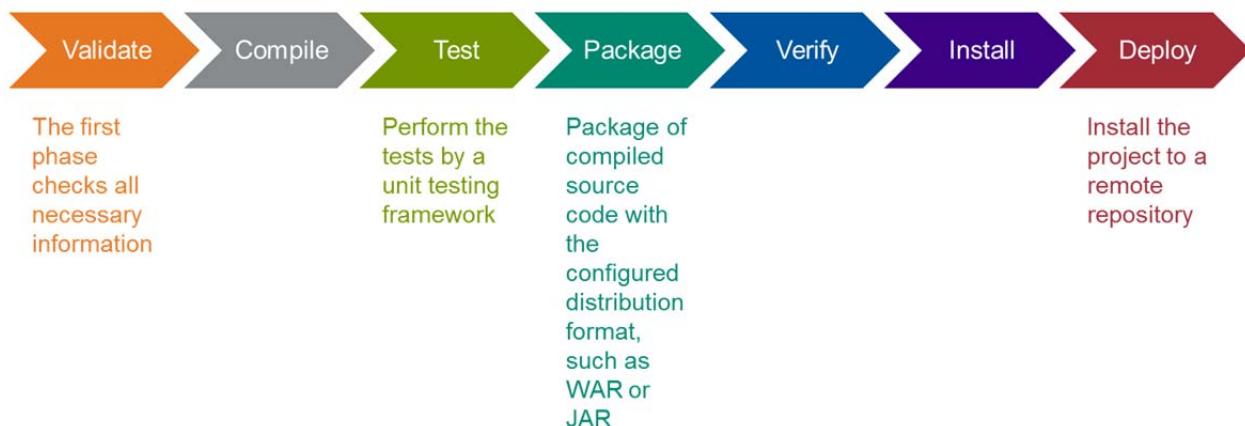
- Continuous Integration (CI) Server
- Allows automated building of the source code and testing it
- Has over a thousand plugins for all possible additional purposes
- Integrates with quality control tools like SonarQube

What is Maven?

- Code build system (in use by Jenkins)
- Permits inclusion of required dependencies by pulling them from central repositories
- Allows builds of any complexity

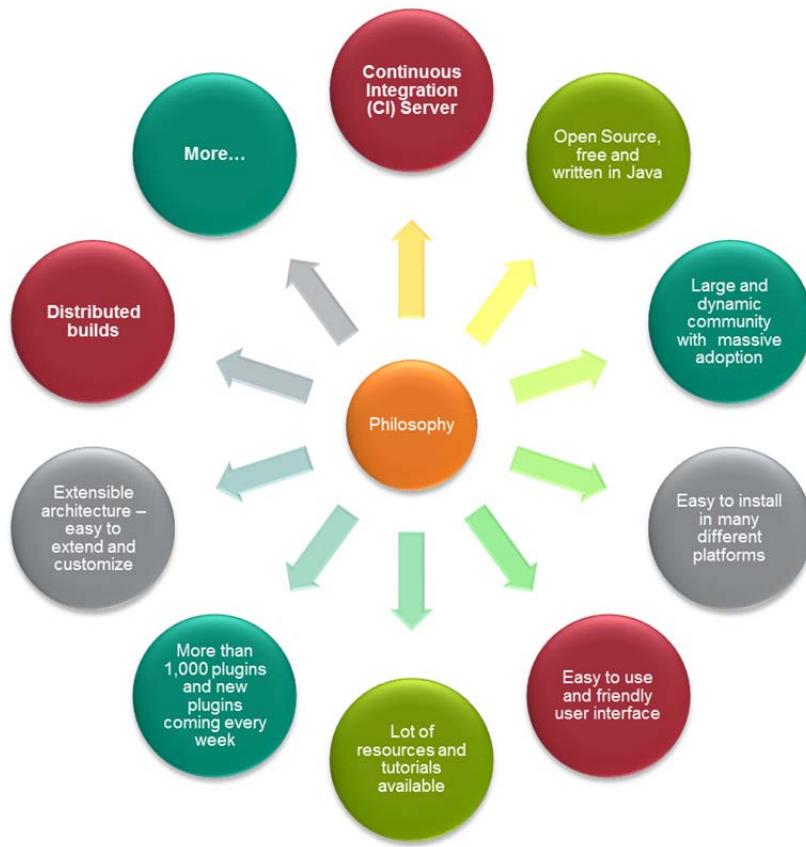
Jenkins started its life as a CI server. Later, this tool, together with the supporting organization, CloudBees, became one of the significant influencers and voices in the software development methodologies.

Build Lifecycle



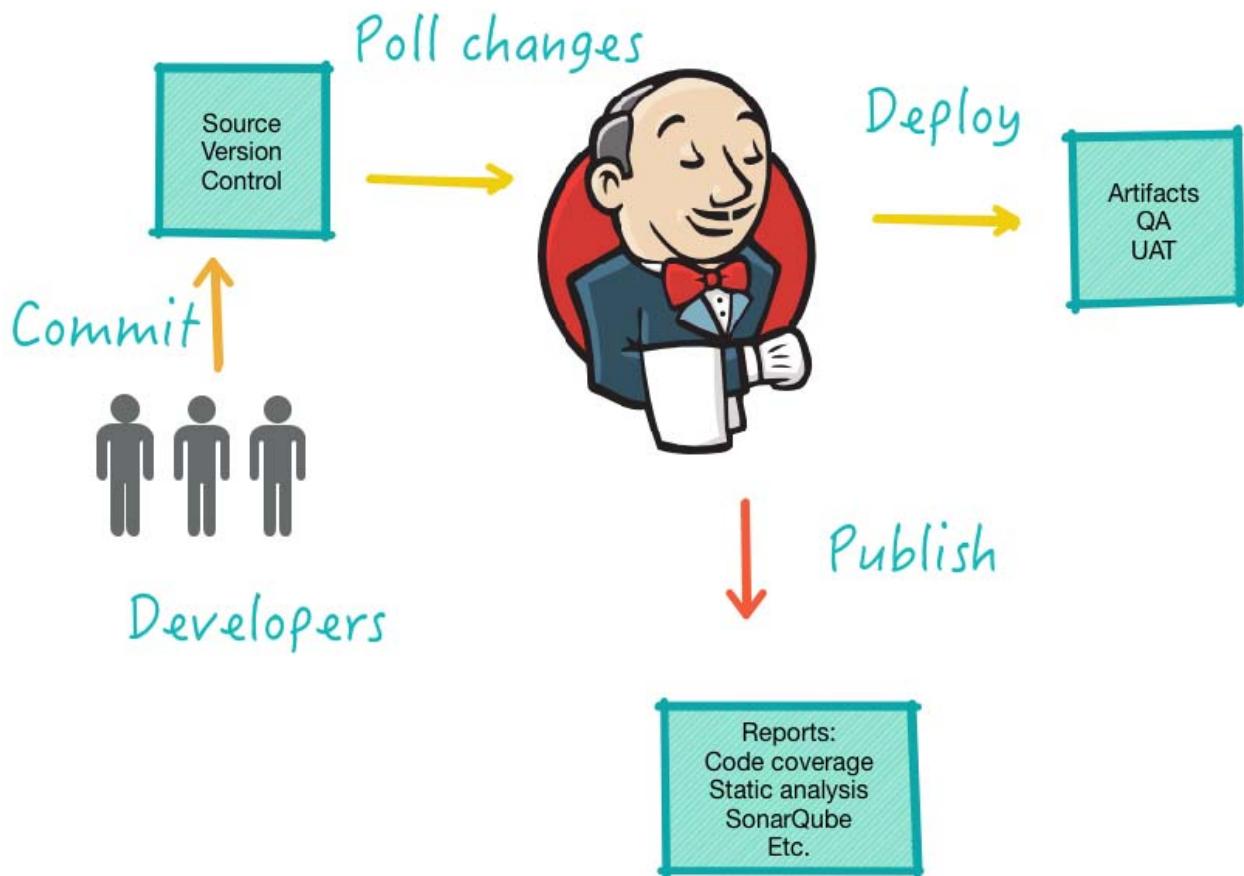
These are the major steps in the software development build lifecycle. It will be instructive, as we go through Jenkins, to point out which of these steps are implemented in each part of the Jenkins functionality.

More About Jenkins



The specific advantages of Jenkins and the reasons for its adoption are enumerated here. These will be illustrated later as we go through Jenkins in more details.

How does Jenkins Fit in with CI?



