

**WA2917 Booz Allen Hamilton  
Tech Excellence Cloud  
Engineering Program - Phase  
1**



The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: [getinfousa@webagesolutions.com](mailto:getinfousa@webagesolutions.com)

Canada: 1-877-812-8887 toll free, email: [getinfo@webagesolutions.com](mailto:getinfo@webagesolutions.com)

Copyright © 2021 Booz Allen Hamilton.

This publication is protected by the copyright laws of Canada, United States and any other country where this book is sold. Unauthorized use of this material, including but not limited to, reproduction of the whole or part of the content, re-sale or transmission through fax, photocopy or e-mail is prohibited. To obtain authorization for any such activities, please write to:

Web Age Solutions Inc.  
821A Bloor Street West  
Toronto  
Ontario, M6G 1M1

## Table of Contents

Chapter 1 - DevOps Fundamentals.....	11
1.1 Why DevOps?.....	12
1.2 What is DevOps?.....	13
1.3 What is DevOps?.....	14
1.4 What is DevOps?.....	15
1.5 Collaborative, Matrixed and Cross-Functional Teams.....	16
1.6 Key Components of Successful DevOps Teams.....	17
1.7 DevOps-ification.....	19
1.8 DevOps Vocabulary.....	21
1.9 DevOps Goals.....	23
1.10 Not DevOps - Crush Buzzwords.....	25
1.11 Driving Business Outcomes with DevOps.....	27
1.12 Technology-Enabled Business.....	29
1.13 DevOps Key Enabler for Digital Transformation.....	31
1.14 Core Values and Mission.....	32
1.15 Core Values - Culture.....	34
1.16 Core Values - Automation.....	35
1.17 Core Values - Measurement.....	37
1.18 Core Values - Sharing.....	39
1.19 Communication.....	40
1.20 Collaboration.....	41
1.21 Value Stream Mapping.....	42
1.22 Behavioral Patterns for Success.....	43
1.23 DevOps Org Structures.....	44
1.24 DevOps Team - Separate.....	45
1.25 DevOps Merged Organization.....	46
1.26 DevOps Overlapped Organization.....	47
1.27 Organizational Structure Leadership.....	48
1.28 What Does Continuous Delivery Mean?.....	49
1.29 Deployment Pipelines.....	50
1.30 Your Organization is Doing CD if .....	51
1.31 Pipelining for CD.....	52
1.32 Continuous Integration.....	53
1.33 CI Pipeline.....	54
1.34 CD & CI Methodologies.....	55
1.35 Key Tool Categories for CI/CD.....	56
1.36 Enterprise DevOps Playbook.....	57
1.37 Enterprise DevOps Playbook (Cont'd).....	58
1.38 Enterprise DevOps Playbook (Cont'd).....	59
1.39 Summary.....	60
Chapter 2 - Introduction to Git Flow.....	61
2.1 Why Use an SCM Workflow?.....	62
2.2 Why Use an SCM Workflow? (Cond.).....	63
2.3 What is Git Flow.....	64

2.4 Benefits.....	65
2.5 How Git Flow works?.....	66
2.6 How Git Flow works? (Contd.).....	69
2.7 What is Git Flow? (Contd.).....	72
2.8 How Git Flow works? (Contd.).....	74
2.9 How Git Flow works? (Contd.).....	77
2.10 Git Flow Extension.....	80
2.11 Initializing Git Flow.....	81
2.12 Features.....	82
2.13 Release.....	83
2.14 Hotfixes.....	84
2.15 Git Flow and Continuous Integration.....	85
2.16 Git Flow – Summary.....	86
2.17 Git Flow – Pros and Cons.....	87
2.18 Git Flow – When it Works Best?.....	88
2.19 Git Flow – When it Doesn’t Work?.....	89
2.20 Git Flow Alternatives.....	90
2.21 Trunk-based Development.....	91
2.22 Trunk-based Development (Contd.).....	92
2.23 Trunk-based Development – When it Works?.....	94
2.24 Trunk-based Development – When it Doesn’t Work?.....	95
2.25 GitHub Flow.....	96
2.26 GitHub Flow – Pros and Cons.....	97
2.27 GitLab Flow.....	98
2.28 GitLab Flow – Environment Branches.....	99
2.29 GitLab Flow – Release Branches.....	101
2.30 GitLab Flow – Release Branches (Contd.).....	102
2.31 GitLab Flow – Pros and Cons.....	103
2.32 Summary.....	104
Chapter 3 - Twelve-factor Applications [OPTIONAL].....	105
3.1 Twelve-factor Applications.....	106
3.2 Twelve Factors, Microservices, and App Modernization.....	107
3.3 The Twelve Factors.....	108
3.4 Categorizing the 12 Factors.....	109
3.5 12-Factor Microservice Codebase.....	110
3.6 12-Factor Microservice Dependencies.....	111
3.7 12-Factor Microservice Config.....	112
3.8 12-Factor Microservice Backing Services.....	113
3.9 12-Factor Microservice Build, Release, Run.....	114
3.10 12-Factor Microservice Processes.....	115
3.11 12-Factor Microservice Port Binding.....	116
3.12 12-Factor Microservice Concurrency.....	118
3.13 12-Factor Microservice Disposability.....	119
3.14 12-Factor Microservice Dev/Prod Parity.....	120
3.15 12-Factor Microservice Logs.....	121
3.16 12-Factor Microservice Admin Processes.....	122

3.17	Kubernetes and the Twelve Factors - 1 Codebase.....	123
3.18	Kubernetes and the Twelve Factors - 2 Dependencies.....	124
3.19	Kubernetes and the Twelve Factors - 3 Config.....	125
3.20	Kubernetes and the Twelve Factors - 4 Backing Services.....	126
3.21	Kubernetes and the Twelve Factors - 5 Build, Release, Run.....	127
3.22	Kubernetes and the Twelve Factors - 6 Processes.....	128
3.23	Kubernetes and the Twelve Factors - 7 Port Binding.....	129
3.24	Kubernetes and the Twelve Factors - 8 Concurrency.....	130
3.25	Kubernetes and the Twelve Factors - 9 Disposability.....	131
3.26	Kubernetes and the Twelve Factors - 10 Dev/Prod Parity.....	132
3.27	Kubernetes and the Twelve Factors - 11 Logs.....	133
3.28	Kubernetes and the Twelve Factors - 12 Admin Processes.....	134
3.29	Summary.....	135
Chapter 4 - Microservice Development.....		136
4.1	What are Microservices?.....	137
4.2	Microservices vs Classic SOA.....	138
4.3	Principles of Microservices Architecture Design.....	139
4.4	Domain-Driven Design.....	141
4.5	Domain-Driven Design - Benefits.....	142
4.6	Microservices and Domain-Driven Design.....	143
4.7	Designing for failure.....	144
4.8	Microservices Architecture – Pros.....	145
4.9	Microservices Architecture – Cons.....	146
4.10	Docker and Microservices.....	147
4.11	Microservice Deployment with Docker – Workflow.....	148
4.12	Writing Dockerfile.....	149
4.13	Kubernetes.....	150
4.14	What is OpenShift.....	151
4.15	OpenShift Architecture.....	152
4.16	Microservices and Various Applications.....	153
4.17	Web Applications.....	154
4.18	Web Applications – Reference Architecture.....	155
4.19	Web Applications – When to use?.....	158
4.20	Single Page Applications.....	159
4.21	Single Page Applications – Benefits.....	160
4.22	Traditional Enterprise Application Architecture.....	161
4.23	Sample Microservices Architecture.....	163
4.24	Serverless & Event-driven Microservice – AWS Lambda.....	164
4.25	Summary.....	165
Chapter 5 - Introduction to Gradle.....		166
5.1	What is Gradle.....	167
5.2	Why Groovy?.....	168
5.3	Build Script.....	169
5.4	Sample Build Script.....	170
5.5	Task Dependencies.....	171
5.6	Plugins.....	172