Exactly how does Jenkins play into the CI? This diagram explains the flow of events in the regular Jenkins build job.

Jenkins History

- Jenkins is formerly known as Hudson.
- Hudson was first released by Kohsuke Kawaguchi of Sun Microsystems in **2005**.
- Initially, it was only used within Sun, by **2010** Hudson captured 70% of the CI market share.
- Oracle bought Sun Microsystems in **2010**.
- Due to naming and open source dispute, the Original Hudson team created a new project, Jenkins forked from Hudson.
- Oracle continued the development of Original Hudson.
- Majority of Hudson users migrated to Jenkins within few months of initial Jenkins phase.



Jenkins history is good background information to learn from. Incidentally, looking at some Jenkins' not-so-polished features, one might be surprised to find them in a product that is definitely a leader in the field. This can likely be explained by the open source nature of it. Why fix what works? And, the product is free, so more money is not tied to more polish. Nevertheless, students might be tempted to implement their suggestions and offer them as pull requests on the GitHub for this project.

Continuous Integration is Elusive

Jenkins is king; however, survey says:

14%

Deploy on an hourly basis

34%

Deploy once a day

21%

Deploy weekly

31%

Deploy less often than weekly

<https://www.infoq.com/news/2017/03/agile-king-ci-ilusive-goal>

Here are the numbers that show the actual state of affairs in CI and CD. They are far from perfect. Students may want to compare these numbers to what they see in their own organization.

Discussion: CI/CD Concepts

Questions:

- What is CI?
- What is CD?
- Are these worthy goals achievable?
- How does Jenkins help with CI and CD?
- What is the state of CI, CD, and Jenkins in your organization?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Installing Jenkins (Demo)

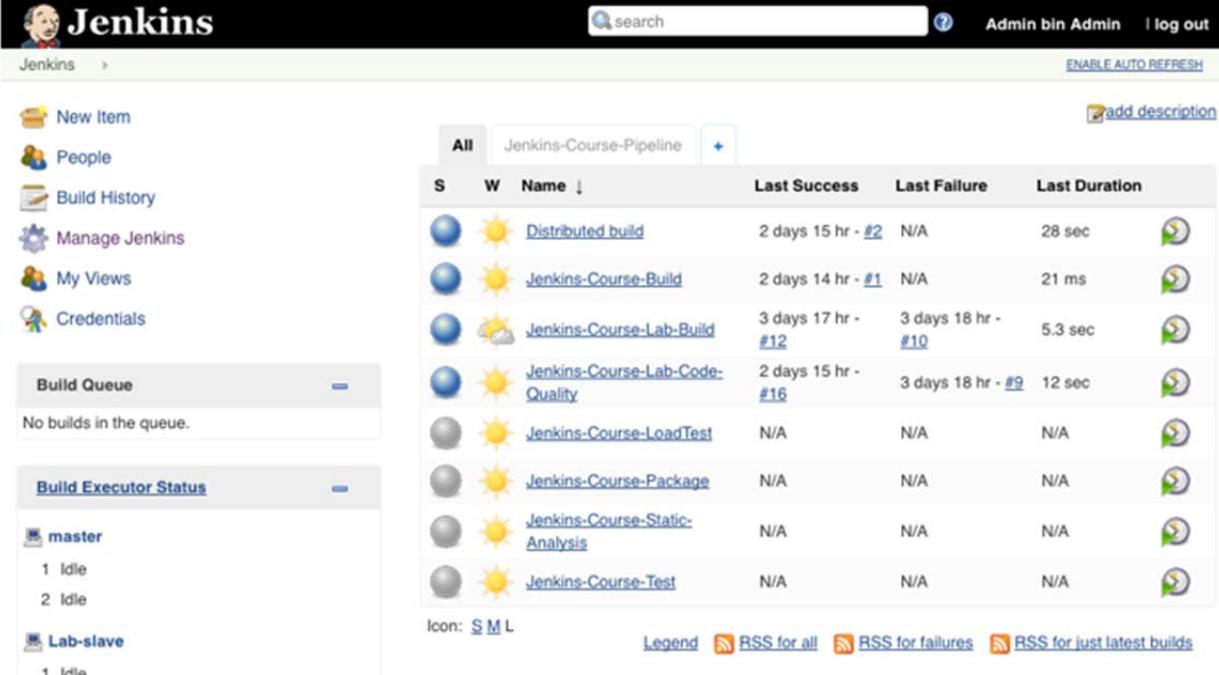
- Easy to install on different operating systems
 - Also available as installer or native package
 - A couple of options for installing Jenkins
 - Run as a standalone application by launching with java -jar
 - Deployed on Servlet Container
 - Use platform-specific package or installer
 - Java is the only requirement for installing Jenkins
 - Install latest Java
 - Set JAVA_HOME environment variable
 - Demo/Lab 1: Install and Configure Jenkins
-



Installing Jenkins is described in the accompanying lab. The trainer may choose to have an instance with the install to constantly store in the cloud, whether internal or external. Then, this instance can be brought up before the beginning of the class. Later on, it can be stopped, to save money.

Running Jenkins with All Demos

Verify Jenkins by pointing your browser to <http://localhost:8080>



The screenshot shows the Jenkins dashboard with the following details:

- Left Sidebar:** Includes links for New Item, People, Build History, Manage Jenkins, My Views, and Credentials.
- Build Queue:** Shows "No builds in the queue."
- Build Executor Status:** Shows 1 master (idle) and 1 Lab-slave (idle).
- Central View:** A table showing Jenkins-Course-Pipeline jobs. The columns are S (Status), W (Last build), Name, Last Success, Last Failure, and Last Duration.

S	W	Name	Last Success	Last Failure	Last Duration
		Distributed build	2 days 15 hr - #2	N/A	28 sec
		Jenkins-Course-Build	2 days 14 hr - #1	N/A	21 ms
		Jenkins-Course-Lab-Build	3 days 17 hr - #12	3 days 18 hr - #10	5.3 sec
		Jenkins-Course-Lab-Code-Quality	2 days 15 hr - #16	3 days 18 hr - #9	12 sec
		Jenkins-Course-LoadTest	N/A	N/A	N/A
		Jenkins-Course-Package	N/A	N/A	N/A
		Jenkins-Course-Static-Analysis	N/A	N/A	N/A
		Jenkins-Course-Test	N/A	N/A	N/A

- Bottom Navigation:** Includes icons for S M L, Legend, and RSS feeds for all, failures, and latest builds.

This is the final state of the Jenkins install, after all the steps in the accompanying lab have been performed. For deeper learning, the students may be encouraged to perform the labs, and then compare their results of these to the trainer.

Maven with Jenkins

- Maven build steps are simple and easy to configure.
- You only need to enter the Maven goal that you want to run.



One of the most popular build tools to go with Jenkins is Maven. In fact, it is implemented by Jenkins as a default. However, Gradle is also very popular and arguably fits better with the Jenkins architecture.

Use Cases

- CI (of course)
 - Push mobile apps to devices
 - Execute anything (such as Matlab)
 - Run all kinds of configurations
-

This slide describes the common popular use cases.

- CI is, of course, the primary use case, being that Jenkins is a de-facto standard for CI implementation.
- Pushing mobile apps to devices is another popular use case. In general, you can make deployment a part of a Jenkins job. Officially, this is part of CD.
- In general, one can configure any post-step, not only deployment. For example, one may have Machine Learning that will train a model to work with the new build, and then the trained model may be deployed together with the software's new version. In fact, this may become a more common use case as the practice of Machine Learning Everywhere becomes more common.

Architecture

- Stapler
- Views
- Taglibs
- Persistence
- Plugins



The basic architecture is built on Stapler, with Jenkins classes bound to URLs by using Stapler. The singleton Hudson instance is bound to the context root (e.g., "/") URL, and the rest of the objects are bound according to their reachability from this root object.

Views - Jenkins' model objects have multiple "views" that are used to render HTML pages about each object.

Taglibs - Jenkins defines a few Jelly tag libraries to encourage views to have the common theme.

Persistence - Jenkins uses the file system to store its data.

Plugins - Jenkins' object model is extensible with plugins. Jenkins loads each plugin into a separate class loader to avoid conflicts.

Discussion: CI/CD Scenarios

Questions:

- What major software components are you working on, which build systems do they use, how long do the builds take?
- Do you have a need for distributed builds?
- What is the role of Jenkins automation in your software development process?
- Which parts do you automate (build, test, stress test, etc.)?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Module Summary

Now that you have completed this module, you should be able to:

- Describe the Build Lifecycle at Optum.
 - Describe the goals of Continuous Integration and Continuous Delivery.
 - Explain the purpose of version control.
 - Understand Git as a version control system.
 - Explain the goals of Maven and Jenkins.
 - Describe common use cases.
-

This page intentionally left blank.

Container Fundamentals

Module 8

Module Objectives

After this module, participants will be able to:

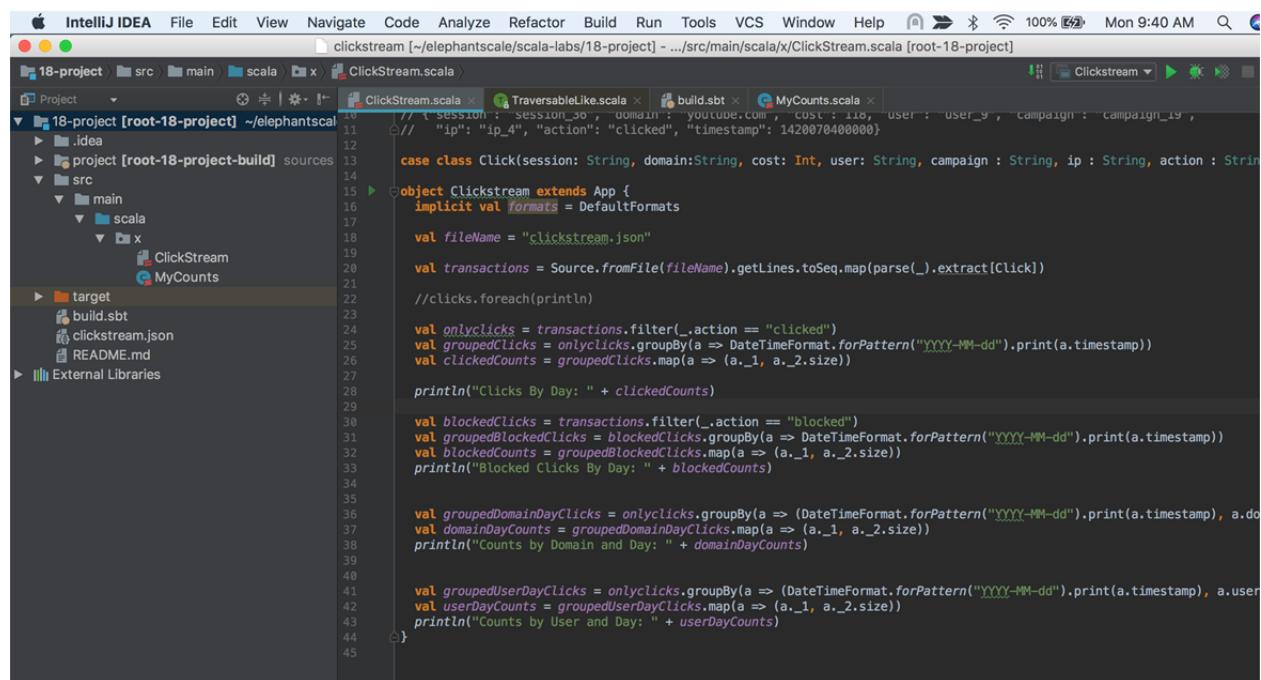
- Gain a general overview of containers.
 - Learn about Docker.
 - Learn how to operate applications with Docker.
 - Describe Docker use cases.
 - Gain familiarity with CLI Tools.
 - Describe health checks of Docker.
 - Describe the anatomy of a Dockerfile.
-

Container Introduction

The Problem

You develop an app on your workstation.

You test it. Seems to run fine. Time to deploy.



```
// t session: "session_30", domain: "youtube.com", cost: 118, user: "user_9", campaign: "campaign_19",
// ip: "ip_4", "action": "clicked", "timestamp": 1420070400000
case class Click(session: String, domain: String, cost: Int, user: String, campaign: String, ip: String, action: String)
object Clickstream extends App {
    implicit val formats = DefaultFormats
    val fileName = "clickstream.json"
    val transactions = Source.fromFile(fileName).getLines.toSeq.map(parse(_).extract[Click])
    //clicks.foreach(println)
    val onlyclicks = transactions.filter(_.action == "clicked")
    val groupedClicks = onlyclicks.groupBy(a => DateTimeFormat.forPattern("YYYY-MM-dd").print(a.timestamp))
    val clickedCounts = groupedClicks.map(a => (a._1, a._2.size))
    println("Clicks By Day: " + clickedCounts)
    val blockedClicks = transactions.filter(_.action == "blocked")
    val groupedBlockedClicks = blockedClicks.groupBy(a => DateTimeFormat.forPattern("YYYY-MM-dd").print(a.timestamp))
    val blockedCounts = groupedBlockedClicks.map(a => (a._1, a._2.size))
    println("Blocked Clicks By Day: " + blockedCounts)

    val groupedDomainDayClicks = onlyclicks.groupBy(a => (DateTimeFormat.forPattern("YYYY-MM-dd").print(a.timestamp), a.domain))
    val domainDayCounts = groupedDomainDayClicks.map(a => (a._1, a._2.size))
    println("Counts by Domain and Day: " + domainDayCounts)

    val groupedUserDayClicks = onlyclicks.groupBy(a => (DateTimeFormat.forPattern("YYYY-MM-dd").print(a.timestamp), a.user))
    val userDayCounts = groupedUserDayClicks.map(a => (a._1, a._2.size))
    println("Counts by User and Day: " + userDayCounts)
}
```

We are introducing the problem of deployment. The problem here is that an application is far more than just code. The code is one small part of an application.

We are introducing the "It worked on my computer" problem. VMs are one way to solve that problem, and Containers are another.

Deploying to the Server

We're ready to deploy to the server.

Provision a new server:

1. Install OS.
2. Install JDK.
3. Install dependency libraries.

Oops... doesn't work. Why not?

- Missing file?
- Wrong version? Path not right?

"But it worked on MY computer!"



Setting up the environment right usually results in problems. Developers have implicit dependencies on libraries, versions, etc. that are hidden. It's often very difficult to figure out what these dependencies are.

Take, for example, a Python application. The developer has already installed dozens of packages for other apps. So, this app "just works" because the developer has certain assumptions.

On a server, those assumptions won't apply, and the app fails to run. Developers are famous for insisting that "It worked on my computer!"

Much Later... Finally Working

Fixed all the problems, now it's finally working!

Oops... don't touch anything!

- Might break!
- Change environment?
- Install another app?



Fix breakage.

- Make things work together.
- Set profiles.

"Write once, run anywhere?"

- Not exactly!
-

Once we fix our problems, we're by no means done. Changes that happen can affect our application. If we install something else on the same server, it can break our application. The problem is that there is no isolation. One app on the server will affect others on the server.

We often have to walk on tiptoes on our server to make sure that we don't break something else. Hosting two apps on the same server is a non-starter.

The old Java maxim "write once, run anywhere" doesn't really work. Because, the Java app itself may run anywhere, but there's so much else that comes into play that will either allow or prevent the app from working.

Redeploying to a Different Server

Great news! Our app is very popular. Need to scale up to a bigger, better server!

No problem, right? Wrong!

Create new server:

- Complete steps: Install OS, install dependencies.
- Fight fires.
- Fix problems.



Here we come to the point where our deployment environment can take on a life of its own, so we can get to a "but it worked on my server!" kind of mentality.

This makes it very difficult for us to move servers, especially if that server is different in any way from the original one.

Some will rightly point out that this is the basic problem that VMs are trying to solve, and, indeed, VMs are a reasonable solution to this.

Redeploy to Cloud

Redeploy to Cloud

- Cloud configuration is different
- Redeploy
- Re-Test
- Fix fires



Cloud is yet another configuration compared to on-prem. How will that affect things? Likely, it will involve even more firefighting.

But It Worked on MY Computer

How many times have you heard this line:

- It worked on my computer!

When we develop, we're not really developing an application:

- We are developing an environment.
- The code itself may be portable.
- But the environment is not!



This is the main point – we are developing an environment. The app is just one small piece of the stack. We have the OS, other applications, the filesystem, the IO devices – all of these are part of what makes the app do what it does.

Real portability means that we have to somehow manage to transport all of these things together with the app. Some will point, again, to the VM as a solution to this problem, but we will see how containers differ from this.

Remember Windows DLL Hell?

You install one application, which has some DLLs such as MSEXAMPLE.DLL.