5.7 Dependency Management.....	173
5.8 Gradle Command-Line Arguments.....	174
5.9 Summary.....	175
Chapter 6 - Introduction to Spring Boot.....	176
6.1 What is Spring Boot?.....	177
6.2 Spring Framework.....	179
6.3 How is Spring Boot Related to Spring Framework?.....	181
6.4 Spring Boot 2.....	182
6.5 Spring Boot Main Features.....	183
6.6 Spring Boot on the PaaS.....	185
6.7 Understanding Java Annotations.....	186
6.8 Spring MVC Annotations.....	187
6.9 Example of Spring MVC-based RESTful Web Service.....	188
6.10 Spring Booting Your RESTful Web Service.....	189
6.11 Spring Boot Skeletal Application Example.....	190
6.12 Converting a Spring Boot Application to a WAR File.....	192
6.13 Externalized Configuration.....	193
6.14 Starters.....	194
6.15 Maven - The 'pom.xml' File.....	195
6.16 Maven - The 'pom.xml' File.....	196
6.17 Spring Boot Maven Plugin.....	197
6.18 Gradle - The 'build.gradle' File.....	198
6.19 Spring Boot Maven Plugin.....	199
6.20 HOWTO: Create a Spring Boot Application.....	200
6.21 HOWTO: Create a Spring Boot Application.....	201
6.22 Spring Initializr.....	202
6.23 Spring Initializr.....	203
6.24 Summary.....	204
Chapter 7 - Spring REST Services.....	205
7.1 Many Flavors of Services.....	206
7.2 Understanding REST.....	207
7.3 RESTful Services.....	209
7.4 REST Resource Examples.....	210
7.5 @RestController Annotation.....	211
7.6 Implementing JAX-RS Services and Spring.....	212
7.7 JAX-RS Annotations.....	214
7.8 Java Clients Using RestTemplate.....	216
7.9 RestTemplate Methods.....	217
7.10 Summary.....	218
Chapter 8 - Introduction to Continuous Integration, Continuous Delivery and Jenkins-CI	
.....	219
8.1 Foundation of Agile AppDev.....	220
8.2 XP Flow.....	221
8.3 Extreme Programming.....	222
8.4 Agile Development.....	223
8.5 What is Continuous Integration.....	224

8.6	What is Continuous Integration (cont'd).....	225
8.7	What is Continuous Integration (cont'd).....	226
8.8	What is Continuous Integration (cont'd).....	227
8.9	Typical Setup for Continuous Integration.....	228
8.10	Setup Notes for Continuous Integration.....	229
8.11	CI with Artifact Management.....	230
8.12	What is Continuous Delivery?.....	231
8.13	Why Continuous Delivery?.....	232
8.14	DevOps and Continuous Delivery.....	235
8.15	Continuous Delivery Challenges.....	237
8.16	Continuous Delivery vs Continuous Deployment.....	239
8.17	Jenkins Continuous Integration.....	240
8.18	Jenkins Features.....	241
8.19	Running Jenkins.....	242
8.20	Summary.....	243
Chapter 9	- Installing and Running Jenkins.....	244
9.1	Downloading and Installing Jenkins.....	245
9.2	Running Jenkins as a Stand-Alone Application.....	246
9.3	Running Jenkins as a Stand-Alone Application (cont'd).....	247
9.4	Running Jenkins on an Application Server.....	248
9.5	The Jenkins Home Folder.....	249
9.6	Installing Jenkins as a Windows Service.....	250
9.7	Initial Configuration.....	251
9.8	Configuration Wizard.....	252
9.9	Configuration Wizard (cont'd).....	253
9.10	Configuration Wizard (cont'd).....	254
9.11	Configuring Tools.....	255
9.12	Configuring Tools - Best Practices.....	256
9.13	Logging in Jenkins.....	257
9.14	Custom Log Recorders.....	258
9.15	Summary.....	259
Chapter 10	- Job Types in Jenkins.....	260
10.1	Introduction.....	261
10.2	Different types of Jenkins Items.....	262
10.3	Different types of Jenkins Items (cont'd).....	263
10.4	Different types of Jenkins Items (cont'd).....	264
10.5	Configuring Source Code Management(SCM).....	265
10.6	Working with Subversion.....	266
10.7	Working with Subversion (cont'd).....	267
10.8	Working with Git.....	268
10.9	Storing Credentials.....	270
10.10	Service Accounts.....	271
10.11	Storing Credentials (cont'd).....	272
10.12	Build Triggers.....	273
10.13	Schedule Build Jobs.....	274
10.14	Polling the SCM.....	276

10.15 Polling vs Triggers.....	277
10.16 Maven Build Steps.....	278
10.17 Summary.....	279
Chapter 11 - Continuous Delivery and the Jenkins Pipeline.....	280
11.1 Continuous Delivery.....	281
11.2 Continuous Delivery (cont'd).....	284
11.3 DevOps and Continuous Delivery.....	286
11.4 Continuous Delivery Challenges.....	288
11.5 Continuous Delivery with Jenkins.....	290
11.6 The Pipeline Plugin.....	292
11.7 The Pipeline Plugin (cont'd).....	293
11.8 Defining a Pipeline.....	294
11.9 A Pipeline Example.....	295
11.10 Pipeline Example (cont'd).....	296
11.11 Parallel Execution.....	297
11.12 Creating a Pipeline.....	298
11.13 Invoking the Pipeline.....	299
11.14 Interacting with the Pipeline.....	300
11.15 Pipeline vs Traditional Jobs.....	301
11.16 Conclusion.....	302
Chapter 12 - Groovy DSL [OPTIONAL].....	303
12.1 What is Groovy.....	304
12.2 Groovy in Jenkins.....	305
12.3 Comments in Groovy.....	306
12.4 Data Types.....	307
12.5 Identifiers.....	309
12.6 Variables.....	310
12.7 def.....	311
12.8 String Interpolation.....	312
12.9 Operators.....	313
12.10 Ranges.....	314
12.11 Conditional Statements.....	315
12.12 Loops.....	317
12.13 Lists.....	319
12.14 Maps.....	321
12.15 Exception Handling.....	322
12.16 Methods.....	323
12.17 Closures.....	324
12.18 this Keyword.....	325
12.19 Classes.....	326
12.20 Static Methods.....	327
12.21 Inheritance.....	328
12.22 Abstract Classes.....	329
12.23 Interfaces.....	330
12.24 Generics.....	331
12.25 Jenkins Script Console.....	332



12.26	Extending with Shared Libraries.....	333
12.27	Directory Structure.....	334
12.28	Sample Groovy Code.....	335
12.29	Defining Shared Libraries.....	336
12.30	Using Shared Libraries.....	337
12.31	Same Shared Library Usage Code.....	338
12.32	Defining Global Variables.....	339
12.33	Summary.....	340
Chapter 13 -	Securing Jenkins.....	341
13.1	Jenkins Security - Overview.....	342
13.2	Jenkins Security.....	344
13.3	Authentication.....	346
13.4	Authorization.....	348
13.5	Confidentiality.....	350
13.6	Activating Security.....	352
13.7	Configure Authentication.....	354
13.8	Using Jenkins's Internal User Database.....	357
13.9	Creating Users.....	359
13.10	Authorization.....	361
13.11	Authorization.....	363
13.12	Matrix-Based Security.....	365
13.13	Note – Create the Administrative User.....	367
13.14	Project-based Matrix Authorization.....	369
13.15	Project-Based Authentication.....	371
13.16	Role Based Access Control.....	373
13.17	Conclusion.....	374
Chapter 14 -	Jenkins Plugins.....	375
14.1	Introduction.....	376
14.2	Jenkins Plugins - SCM.....	377
14.3	Jenkins Plugins – Build and Test.....	378
14.4	Jenkins Plugins – Analyzers.....	379
14.5	Jenkins for Teams.....	380
14.6	Installing Jenkins Plugins.....	381
14.7	Summary.....	385
Chapter 15 -	Distributed Builds with Jenkins.....	386
15.1	Distributed Builds - Overview.....	387
15.2	Distributed Builds – How?.....	388
15.3	Agent Machines.....	389
15.4	Configure Jenkins Master.....	390
15.5	Configure Projects.....	391
15.6	Conclusion.....	392
Chapter 16 -	Scripting.....	393
16.1	Introduction to Scripting.....	394
16.2	Benefits of Scripting.....	395
16.3	The Jenkins Script Console.....	396
16.4	Calling Scripts using cURL.....	397