- Some of those DLLs are shared between applications.
- So, we register them in the COM (Component Object Model) as MSEXAMPLE -> c:\Windows\MSEXAMPLE.DLL.

Now, install a new application, which also needs MSEXAMPLE.DLL.

- It now re-registers the same DLL:
- MSEXAMPLE -> C:\Program Files\MyAPP\MSEXAMPLE.DLL.



DLL Hell is one of the reasons why Windows deservedly gained such a reputation for instability. By putting code in shared libraries, but giving no way to control or version those libraries, early Windows systems could find their way into insidious dependency scenarios where neither app would work. The solution, more often than not, was to simply reinstall everything, including the OS, from scratch.

While modern Windows is more mature, the problem still exists on Windows systems, just like it does on other OS systems. Different apps, once installed, can have side-effects that affect the running of other apps.

To be fair, other OS platforms suffer from the same issue. In fact, it's a pretty universal problem.

Dependency Hell

So what's the problem?

- What if there are several different versions of the same DLL?
- Last one wins! (DLL Stomping)
- First app no longer works!

Uninstall the second app?

- Nope, still doesn't work.

Start over from scratch!

DLL Hell is an example of Dependency Hell

Dependency Hell:

- Multiple dependencies.
- Multiple configurations.
- Applications not isolated.



"Dependency Hell" is the extension of the DLL hell problem to other platforms and situations. Indeed, the fact that we have decades of "DLL Hell" or "Dependency Hell" war stories means that we've not yet come up with the perfect solution to this problem.

Solutions to Dependency Hell

Static Linking	Some Kind of Application Registry	Package Management Systems	Problems With These Solutions
<ul style="list-style-type: none">Include all dependencies in .EXE (Windows)Make a Fat JAR (Java)	<ul style="list-style-type: none">NET GAC (Global Application Cache)	<ul style="list-style-type: none">Apt (Ubuntu/Debian), RPM (RedHat)	<ul style="list-style-type: none">Only good for one application.Filesystem itself not included.No isolation.

Fortunately, Dependency Hell isn't usually a fatal problem. Static Linking is one solution that we see in both Windows and Java – just include all dependencies together in the same .EXE (Windows) or .JAR (java). Since most application code isn't really all that big on a modern system, sometimes trying to share code between multiple applications is just a big waste of time.

There are some ingenious solutions to the problem. Microsoft, for example, invented the GAC (Global Application Cache) for .NET assemblies, to avoid dependency hell problems. Unfortunately, it only applies to managed .NET code and the .DLL hell problems tend to occur more with the unmanaged Windows DLLs.

Even when these solutions work, they only fix the problem for the one application. And, any dependencies on the filesystem aren't helped at all. Nor is there any isolation between processes. This means that these solutions aren't the silver bullet that we would like to see.

The Matrix from Hell

	Dev VM	QA Server	Prod Instance	Prod Cluster	AWS	Laptop
HTML/CSS	???	???	???	???	???	???
Back-End Code	???	???	???	???	???	???
App DB	???	???	???	???	???	???
Analytics DB	???	???	???	???	???	???
Worker Processes	???	???	???	???	???	???
Queue	???	???	???	???	???	???

This “Matrix From Hell” is adapted from the website <http://www.docker.com/>. The idea here is that we have a number of different components to our app: front-end, back-end, various DBs, services, queues, etc. These things all come together to make our app, like pieces of the puzzle. Those pieces will have to be assembled in just the right way on each of our deployment platforms, from local to test, to QA. How do we ensure that our app works the same way on each of them? There are many combinations of factors, which is why we call this the “Matrix from Hell”.

It becomes a big headache to make sure that all the various components of our app, on the rows of the matrix, all work properly on the columns of our matrix, which are the various deployment platforms.

Interactivity: Matching Question

Question: Match each dependency hell solution to its characteristic.

Instructions: Raise Hand to volunteer and then use the Line tool.



Package Management Systems

NET GAC

Static Linking

Apt, RPM

Application Registry

Include all dependencies in .EXE

Wouldn't It Be Nice?

What if we could move all of the following:

- Application code
- Application dependencies
- Configuration files
- Local database(s)
- Environment variables
- Filesystem
- Service processes

All as a "sealed unit"

- Without having to re-deploy
- Reconfigure



Here we're talking about the use case of taking all of our application components and delivering them all as a sealed unit, so that none of the components can be deployed without the others.

We are trying to communicate the value of having our application delivered as a whole. In the next section we'll examine the first way that this problem has been addressed: through virtualization.

Virtual Machines (VMs)

There is a Better Way: Virtualization

Virtualization has taken the Datacenter by storm.

With Virtualization, we can easily move a whole virtual environment from server to server.

We can pause, stop, restart, and terminate instances on the fly.

Frees datacenter ops to focus on server infrastructure and not application.

Agility: much easier to get new VM instance than a new hardware server provisioned.



Virtualization has historically been the answer to these problems. We'll see in a minute the advantages and disadvantages of containerization versus virtualization.

Virtualization allows us to easily start, stop, and move VMs between machines, simplifying many of our deployment headaches.

The History of Virtualization

Mainframes have used VMs for a long time. (1960s/1970-Present)

- Thousands of VMs on a single mainframe.
- Achieves scalability and isolation on mainframe systems.

PC Era (1990s)

- VMs a “mainframe” technology and out of vogue.
- Just run the app on the “bare metal”.



VMs have been around since the mainframe era and allow multi-tenancy much more effectively. As we can see, VMs became much more popular in the mid-2000s as VMWare and other providers came out and CPU vendors started to make hardware virtualization.

That said, there are disadvantages to virtualization. “Bare metal” undeniably runs faster.

History of Virtualization (Cont.)

Application Servers (Late 1990s – 2000s)

- Deploy app to application server.
- Sysadmins managed deployment of application.
- Still "bare metal".

Intel/AMD Support Hardware Virtualization (2006-)

- Enables isolation of VM processes.
- VM security.

In the modern era, virtualization is ubiquitous and assumed -- if I have a server instance I'm sure it's really a VM and not a "bare metal" instance. The ability to copy my instance and redeploy it is something that we normally take for granted in this day and age.

Virtualization Providers



This is just a list of the common Virtualization providers. VMWare is the industry leader in the space, especially for deployment. Windows Hyper-V is used in Microsoft-centric deployment environments. Xen/KVM and others are open-source technologies used in some environments.

All basically work the same way – they are the application team that creates an SM image, and then that image can be deployed at will to send the app (as long as licensing requirements are met). Changes to the app can then go to the VM image, which can then be deployed properly.

The Cost of Virtualization

Performance:

- Virtual hardware doesn't run as well as bare-metal.
- It can be MUCH slower in some cases.

We end up with multiple redundant operating systems for each VM:

- Overall, though, it's worth the cost.
- Our application VMs are sealed from each other.
- Sealed from host OS.
- We can copy VMs between machines without much problem.

Virtual Instances make better use of hardware.