16.5 Sample Scripts.....	398
16.6 Calling Groovy Scripts from a Jenkins Job.....	400
16.7 Jenkins API.....	401
16.8 Summary.....	402
Chapter 17 - Best Practices for Jenkins.....	403
17.1 Best Practices - Secure Jenkins.....	404
17.2 Best Practices - Users.....	405
17.3 Best Practices - Backups.....	406
17.4 Best Practices - Reproducible Builds.....	407
17.5 Best Practices - Testing and Reports.....	408
17.6 Best Practices - Large Systems.....	409
17.7 Best Practices - Distributed Jenkins.....	410
17.8 Best Practices - Summary.....	411

## Chapter 1 - DevOps Fundamentals

---



## 1.1 Why DevOps?

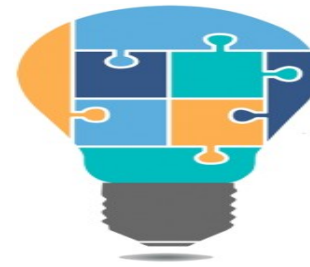
- Rapidly deliver reliable, scalable micro and macro services that are critical for the business time-to-market and value
- Bring the Development and Operations teams
- Technical benefits:
  - ◇ Continuous delivery
  - ◇ Reduction in component complexity
  - ◇ Agility in problem resolution
- Business benefits:
  - ◇ Agility in delivery of business benefit
  - ◇ Increased stability
  - ◇ More focus on competitive advantage, rather than simply “keeping the lights on”





## 1.2 What is DevOps?

- Combination of people, process, tools, vendors,
- All the **–ity** words, agility, reliability, scalability, reproducibility, ...
- Focus on
  - ◇ Knowledge sharing
  - ◇ Automation
  - ◇ Collaborative culture
  - ◇ Time to market & value



partners  
sustainability,



### 1.3 What is DevOps?

- DevOps is an organizational culture for application development and maintenance success. It encompasses technology and process
- DevOps encourages transparency and candid communication between software developers and IT operations and administration to increase the agility with which applications are delivered
- By automating, monitoring, and measuring the production of code through develop, release, test, and maintenance the process becomes more sustainable, scalable, and repeatable
- Optimally all manual tasks are eliminated, reducing the potential for human error and making delivery and roll-back less error prone



## 1.4 What is DevOps?

- Creating common goals for development and operations of getting apps deployed as quickly, safely, and efficiently as possible
- Developers building in switches, dials, and flags that enable operations to disable features or dial them down, when traffic spikes or problems occur
- Operations teams working closely with developers to facilitate agile and on-going deployments – no over the fence handoffs
- Teams building reliable and repeatable processes to support business, as well as, understanding their purpose in the organizational value chain



## 1.5 Collaborative, Matrixed and Cross-Functional Teams

- Start by understanding the interactions between different teams in the organization
- Are builds “tossed over the wall” from Development to Testing, and then from Testing to Operations?
- What is the visibility between the various teams?
- Do teams “wash hands” after a hand-off or stay involved?
- How agile and focused are communication for releases, builds, and bugs?







## 1.6 Key Components of Successful DevOps Teams

- “A3 & C”:
  - ◇ Shared **A**ccountability
  - ◇ **A**utomation of almost anything
  - ◇ **A**nalytics
  - ◇ Culture of **C**ollaboration
- Other keys areas include leadership, consistent messaging, and standards



## 1.7 DevOps-ification

- When deciding on application of DevOps, ask these four questions
  - ◇ What is the optimal or required business outcome?
  - ◇ How are current processes meeting requirements?
  - ◇ Why is this project or process a priority for change?
  - ◇ Who would benefit from a change in daily work?



## 1.8 DevOps Vocabulary

- **Silo** – The antithesis of DevOps; consider this the “dark side”
- **Release** – What does this term mean to different teams and how do you come to one definition across the organization
- **Transparency** – Requires cross team communication, consider this the “light side”
- **Automation** – Any task that could be done more than once should be automated
- **Measurement** – Anything worth doing needs to be measured, monitored, and analyzed for success or failure
- **Notification** – Tell people when it goes right, tell more people if it goes wrong
- **Leadership** – DevOps like any organizational initiative needs buy0in from the top and grass-roots support
- **Microservices** – An architectural style characterized by product focus, automated release, intelligent endpoints, and distributed governance and control



## 1.9 DevOps Goals

- **Shift Left** - DevOps goes hand-in-hand with Agile software development methodologies. This places emphasis on rapid identification of defects.
- **Competitive Advantage** - Faster
- **Technology Enabled Business** - Time to market and value
- **Agility** - Any developer or server environment should be able to be reloaded in a short amount of time, e.g. 4 hours from request
- **Happy people** - Effective DevOps implementations incur less daily stress and remove constant fire drills. This converts to better customer experience, both internally and externally



## 1.10 Not DevOps - Crush Buzzwords

- Automating infrastructure
- Building out CI/CD pipelines
- Writing scripts
- Create new silo called DevOps
- Doing anything on cloud





## **1.11 Driving Business Outcomes with DevOps**

- Business teams are highly motivated by time to market (agility)
- Less churn means the right frequency for releases and less wasted time
- Business teams see components early in development to reduce rework
- Efficiency in Digital Data Supply Chain



## 1.12 Technology-Enabled Business

- Data as an asset through the entire organizational digital data supply chain and enabled by analytics
- Quality improvement in product or services due to more rapid iterations, increased accuracy of models, and faster feedback loops
- Competitive advantage through time to market and value
- Improved customer satisfaction by enabling the ability to customize, improved quality, and agile delivery of new products the customers desire



## 1.13 DevOps Key Enabler for Digital Transformation







## 1.14 Core Values and Mission

- How does the team align to the core values and mission of the organization?
- Do all team members understand how their roles and responsibilities support the organization?



**Culture**  
**Automation**  
**Measurement**  
**Sharing**





## 1.15 Core Values - Culture

- Culture is the “way of life” of the organization
- Culture is pervasive and enduring
- DevOps culture is expressed by
  - Understanding motivations of other groups
  - Trust, elimination of blame
  - Embracing failure as necessary for improvement
  - Focus on process and bottlenecks
  - Elimination of unplanned work
  - Emergence of dedicated cross-functional teams
  - Candor and transparency





## 1.16 Core Values - Automation

- Automation
  - ◇ Allows focus on value-adding activities
  - ◇ Reduces errors
  - ◇ Makes processes explicit (and versionable)
- Automation is not a goal – it is a means to an end
  - ◇ Wrong automation is worse than no automation
  - ◇ Automation is a tool to improve process performance and consistency
- Effective automation requires understanding of end-to-end processes





## 1.17 Core Values - Measurement

- “If you can’t measure it, you can’t improve it.” – Peter Drucker
- Measure all things – within reason
- Define key performance indicators (KPIs)
  - ◇ KPIs should never be viewed in isolation - always interpret in the context of other KPIs
  - ◇ KPIs for DevOps should measure performance across departments or functions
  - ◇ Define the business question first before defining the KPI





## 1.18 Core Values - Sharing

- Sharing closes feedback loop across functions, teams, departments
- Sharing enables continuous learning and improvement
- Consider knowledge-sharing culture of organization:
  - ◇ How knowledge is captured, disseminated and propagated
  - ◇ Is knowledge shared collaboratively or is maintained by individuals or within teams
  - ◇ Are repositories of knowledge formal and maintained, or ad-hoc and prone to decay over time





## 1.19 Communication

- Challenges to communication
  - ◇ Organizational silos, physical location
  - ◇ Differences in concerns and priorities
  - ◇ Formal mechanisms like change requests, stage gates, and service tickets
- Improve communication through
  - ◇ More frequent interactions
  - ◇ Appropriate tools (meetings, huddles, email, Skype, Snapchat, KIK, IM, Slack, ...)
  - ◇ Communication is less about formal process, and more about collaboration



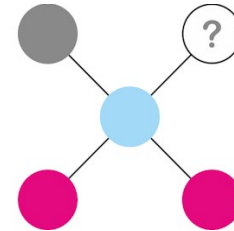
## 1.20 Collaboration

- Development-to-operations lifecycle is a single end-to-end process
- End-to-end process may be assembled from more focused processes
- Agile for software development
- ITIL for deployment and operations
- Support collaboration with unified tooling
- Jenkins or GoCD for continuous integration and delivery
- Infrastructure as code with Puppet, Chef, or similar
- Single knowledge repository across entire process
- Support collaboration through team structure
- One team unified by team building, candor, and transparency
- Or several closely connected teams with a common objective



## 1.21 Value Stream Mapping

- Method for analysis of current practices, design of new practices
- “Practices” are an orchestrated sequence of activities that meet needs of customers
- Based on process view of organizations, treating the organization as a system of interlinked processes
- Each process has inputs, transformations, and outputs
- Value Stream Mapping provides framework for formally designing future state of software delivery process







## 1.22 Behavioral Patterns for Success

- 7 Habits of Highly Effective DevOps Experts
- Think culture
  - Culture → People → Process → Tools
  - ◇ Automation – for anything that is difficult or repetitive
  - ◇ Shift left – quality through shared understanding
  - ◇ Transparency – communication of purpose and intent
  - ◇ Standardization – drive savings, sharing, efficiency
  - ◇ Compliance – design and include those teams upfront
  - ◇ Knowledge sharing – runbooks, releases, education
  - ◇ Collaboration

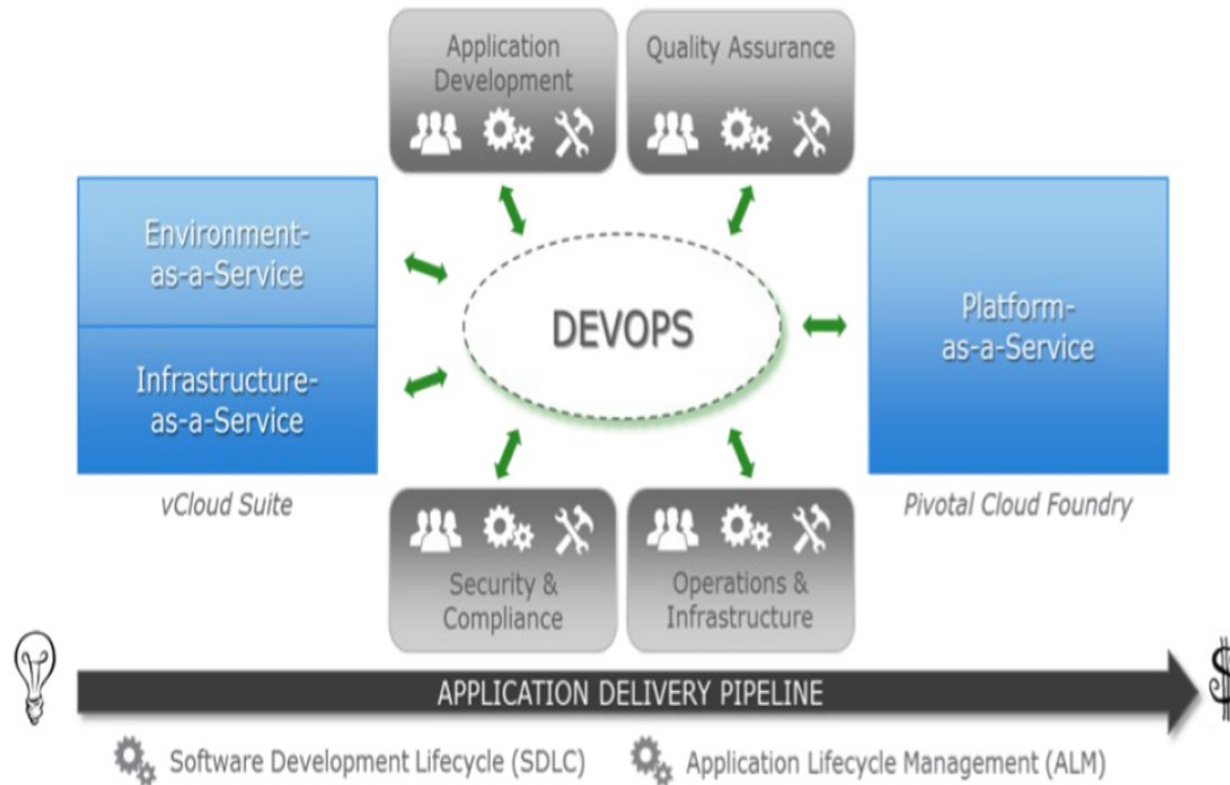


## 1.23 DevOps Org Structures

- DevOps team AKA Development → DevOps ← Operations
  - ◇ Do not create a DevOps Silo, but rather a meet in the middle organization through either a COE or an entire focused team with staff from both Development, Operations, and other relevant groups
  - ◇ Development merged with Operations
  - ◇ Development overlapped with Operations
- Organizational structure will determine team communications and goals. Breaking down organizational silos is key to DevOps implementation
- Proof of Concept or Pilot team should be used with a medium risk/priority project for a tactical win
- Do not start with mission critical application for first round, unless it is the only application you have, and then take a small to medium bite

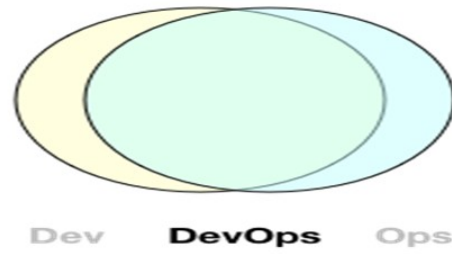


## 1.24 DevOps Team - Separate





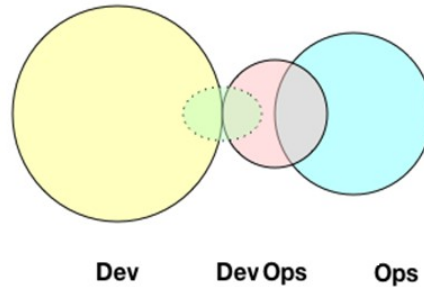
## 1.25 DevOps Merged Organization



- Fully embedded DevOps
  - ◇ Single-product offering focus
  - ◇ “We build it, we run it” culture
  - ◇ Each product has it's own DevOps team



## 1.26 DevOps Overlapped Organization



- For mature orgs with deep focus on operational excellence
- Development meets specified operational criteria
- Development focused on new features, not production ops
- SRE team engineers operating environment, operational criteria



## 1.27 Organizational Structure Leadership

- Key considerations
  - ◇ Single Executive or Single Group Leadership
  - ◇ Business case funding for implementation
  - ◇ Priorities – Top n projects, applications, products
  - ◇ Steering committee sets priority
  - ◇ Clear communication on business value chain
  - ◇ Inclusion of key stakeholders to get every brain in the game