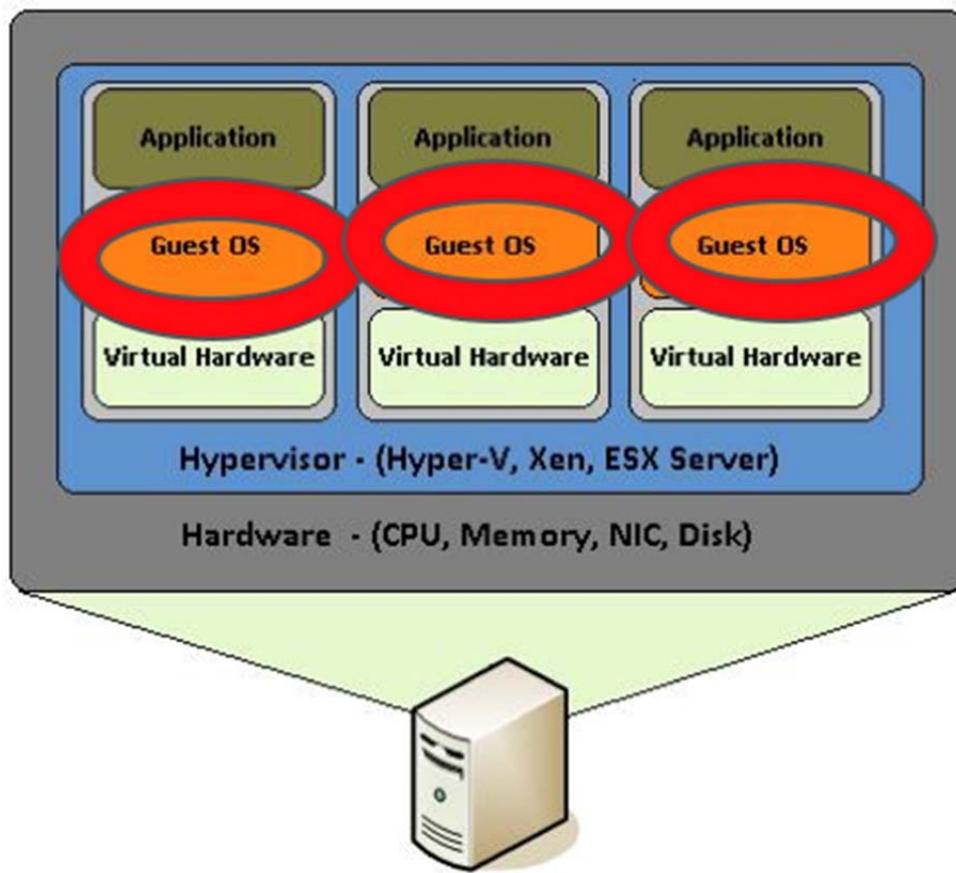


There is a cost to virtualization. It's a cost that's worth it, and this is less than the cost of buying a new computer for every application, but there is no denying that there's a cost. Today, we normally think of the total cost of ownership. If we need more power, we can just make ourselves a bit of a bigger instance than what we would ordinarily expect on bare-metal.

The Problem with Virtualization

Notice how many Guest OS we have here.

Do we really need a separate guest OS for each VM?



This image indicates the problem with virtualization. We are virtualizing an entire machine, including the Operating System.

Of course, this has tremendous advantages in some cases. We are isolating our VMs from each other completely. It also allows us to run different OS platforms simultaneously, which is a primary motivator for desktop virtualization, especially on Mac and Linux platforms (to run Windows apps, for instance).

But most of the time, this is just a waste. The Host itself has an OS running in order to run, then, each of the guests has an OS as well. Do we really need to run copies of the OS?

The solution is often to run the most stripped down, lightweight OS possible in each of the guests. But why do we need the guest OS in the first place – we already have an OS – the host OS!

The performance is one thing, but what about the time it takes to start?

Interactivity: True or False?

Question: True or False? An issue with virtualization is that the entire machine is virtualized, creating multiple copies of the OS.

Instructions: Use ✓ (True) or X (False)



Introducing Containers

What are Containers

Containers are:

- OS Level Virtualization
- Application Code + Filesystem Image code = Packaged Together

How is this different from Virtualization?

- Only the "Userland" is virtualized.
- The kernel is shared with the host.

Both Host and Guest have to be using the same OS (with some exceptions).

- Usually Linux.
- Microsoft has worked with Docker to have Windows containers as well.



Here we introduce containers. Containers are a bit like a VM with no OS. Strictly speaking, the containers DO have some of the OS, but NOT the OS kernel. They give the ability to reuse the OS.

In order for this to work, the kernel has to be the same. In the past, this has meant using Linux, but in modern versions of Docker Windows containers are supported as well.

Containers Versus Virtual Machines

	Container	Virtual Machine
OS	No OS Kernel, use Host's OS Kernel	Each Virtual Instance has its own OS
Filesystem	Dedicated Filesystem	Dedicated Filesystem
Isolation	Uses Virtualization API of host kernel for some isolation	Uses guest isolation for more robust isolation
Guest OS Support	Same as Host (usually Linux)	Can run any OS that supports host CPU
Start/Stop Time	milliseconds	seconds to minutes

Here we compare the similarities and differences between the VM and the Container. The main advantage of the container here is performance. There's minimal startup time for a container, and there's no performance penalty to running on a container.

Note here that the filesystem, libraries, and other dependencies are handled very similarly for containers versus VMs – in both cases, they are on the guest image rather than on the host.

Containers Versus Package Managers

	Container	Package Manager
Filesystem	All dependencies installed to container filesystem.	All dependencies installed to local filesystem.
Isolation	Userland isolation, kernel shared.	No isolation. Processes run together.
Dependencies	Dependencies put in container.	Can have circular dependency issues.

Containers in some ways resemble package management systems like apt on Ubuntu/Debian and rpm/yum on RedHat. In both cases, they are designed to help improve deployment, and are in a sense different solutions to the same problem.

Package Managers, however, can and do have dependency issues, and are generally not suitable for deploying user-developed applications across various platforms.

Solving the Matrix from Hell

	Dev VM	QA Server	Prod Instance	Prod Cluster	AWS	Laptop
HTML/CSS	Container	Container	Container	Container	Container	Container
Back-End Code	Container	Container	Container	Container	Container	Container
App DB	Container	Container	Container	Container	Container	Container
Analytics DB	Container	Container	Container	Container	Container	Container
Worker Processes	Container	Container	Container	Container	Container	Container
Queue	Container	Container	Container	Container	Container	Container

This slide follows up with the Matrix from Hell, showing that the same container is able to run on all of our production, test, QA, staging, and development platforms.

When NOT to use Containers

Multiple OS	VMs allow a single computer to run multiple OS, such as Linux, Windows, etc. For example, we could run a Windows Guest on a Linux Host. This is not possible with containers.
Kernel-Level Isolation	If you need various custom kernels or run kernel-level code, you need virtualization. Containers share the host kernel.
Root or Privileged instructions	Containers can run root, but it only applies to root within the container and not outside.
Tightly-Coupled Data	When the application is tightly coupled to its own data, containers aren't a good solution. This is not generally a good design principle, however.
Security Concerns	Sharing the kernel means that kernel exploits in the container will affect the host as well. Containers should not be used for untrusted code execution.

There are a number of use cases NOT to use containers. Here is some explanation:

Multiple OS: One of the main use cases for desktop virtualization is to run various Operating Systems. For example, Mac users want to run their favorite Windows Application, or Windows users want to run something inside of Ubuntu Linux. Containers don't help in this regard, because they are mainly about sharing the host kernel. That said, we will see how Docker will use a combination of virtualization and containerization on development platforms to allow running containers across different OS platforms.

Kernel Level Isolation. Nothing can beat a VM's isolation at the kernel level. While it may be more efficient to share a kernel, it means we have to share the same kernel. If we want optional kernel modules in one app versus the other, only a VM can do that.

If we want to run privileged code, we can't do that either in a VM or in a container. Of course, in either case we can certainly run as root, but the results will only be seen inside the container or VM, and not outside.

If we have an application that is tightly coupled to its data, then containers will make life complicated. What does it mean to be tightly coupled? Let's say, for example, that the application is designed to store data as local files in its run directory. This is generally a really bad idea, but one that is extremely common. Containers can be complicated because the app is generally designed to run from an immutable container image. Changes to the state of the application are reserved from changing the data, which we want in special attachable volumes.

Security: Running untrusted code is not a good use case for containers. While containers make it more secure to do so than without the container, code run in the container which involves a kernel exploit will then affect the host kernel, and with it the running kernel for all other containers. On the other hand, an exploit running in a VM can, at worst, give the attacker control of the kernel within the VM itself, and in most cases will leave the host untouched.

Container History

History of Containers

1982: Unix "chroot" command: Allows filesystem-level isolation	2000: BSD "jail" command	2005: Linux namespaces: Process isolation	2007: Linux Cgroups: Resource Isolation
<ul style="list-style-type: none"> Allows process and children access to an isolated filesystem. Users, processes, network, and other resources shared. 	<ul style="list-style-type: none"> Allows filesystem, user, processes, and network isolation. Inspired Solaris Zones, which combines jail with dedicated ZFS filesystem. 	<ul style="list-style-type: none"> Segregate processes into groups. Processes can't talk between groups. 	<ul style="list-style-type: none"> Control mem, CPU, resource allocation for running processes and groups.

There is a long history of container building blocks in Unix systems. It's important for learners to remember that containers didn't come out of thin air, and Docker was by no means the first company to create containers. Docker's ingenuity was mainly to create an attractive, easy-to-use ecosystem for containers.

It's a lot like the relationship between Git and GitHub. Git was around long before GitHub, but GitHub created a set of tools and integration that made Git a lot easier to use.

Users should be familiar with "chroot" if they have used Linux. chroot is for filesystem isolation. chroot has been around since forever, and it gives the ability for a process to have a customized, usually restricted view of a filesystem. It does not give users the ability to combine together different filesystems or parts of a filesystem. To do that, you need OverlayFS or a similar tool.

The Jail command only exists in BSD, and interestingly a variation of a BSD jail is used in Apple's iOS. Each app runs in a "jail", which is why installing unauthorized Apps is called "jailbreaking." Typically, breaking out of a jail requires some sort of exploit, it's a bit like gaining root access to a computer. Jails are good for running code which is not completely trusted, but it's not as secure as running inside a VM.

Linux namespaces are designed for process isolation. This means that we can develop groups of processes, each with their own PIDs, names, etc., and the processes in one group cannot see or communicate with those in another group. It is called namespaces because we can have duplicate PIDs in different groups, and that's OK, similar to namespaces in programming languages like C++ and Java.

Cgroups allow us to do resource isolation. It's a bit like "nice" on steroids, allowing us to be able to much more strictly partition resources such as memory, CPU, I/O devices, much in the same way that we would partition for a VM. However, Cgroups are NOT VMs, just merely a way to control existing process's resources.

Linux Containers

2008: Linux Containers (LXC): Combines together:	2013: LXC + Virtualization extensions	2014: LXC + OverlayFS or UnionFS
<ul style="list-style-type: none">• chroot: (filesystem isolation) +• Namespaces (process isolation) +• Cgroups (resource isolation)	<ul style="list-style-type: none">• Allows hardware support for virtualized namespaces.	<ul style="list-style-type: none">• Allows for multiple filesystems to be combined to one virtualized filesystem.• Allows more practical overlay of containers than chroot.

Linux Containers, or LXC, are really the combination of chroot, namespaces, and Cgroups. These three things together make up Linux Containers, which are the Linux replacement for the BSD jail, which does not work on Linux. As LXC is built into the Linux kernel, there is no need for any customization to make it work in modern Linux.

Since Docker containers were originally built on top of LXC, LXC is a good basis for understanding how Docker works. Since version 1.1 of Docker, all LXC dependencies have been removed, though kernel namespaces and Cgroups are still requirements for Docker.

LXC originally did not use hardware virtualization support at all, and is still not required, strictly speaking for simply containerizing. However, the VT-x extensions can be useful for process isolation and so are used.

Docker as we know it today could not exist without OverlayFS or UnionFS. That's because the way we layer containers filesystems on top of one another is not possible with chroot alone. OverlayFS or UnionFS allows us to put many filesystems on top for each other to make a virtual filesystem for the container.

Docker

2013: Docker released

- Docker combines the following:
 - Namespaces
 - Cgroups
 - LXC
 - OverlayFS

Docker created a new container format



- Docker Container format is now the standard container format.

Developer Tools

- Docker has developed developer toolchain to use its container format.
-

Docker has become the de-facto standard component for containers. Interestingly, prior to Docker's release, container management systems like Mesos and Kubernetes used other container formats. However, given Docker's popularity, most of the toolchains have converted to Docker.

It's important to note that Docker relies on standard Linux kernel features in order to perform what it does, the same features that are done by LXC.

Docker's main contribution has been its container format, which is now the standard. Docker containers are also storable via Docker Hub or other Docker repositories.

Docker's developer toolchain is its other main contribution. Docker's tools allow users to very easily create new containers using its Dockerfile format, which is intentionally similar to Makefiles.

Interactivity: Poll Question

Question: What does Docker rely on to perform what it does?

- A. A Mac platform
 - B. Standard Linux kernel features
 - C. Microsoft Hyper-V
 - D. JDX
-



Docker Overview

What is Docker?



- Docker is the current industry standard container format.
- Docker (the company) also makes developer tools:
 - Docker for Windows
 - Docker for Mac
 - Docker for Linux
- Docker also has server-side container frameworks for Linux and Windows.
- Windows versions run on top of Hyper-V.

Docker's name is almost synonymous with containers. While arguably Docker's contribution is not the largest or the most significant to the container ecosystem, it is the one that is closest to the developer -- as the developer relies on Docker tools to create container application images.

Docker uses the Host Kernel

Docker will use the host kernel on Linux.

What about Windows and Mac (Developer) Versions of Docker?

- Windows and Mac versions run a virtualized Linux kernel in a VM.
- Only the kernel runs in the VM. The container runs separately.
- In Windows, the VM is in Hyper-V (and thus requires Hyper-V to be installed).
- On Mac, the VM runs in Apple's Hypervisor Framework.

Windows Server Native Windows Kernels:

- Windows Server Applications can run in Windows Native Containers on the Windows Server.
 - No VMs involved. (Although you may want a VM for security reasons.)
-

It's important to understand how containers work. They generally use the host kernel. We generally say because in fact on non-Linux platforms the kernel is usually virtualized.

The situation is a little complicated when it comes to Windows because there are multiple incompatible Windows kernels for various flavors of Windows. Because of that, most Windows "Native" containers run in virtualized mode except in production.

Docker for Windows (Developer)



Docker for Windows has the following requirements:

- Windows 10 Professional 64 Bit (or Higher) (not Home, Not Home Pro)
- Virtualization Extensions Available in CPU and turned on in BIOS
- Hyper-V (optional component) installed
- Reasonable CPU / Memory requirements

Docker Toolbox:

- Windows users unable to run Docker for Windows can run in Docker Toolbox
 - Toolbox requires Oracle VirtualBox for Virtualization
-

Docker for Windows has fairly steep requirements.

Why does Docker require Hyper-V? Because all containers on Windows require virtualization. Even windows containers require virtualization because those containers require Windows Server kernels which are different from Windows 10 (client) kernels.

Hyper-V does not run on Windows Home versions (most consumer-grade windows machines). So, Docker for Windows will not run on those versions either. You need Windows Professional.

Hyper-V will require that Virtualization extensions be turned on in the BIOS. Typically, from the factory these are turned off on client machines for security reasons, as some exploits have targeted these instruction sets.

Docker Toolbox can be used together with Oracle VirtualBox to run Docker containers on older versions of windows such as 8.1 or 7 or on Home versions of Windows 10. However, it's a pretty painful developer experience, so it's not recommended.

Docker and Windows Server



Docker for Windows Server is designed for app deployment rather than development.

- Developers should use Docker for Windows instead
- Supports running Linux containers (via Hyper-V)
- And native Windows containers (using Hyper-V or native Windows containers)

Native Windows Containers

- Native Windows containers must use the same Windows Server kernel as the host
 - Both host and guest must be Windows Server to run native (not Windows 10)
 - Native Windows containers must use Windows Server kernel
 - Able to run native containers without virtualization
 - Minimal performance penalty
-

Native Windows containers are not all that common at this time, but Microsoft is pushing them because they realize that the Windows platform needs to have this capability to continue to be relevant as a server-side platform.

Docker on Windows Use Cases

	Windows 10 (Linux Kernel)	Windows 10 (Windows Server Kernel)	Windows Server (Virtualized Kernel)	Windows Server (Native Kernel)
Use Case	Developer	Developer	Deployment	Deployment
Container OS	Linux	Windows Server	Linux or Windows Server	Windows Server
Isolation from Host	Yes (via Hyper-V)	Yes (via Hyper-V)	Yes (via Hyper-V)	Not Virtualized
Startup Delay	500ms - 1s	500ms - 1s	500ms - 1s	No Delay

This slide shows the relative use cases of various use cases on Windows. Docker realizes that Windows is its most common platform for development, even for applications that will not be deployed on Windows. Because of this, the Windows platform has the richest number of options. Windows users need to choose between developing "native" Windows server containers or Linux containers. The advantage of the native containers is that there is minimal delay in starting a native container, similar to what one would expect on Linux platforms.

Docker on Mac (Developer)



Docker on Mac is only supported as a development platform.

- There are no "native" Mac containers as there are on Linux and Windows.

Docker for Mac only uses Linux containers.

- Windows containers can be run entirely inside a VM.
- Linux containers use a VM for the Linux kernel only.

Docker for Mac uses Apple's VM Hypervisor.

Apple is an extremely popular platform for development, so it comes as no surprise that Docker has versions for this platform. As Apple has no real presence in the data center, it is strictly for development.

While it's often thought that Apple is built on top of Unix, it has chosen BSD Unix as its base, which is not compatible with the Linux Kernel. So, like Windows, Mac users must run in a VM, though unlike Windows, all modern Mac systems are capable of running virtualization without any special actions.

There is no way to run "native" Windows containers on Mac, except for in a VM. This isn't a common use case.

Interactivity: Poll Question

Question: Why does Docker require Hyper-V?