## 1.28 What Does Continuous Delivery Mean?

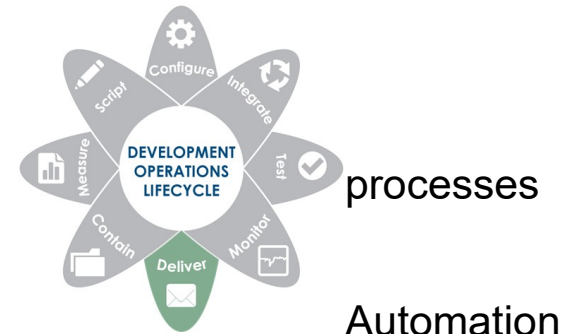
- We'll use the definition from Martin Fowler, a DevOps master craftsperson
  - ◇ Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time
- Core idea of CD is creation of a repeatable, reliable and agile SDLC process from ideation to client
- Goals are sustainability, agility, and repeatability





## 1.29 Deployment Pipelines

- Vehicle for continuous delivery
- Tasks separated by stages
- Implement various testing techniques and
- Support Shift-Left
- Task orchestration AKA Application Release







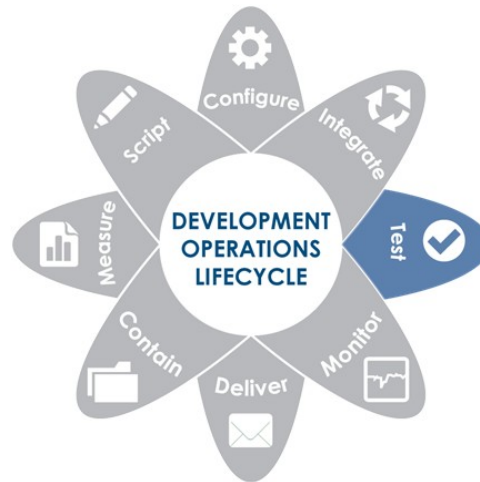
### **1.30 Your Organization is Doing CD if ...**

- Software is deployable throughout its life-cycle
- Team prioritizes keeping the app deployable over working on new features
- Automated feedback is available to anybody on the production readiness of any system at any time when anyone makes a change
- Deployments can be kicked off by a push of a button for any version of the software to any environment on request



## 1.31 Pipelining for CD

- Build
- Static code analysis
- Continuous Integration
- Release management
- Deployment
- Testing
  - ◇ Regression
  - ◇ Smoke
  - ◇ Performance
  - ◇ Business Validation & Verification





## 1.32 Continuous Integration

- Continuous Integration is a software development practice where members of a team integrate their work frequently
- Each person integrates daily – ensuring that the component is still build and deployment ready
- Each integration is verified by an automated build (including smoke testing) to detect integration errors at the earliest stage
- This approach should lead to a significantly reduced integration problems and allow the team to develop software with better agility and reliability – Shift Left



### 1.33 CI Pipeline

- Break build, release, and testing into stages including environmental complexity
- Each progressive stage provides additional detail and confidence on components and microservices
  - ◇ Build, compilation, dependency validation
  - ◇ Deployment to dev/test
  - ◇ Smoke testing
  - ◇ Progress through environments
  - ◇ ( dev, staging, QA, ...)
  - ◇ Production release





### **1.34 CD & CI Methodologies**

- Enable automation of the build early
- Front-load creation of a Dev server environment
- Include unit testing on the Dev server
- Quickly stand up production-like environment, e.g. clustered, integration points, roles, and permissions
- Agile methodology
- Ensure development, architecture, governance, security, operations, test/QA, and business or product team members are engaged



### 1.35 Key Tool Categories for CI/CD

- Monitoring
- Automation
- Configuration Management
- Build Management
- Version Control
- Notification
- Testing
- Knowledge Management
- Orchestration



Don't leave out your Database, Performance, Governance, Security and QA folks in these tool considerations



### 1.36 Enterprise DevOps Playbook

- When following DevOps at BAH, refer to Enterprise DevOps Playbook
- The DevOps Playbook recommends the principles, practices, and roles
- It also recommends various metrics which should be captured and checklists to assess your DevOps maturity level
- It recommends the following DevOps principles:
  - ◇ Treat operations as first-class citizens
  - ◇ Developers act as first responders to issues with the production system
  - ◇ Shorten the time between identification of a production issue and its repair
  - ◇ Shorten the time between code commit and code deploy
  - ◇ Minimize coordination to deploy releases
  - ◇ Stop and fix potential defects identified by continuous flow and monitoring
  - ◇ Enforce standardized processes to ensure predictable outcomes
  - ◇ Become a learning organization through continual feedback and action



### **1.37 Enterprise DevOps Playbook (Cont'd)**

- The DevOps Playbook recommends the following DevOps roles:
  - ◇ The DevOps architect
  - ◇ The DevOps engineer
  - ◇ The test automation engineer
  - ◇ The site reliability engineer
  - ◇ The software engineer





### **1.38 Enterprise DevOps Playbook (Cont'd)**

- The DevOps Playbook recommends the following practices:
  - ◇ Configuration management
  - ◇ Continuous integration
  - ◇ Automated testing
  - ◇ Infrastructure as Code
  - ◇ Continuous delivery
  - ◇ Continuous deployment
  - ◇ Continuous monitoring



### 1.39 Summary

- DevOps is an approach to delivering software value
- It sees the final product not as "software" but "software installed in production"
- View the entire delivery chain as a process, that can be studied and optimized
- Cultural Values and Technology combine to deliver value
- Tools and automation are required for DevOps, but aren't DevOps in themselves

## Chapter 2 - Introduction to Git Flow

---

### *Objectives*

Key objectives of this chapter

- Understanding Git Flow
- Understanding Features
- Understanding Releases
- Understanding Hotfixes
- Git Flow and Continuous Integration
- Git Flow Alternatives



## 2.1 Why Use an SCM Workflow?

- Using the right SCM workflow can help in the various scenarios, such as:
  - ◇ When you are trying to manage the integrity of a code base that involves numerous developers with conflicting release schedules.
  - ◇ When you get tired of wondering what best practices you should follow
  - ◇ When you don't have time to dig through the commit diffs of 500 commits to find out why a push from a specific person causes major unintended changes to your project.
  - ◇ When a feature that's not ready for release sneaks its way into production.
  - ◇ When you don't want to explain to stakeholders that you can't release feature X because feature Y has not yet cleared QA.



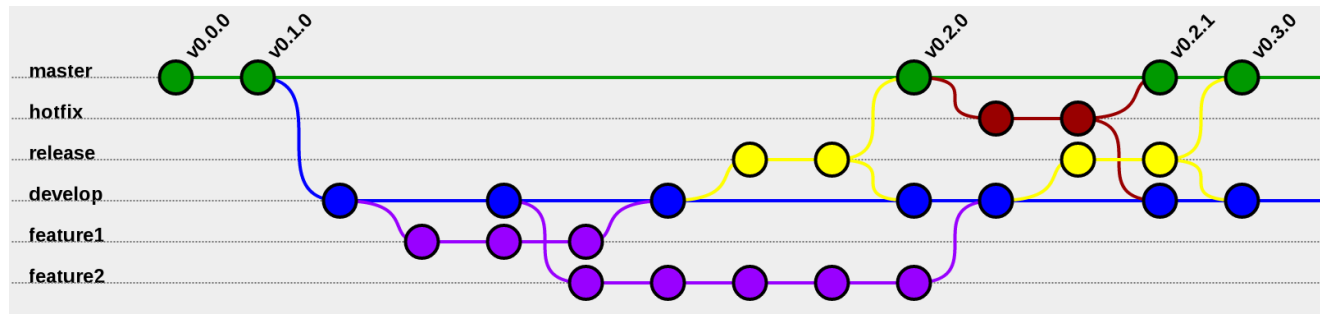
## 2.2 Why Use an SCM Workflow? (Cond.)

- Using the right SCM workflow can provide these benefits:
  - ◇ **Visibility** - If you need a way to understand who is working on which feature and how to pick up where they left off if they get pulled away to work on another project, visibility is very important.
  - ◇ **Flexibility** - If business expectations require you to be able to quickly switch between working on “urgent, must go live now” tasks and “show me what it would look like if...” tasks then your workflow should allow you to isolate features allowing them to go to production on different schedules.
  - ◇ **Collaboration** - Teams should strive to reduce maintenance overhead by standardizing the way you write and contribute code. You need a way to collaborate on code that helps enforce code standards and best practices while pushing each contributor to continually improve.
  - ◇ **Control** - Determining who should and should not have write access to your repository is a huge factor in deciding the right workflow.



## 2.3 What is Git Flow

- Git Flow is a branching model for Git.



- Created by Vincent Driessen and widely adopted by many organizations
- A set of git extensions to provide high-level repository operations.
- The model is scalable.
- Well suited to collaboration due to the isolation of code across branches
- Maps to the Continuous Integration (CI) and Continuous Delivery (CD) models of code being consistently buildable and deployable in the release and master branches



## 2.4 Benefits

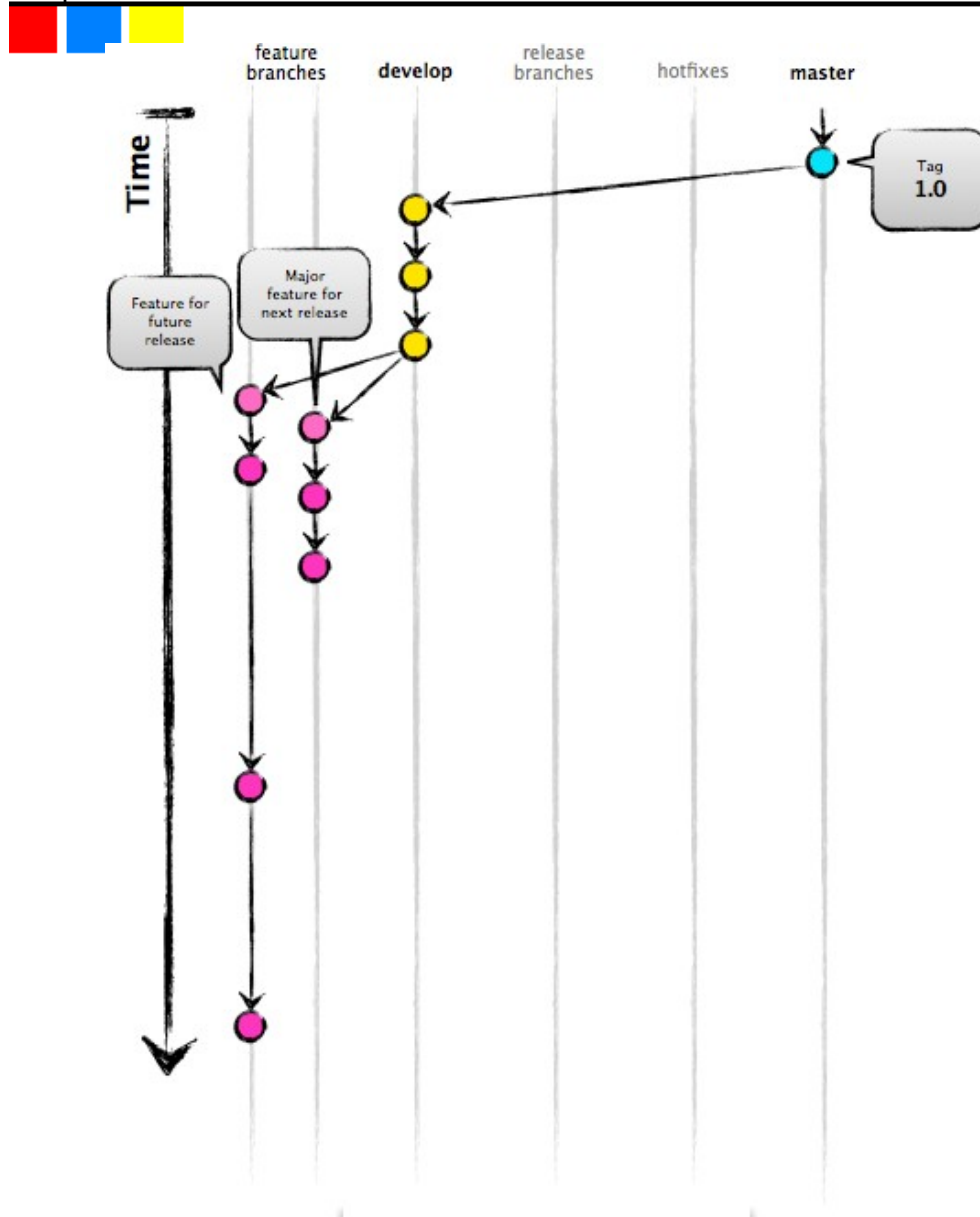
- Parallel development
  - ◇ Isolates new development work from finished work
  - ◇ Code is merged back into main body of code when developers are happy that code is ready for release.
- Collaboration
- Release Staging Area
  - ◇ New development work is merged back into the develop branch
  - ◇ 'Develop' branch is a staging area for all completed features that haven't yet been released.
- Support for hotfixes
  - ◇ Hotfix branches – branches made from a tagged release which can be used to make an emergency change.



## 2.5 How Git Flow works?

- New development are built in **feature** branches.



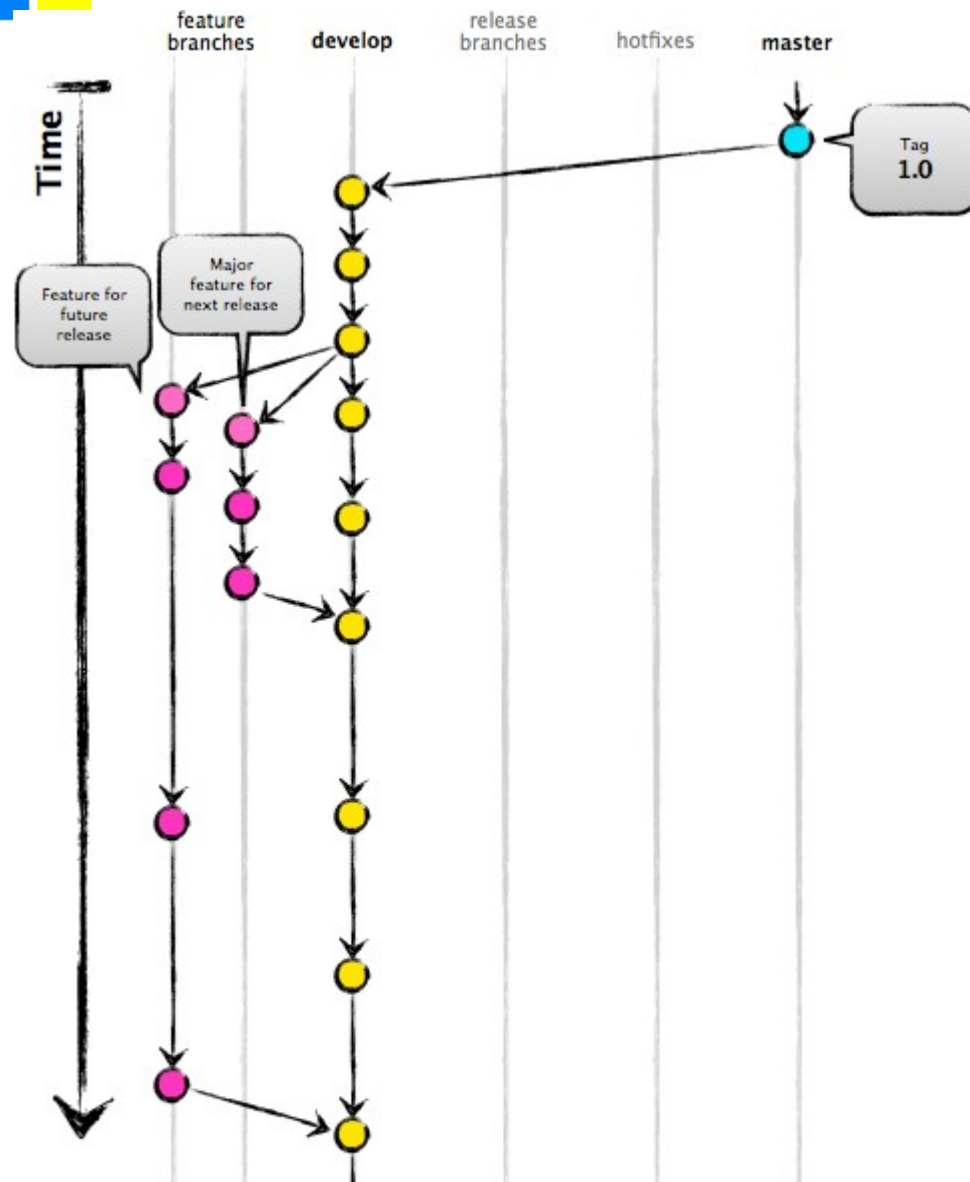






## 2.6 How Git Flow works? (Contd.)

- Feature branches are branched off of the **develop** branch.
- Finished features are merged back into the **develop** branch.