- A. The Windows containers do not need to be virtualized.
 - B. The Docker Hub is private.
 - C. Docker for Mac uses Windows containers.
 - D. All containers on Windows require virtualization.
-



Docker Hub and Container Registries

What is Docker Hub?

- Docker Hub is a registry of containers.
- Think "GitHub" for containers.
- Thousands of containers are available.
- We rarely start from scratch.

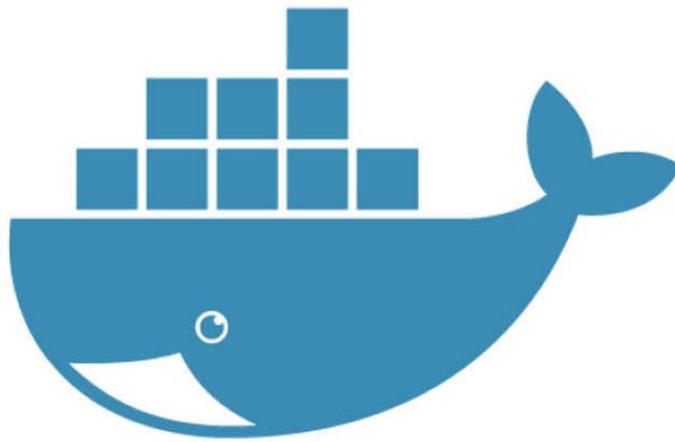


If Docker's container image format is its first great contribution, Docker Hub and similar container registries are the second. The fact that users can share and get container images from Docker Hub is a huge catalyst to containers, as it allows code to be shared across multiple teams. More importantly, it means there's no reason to re-invent the wheel. In fact, Docker Hub is designed to minimize re-invention of any kind.

We will see later how Docker's build process allows us to leverage other people's work.

Container Registries

- Docker Hub is public.
- Your company data is **NOT**.
- You may want to have your own container registry.
- But we may still use Docker Hub as well.



Corporate environments invariably have Docker Hub blocked, much like Maven repository coordinate and GitHub itself. Because of this, companies are able to use private container registries. In fact, this is encouraged in Docker and rarely will someone attempt to put something out on public Docker Hub unless there is a very good reason to do so.

This doesn't mean that you can't use Docker Hub. If you want to start with, say, a basic Linux image, using something from Docker Hub as a starting point makes a lot of sense.

Docker Trusted Registry

- Just as with GitHub, many companies want “their own Github”
 - This is provided as software product, and it is called “Docker Trusted Registry” (DTR)
 - It is created and provided by the Docker company.
 - It is a professional, enterprise-grade solution.
 - You install it behind your firewall in order to:
 - Securely store Docker images in your applications.
 - Securely manage Docker images in your applications.
-

DTR has the following desired properties:

- Highly available
- Efficiency in performance
- Built-in access control
- Security scanning
- Image signing
- There are multiple editions described here <https://forums.docker.com/t/dtr-pricing-question/3258>

Docker Pull

Typing `docker pull` will download a container.

In the lab, we will use a container called `alpine`:

- Ultra-lightweight Linux
- 4MB in Size (tiny)

```
1 | docker pull alpine
```

Anyone who has used GitHub and Git will find the Docker syntax easy to remember. Instead of using a copy of the source code repository, Docker will store the binary container image (usually a hash) in a local place on the user's hard drive.

Docker Commands

Docker LS (list)

Typing docker ls will **list** your containers.

It will only show **running** containers.

Stopped containers say `docker ls -a'

	\$ docker ls					
1						
2						
3	REPOSITORY	TAG	IMAGE ID	CREATED	SIZE	
4	hello-world	latest	e38bc07ac18e	5 weeks ago	1.85kB	
5	alpine	latest	3fd9065eaf02	4 months ago	4.15MB	

Running "docker ls" will show all the containers that you currently have. Note that you may see containers show up that aren't in fact directly being used. That's because the docker pull will recursively bring down containers and store them.

Note that every time you start a container, you will continue to see that container in docker ls. Do not be alarmed, because only the differences between the original image and the stopped container will be stored.

Docker Run

We can run a Docker container with `docker run`

Example: running alpine

```
1 | docker container run alpine /bin/ash
```

This will run alpine and start a shell

Alpine doesn't have "bash" -- too big!

Docker containers can be started with "run", this is because it's better to look at running a container as more like an app than a VM.

Alpine is extremely popular as mini-sized Linux to start containers. While running with a starter like ubuntu certainly gives a lot more power plus the familiarity with ordinary tools like "bash", alpine is great when all we really need is just enough of an OS to run our application.

Running a Shell

We don't *typically* ssh to our container.

We simply run a shell in the container.

```
1 | docker container run -it --rm alpine /bin/ash
```

What does this mean?

- `-i`: interactive mode
 - `-t`: terminal mode
 - `--rm`: remove container after we are done
 - `/bin/ash`: bash is big and needs to be installed. ash (almquist shell) is small. We also have old-school sh
-

Many users ask how to "ssh" to the container. This isn't typically how we accomplish this, instead, what we do is to attach our terminal to the container.

Some containers we probably want to automatically stop when we are no longer using the container. This depends a lot on the use case. If the container is more of a background service, then we probably do not want to do this. However, if it is an interactive application then this is not a bad idea.

Lab: Pulling a Container

Overview:

Time: 30 Minutes

In this lab, you will:

- Perform multiple Docker functions.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Docker RM

What happens if we need to delete a container?

The docker rm command will do the trick for us.

```
1 | docker container --rm ls -l
```

Running "docker rm" is a good way to clean up and de-clutter our container registry. That said, one should not expect to save lots of space by running docker rm, because even larger container images are stored in the base repository, and deleting our derived images from these will not save an extraordinary amount of space.

Remember, Docker overlays changes from one filesystem to the next, so only the changes will be removed.

Removing a Container

`docker rm` does **NOT** delete the container **image**.

- The container image is from Docker Hub.
- It is stored locally in our local repo.

It deletes the container instance.

- The instance will contain an overlay.
- Any changes to the original image are in the overlay.
- This is why we need OverlayFS in Docker.



We differentiate between the container image and the container instance. The image is not going to be directly deleted this way, but the instance will be.

Knowing the difference between instance and image is extremely important.

Lab: Manipulating a Container

Overview:

Time: 30 Minutes

In this lab, you will:

- Manipulate our containers.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Docker Operations

Port Mapping

The host can forward the network to the container.

This is done by port mapping.

```
1 | docker container run -p HOSTPORT:CONTAINERPORT
```

Instead of doing all the work itself, the host can refer to the container. This is done by port mapping. The command above shows the port mapping from the command line.

Running a Web Server

We can use `nginx`, a web server

Why not `httpd` (apache)?

`nginx` is really **tiny**

- Perfect for a container

The `nginx` is a container name on Docker Hub



Say, you want to run a web server from a container? Then you have a choice of `nginx` or `httpd` (Apache). So, let's go for the smallest service that does it.

Mapping nginx

nginx needs access to port 80

But we don't want to map port 80 of the host

Let's use 8002

```
1 docker container run -p 8002:80 nginx
```

Here is the response:

```
1 Unable to find image 'nginx:latest' locally
2 latest: Pulling from library/nginx
```

We already saw that port mapping could forward to the container. In this case, however, we don't want to do this because we don't want to take up port 80. Therefore, we will forward to 8002, as shown on the command line on the slide above.

Running in Background

So far, our containers always get stopped when done.

How do we run it as a background (daemon) process?

```
1 | docker container run -p 8002:80 -d nginx
```

Often, you have a need to keep running the service. The usual default container behavior is to stop. Here we are changing this with the command 'run'.

Lab: Ports

Overview:

Time: 30 Minutes

In this lab, you will:

- Network with containers.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Container Logging



Docker logs command:

```
1 | docker logs <container name>
```

See the log interactively (tail -f)

```
1 | docker logs -f <container name>
```

Note that this will give us the output of PID 1.

Running without logging is often compared to flying blindfolded. The commands above allow us to analyze the logs. For example, the '-f' option allows us to see the logs interactively.

Application Logs

Our applications will likely have their own logs

log4j / apache / etc.

We can map a volume to the host using -v

```
1 | docker run -d -P -v ~/web-logs:/var/log/nginx nginx
```

This will map ~/web-logs in the host to /var/log/nginx in the container.

Each application has its own logging preference and the place to which it logs. In this case, we show how to map the preferred application's log directory to our desired output.

Is Docker Secure?

Running in the container is more secure than running on the host.

Namespaces isolate running apps from each other.