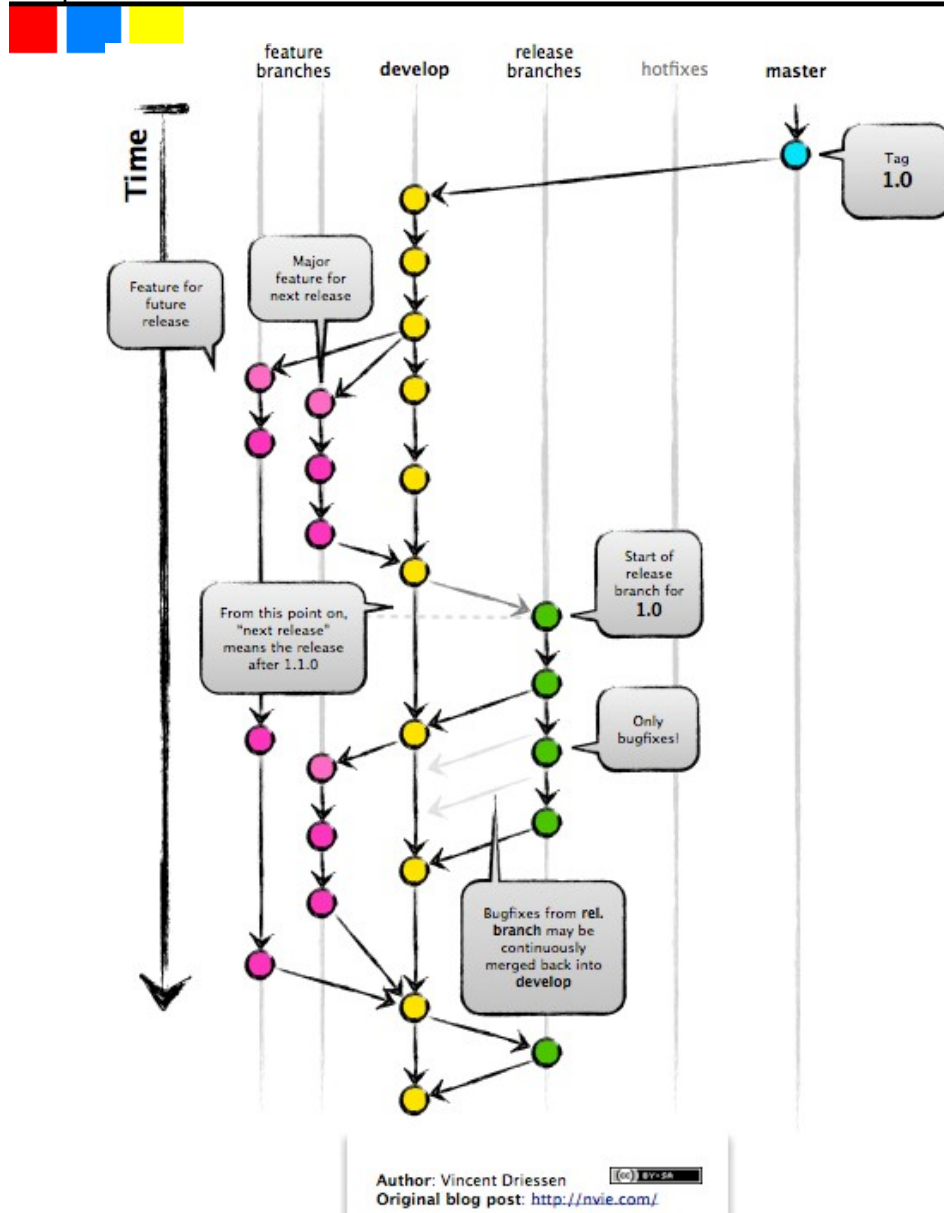




## 2.7 What is Git Flow? (Contd.)

- When features are completed, a **release** branch is created off of **develop** branch.

## Chapter 2 - Introduction to Git Flow





## **2.8 How Git Flow works? (Contd.)**

- The code in the release branch is deployed, tested, fixed, redeployed, and retested.
- Release branch is merged into master and into develop branch.



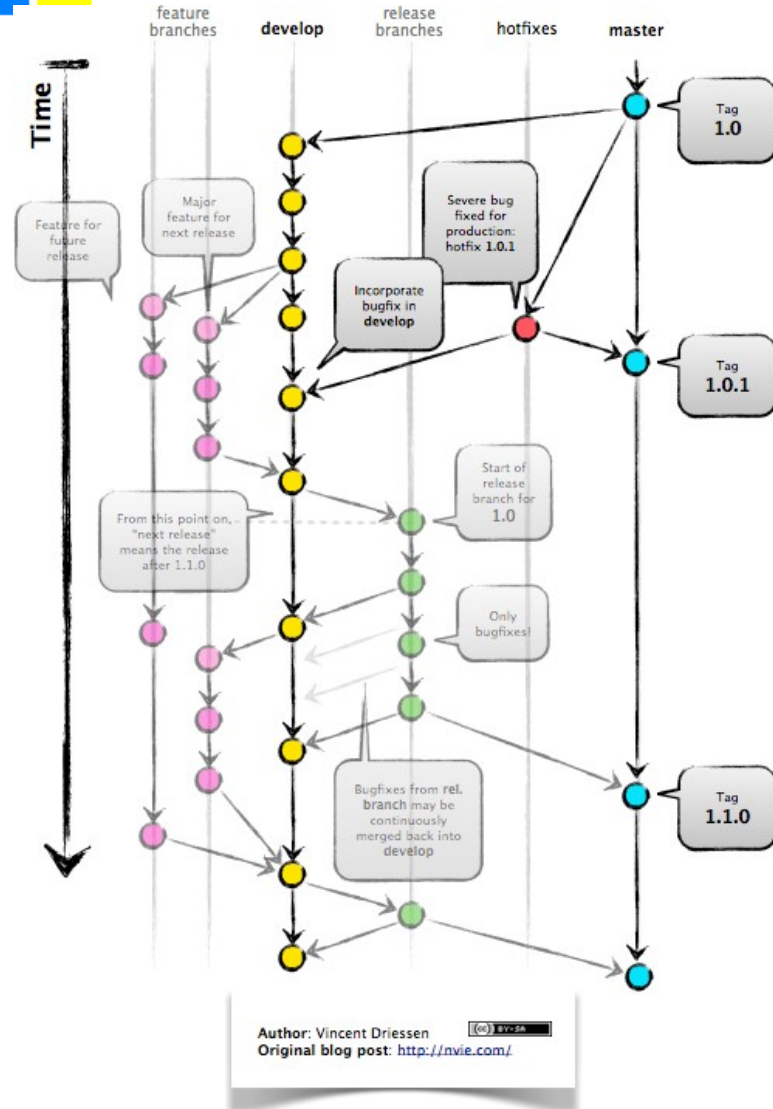






## 2.9 How Git Flow works? (Contd.)

- The master branch contains released code only.
- The only commits to master are merges from **release** branches and **hotfix** branches.
- Hotfix branches are branched directly from a tagged release in the master branch.
- Hotfixes, when finished, are merged back into both master and develop to ensure the hotfixes aren't accidentally lost when next release occurs.