Cgroups isolate resources by container.

- Helps avoid one app exhausting all resources

Containers have a very light "attack surface"

- Fewer running processes
- Limited libraries
- Less things to attack

Docker daemon runs as root.

Adding users to the Docker group gives them access to root.

Don't give access to users that don't have sudo access.



Containers allow us to do many different things efficiently, but we have to take precautions so that other people can't use them. In today's world, insecure applications are a no-go. There is no 'security by obscurity' because of the constant bot's activity that monitors for possible holes. The best practice, therefore, is not to give unnecessary root privileges to Docker groups.

Kernel Exploits

- Docker containers share the host kernel
- Kernel exploits in the container can affect host
- Compromised code can be run on the host
- Kernel panics will affect **all** containers



As we've been saying all along, Docker containers share the host kernel. That is the whole reason, raison d'etre, of Docker. The reverse side of this is that the kernel panic will affect all Docker containers that are running on top of this kernel.

Resource Starvation

Cgroups can limit access to resources.

If not limited, the app can use it up.



All applications want resources. Sometimes, this desire may cause the system to hang. We need to limit the access to resources for each application, because if an app doesn't have limits, it could use up all the available resources. This is accomplished with Cgroups.

Docker CLI Tools

CLI

- Most work in Docker is done on the CLI.
- We have already seen this.
- We use the docker command to execute most things.



Docker is a tool for scalability. It should, therefore, be scriptable and not depend on user's clicks. The 'docker' command is the basis for this scripting ability.

Interactivity: Poll Question

Question: What command is usually used to execute Docker commands on the CLI?

- A. docker
 - B. docker execute
 - C. CLI docker
 - D. CLI execute
-



Permissions

Docker service runs as root.

Regular users need to run every command as "sudo"

- Very **annoying** for development.

Can add user to docker group:

- **BEWARE!** Gives the users root access!



A potentially dangerous security practice which nevertheless simplifies a developer's life, is giving 'root' access to the docker user. Keep in mind though that if you add a user to the 'docker' group, then, if the user is hacked, it is 'game over' for docker.

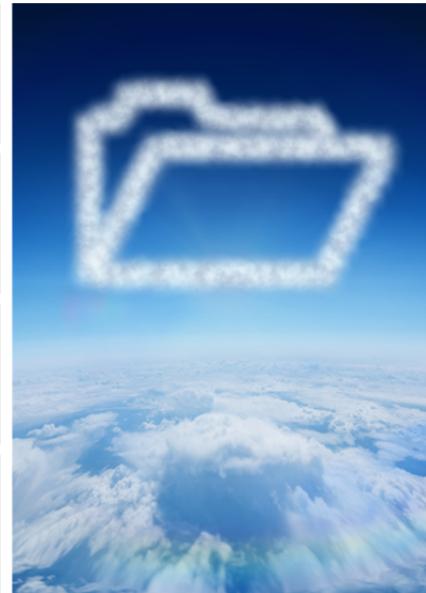
What is a Dockerfile?

Remember that containers are **layered**.

The base image usually comes from Docker Hub.

Our image has changes to that.

These changes are specified in a Dockerfile.



Dockerfile is just the abstraction that allows building the Docker images by adding changes to the basic image.

FROM Statement

Dockerfiles always come from **somewhere**.

We don't start from scratch.

Usually some Linux, like Alpine.

Here, we will use Ubuntu:

```
1 # The base image
2 FROM ubuntu:latest
```

The first statement in the Dockerfiles is “FROM”. It specifies the base image for this container.

Changes to Image

We normally make changes to the image.

Example commands:

- **RUN**: Executes a command on the image.
- **COPY**: Copies a file to the image.
- **CMD**: Executes our app.
- **HEALTHCHECK**: Gives us a health check.



After the Docker container specification starts its life from the base image, the allowed operations are run, copy, cmd, and healthcheck.

Networking

- Typically, we expose networking.
- Here, we expose 5000.

```
1 | EXPOSE 5000
```

In the “Networking” specification, we need to expose the network we’ll be dealing with. This is accomplished with the “EXPOSE” command.

Building

- We use the `docker build` command to build.
- We have to have a Dockerfile in the root of the directory.

Example:

```
1 | cd myfirstapp
2 | docker build -t myfirstapp
```

Watch the output -- it will be long!

Now building a docker image is easy. Assuming that there is a Dockerfile in the root of our directory, the two lines above accomplish building a Docker container.

Lab: Dockerfile

Overview:

Time: 30 Minutes

In this lab, you will:

- Create a Python app using flask, and deploy it using Docker.

**Instructions:**

- Open the Student Lab Manual and follow the steps to perform the lab.
-

Please find the lab instructions in the file student lab manual.

Health Checks

Container Health

- How do we know if a container is healthy?
- Docker doesn't interfere with the container
- "Agnostic Design"
- Health Checks should be specified in the Dockerfile.



Since Docker just allows the containers to run, it cannot ask them about their health. You should instead build the health checks into the container itself. This, too, is accomplished through the Dockerfile.

Discussion: DTR

Questions:

- What is DTR?
- What advantages does DTR provide?
- What is Dockerfile, what purpose does it serve?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Health Check Example

- The following is a line in a DockerFile.
- This is a health check.
- We will use "curl" to check a webservice.

```
1 | HEALTHCHECK CMD curl --fail http://localhost:3000/ || exit 1
```

Note that the health check returns 1 (bad) if failed.

Here is an example of health check in a Dockerfile. It checks whether the localhost on port 3000 responds correctly. If it does not, the health check fails.

Anatomy of a Dockerfile – FROM and ENV

Dockerfile breakdown: FROM and ENV

```
FROM docker.optum.com/optum_et/alpine:3.3
ENV JAVA_HOME /usr/lib/jvm/java-1.8-openjdk

ENV PATH $PATH:$JAVA_HOME/bin
ENV JAVA_VERSION 8u162
ENV JAVA_ALPINE_VERSION 8.92.14-r0
```

The Dockerfile must start with a “FROM” instruction. The FROM instruction specifies the base image from which you are building.

Environment variables (declared with the ENV statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile.

Anatomy of a Dockerfile – RUN

Dockerfile breakdown: RUN

```
RUN apk add --no-cache openjdk8="$JAVA_ALPINE_VERSION"

EXPOSE 8080 8443

RUN mkdir -p /opt/optum/
RUN mkdir -p /logs
RUN touch /logs/output.log

USER root
RUN chown -R 1001:1001 /opt/optum/
RUN chown -R 1001:1001 /logs
USER 1001

CMD ["java", "-XX:MaxRAM=768m", "-Xms40M", "-Xmx512M", "-Xss228k",
"-XX:MaxMetaspaceSize=256m", "-XX:MinHeapFreeRatio=20", "-
XX:MaxHeapFreeRatio=40", "-XX:CompressedClassSpaceSize=64m", "-
XX:+UseSerialGC", "-XX:GCTimeRatio=4", "-
XX:AdaptiveSizePolicyWeight=90", "-jar", "/opt/optum/app.jar"]
```

The RUN instruction has 2 forms:

- RUN <command> (*shell* form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- RUN ["executable", "param1", "param2"] (*exec* form)

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

Dockerfile Best Practices

1. Use .dockerignore file
 2. Make your container immutable and ephemeral
 3. Minimize the number of layers and consolidate instructions
 4. Do not install unused packages
 5. Sort multi-line arguments
 6. Build every time
-

1. Docker allows you to ignore some directories and files in build, and that improves build performance.
2. It should be easy to create a new container with minimal setup.
3. Each instruction in the Dockerfile translates into another layer in the image. Therefore, if you consolidate instructions you will minimize the number of layers. This leads to better performance.
4. Things will be simpler if you do not install unused packages.
5. Sorting multi-line arguments is simple convenience which helps avoid duplication. Think about developers.
6. Building every time makes your containers more reliable.

Discussion: Docker

Questions:

- What is the purpose of Docker?
- What use cases can you name?
- How do you implement Docker health check?
- What are the best practices for Docker security?



Instructions: Enter your answers into Chat (All Participants).

Take a moment to think about each question and write down your answers.

Module Summary

Now that you have completed this module, you should be able to:

- Gain a general overview of containers.
 - Learn about Docker.
 - Learn how to operate applications with Docker.
 - Describe Docker use cases.
 - Gain familiarity with CLI Tools.
 - Describe health checks of Docker.
 - Describe the anatomy of a Dockerfile.
-

Further Readings

The Twelve Factor App

- <https://12factor.net/>
-