## 2.10 Git Flow Extension

- All the above can be implemented using Git's inbuilt branching and merging
- To simplify a little, there's an “extension” to Git called “Git Flow”
- The extension is shipped by default with “Git for Windows”
- Available through package managers
  - ◇ e.g. 'sudo apt-get install git-flow'



## 2.11 Initializing Git Flow

- Initialized inside an existing git repository

```
git flow init
```

- Recommended to use the default values for branches



## 2.12 Features

- New development is organized in the form features
- Typically exist in developers repository only
- Start a new feature

```
git flow feature start FEATURE_NAME
```

- Finish up a feature
  - ◇ Merges feature into develop branch
  - ◇ Removes the **feature** branch (unless you use -k switch to keep it)
  - ◇ Switches back to **develop** branch

```
git flow feature finish FEATURE_NAME
```

- Published feature
  - ◇ Useful for pushing a feature to the remote repository so it can be used by other users

```
git flow feature publish FEATURE_NAME
```

```
git flow feature pull origin FEATURE_NAME
```





## 2.13 Release

- Finalized features are merged into a release branch
- Allow for minor bug fixes
- Start a release
  - ◇ **Release** branch is created from the **develop** branch

```
git flow release start RELEASE [BASE]
```

e.g.

```
git flow release start REL_1.0 develop
```

- Finish up a release
  - ◇ Merges the release branch back into 'master'
  - ◇ Tags the release with its name
  - ◇ Back-merges the release into 'develop' branched
  - ◇ Removes the release branch unless you use -k switch to keep it

```
git flow release finish RELEASE -m "tag"
```

- Publish a release
  - ◇ For collaboration, a release can be published to a remote repository

```
git flow release publish RELEASE
```



## 2.14 Hotfixes

- Changes required to fix an undesired state of a live production version
- Hotfixes can be branched off from the corresponding tag on the master branch that marks the production version
- Create a hotfix branch

```
git flow hotfix start VERSION [BASENAME]
```

- e.g.

```
git flow hotfix start 'REL_1.0.1' develop
```

- Finish a hotfix
- Finishing a hotfix gets merged back into **develop** and **master** branches.
- Master merge is tagged with the hotfix version

```
git flow hotfix finish VERSION
```

- e.g.

```
git flow hotfix finish REL_1.0.1
```



## 2.15 Git Flow and Continuous Integration

- Each commit that gets pushed to a repository should get picked up automatically by some kind of Continuous Integration build system, such as Jenkins
- The CI system is responsible for trying to integrate proposed code changes with a destination branch, for example when creating a pull-request for a feature branch, the CI system will:
  - ◇ try to merge the feature branch into the destination branch (develop in this case)
  - ◇ build the merged result while enforcing coding style
  - ◇ run unit-tests while capturing code-coverage
  - ◇ run integration-tests
  - ◇ create a package
  - ◇ deploy the package (if continuous deployment is being used for the given branch)
- For lengthy integration-tests, you can decide not to run the tests for pull-request and only run them for release and hotfix branches.



## 2.16 Git Flow – Summary

- It is based on two main branches with an infinite lifetime:
  - ◇ **master**—this branch contains production code. All development code is merged into master
  - ◇ **develop**—this branch contains pre-production code. When the features are finished then they are merged into develop.
- During the development cycle, a variety of supporting branches are used:
  - ◇ **feature-\***—feature branches are used to develop new features for the upcoming releases. May branch off from develop and must merge into develop.
  - ◇ **hotfix-\***—hotfix branches are necessary to act immediately upon an undesired status of master. May branch off from master and must merge into master and develop.
  - ◇ **release-\***—release branch is created for each production release. May branch off from develop and must merge into master and develop.



## 2.17 Git Flow – Pros and Cons

- Pros
  - ◇ Ensures a clean state of branches at any given moment in the life cycle of project
  - ◇ The branches naming follows a systematic pattern making it easier to comprehend
  - ◇ It has extensions and support on most used git tools
  - ◇ It is recommended when there needs to be multiple versions in the production.
- Cons
  - ◇ The master/develop split is considered redundant and makes the Continuous Delivery and the Continuous Integration harder
  - ◇ It isn't recommended when it need to maintain single version in production



## 2.18 Git Flow – When it Works Best?

- **You run an open-source project:** Since everyone can contribute, you want to have very strict access to all the changes.
- **You have a lot of junior developers:** You want to have a way to check their work closely. People who accept pull requests have strict control over recurring changes so they can prevent deteriorating code quality.
- **You have an established product and large enterprises:** In such cases, the focus is usually on application performance and load capabilities. That kind of optimization requires very precise changes. Usually, development time is not a constraint, since they don't want to break their multi-million dollar investment.



## 2.19 Git Flow – When it Doesn't Work?

- **You are just starting up:** Chances are you want to create a minimal viable product quickly. Doing pull requests creates a huge bottleneck that slows the whole team down dramatically.
- **When you need to iterate quickly:** Multiple branches and pull requests reduce development speed dramatically and are not advised in such cases.
- **When you work mostly with senior developers:** If your team consists mainly of senior developers who have worked with one another for a longer period of time, then you don't really need the pull request micromanagement.



## 2.20 Git Flow Alternatives

- There are several alternatives to Git Flow. The most popular ones are:
  - ◇ Trunk-based Development
  - ◇ GitHub Flow
  - ◇ GitLab Flow



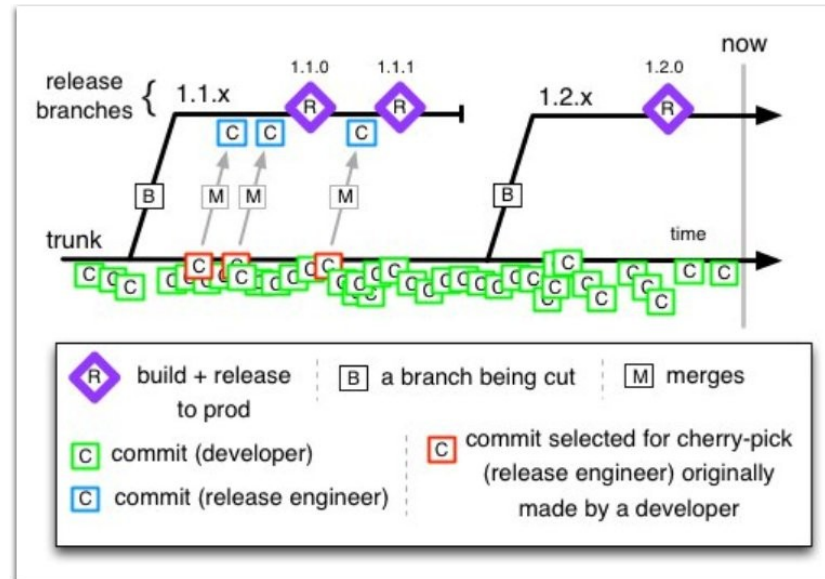


## 2.21 Trunk-based Development

- Trunk-Based Development rejects any feature branches, hotfix branches, parallel release branches. There is only one branch available to the developers—the Trunk.
- All developers work on the trunk where developers directly commit
- A release manager can create release branches and no one can commit to those
- Developers commit small changes as often as they can,
- Commits should be reviewed and tested and must not destroy the mainline.



## 2.22 Trunk-based Development (Contd.)



<https://paulhammant.com/2013/04/05/what-is-trunk-based-development/>

- Developers perform commits, as many as they can, on the "trunk" branch.
- Every commit is a small part of the code, e.g. one function or method with unit tests (green squares).
- At some point, when the trunk branch contains every feature that we want, one person creates a new release branch from the trunk. Such a person is called the release manager or release engineer.





## 2.23 Trunk-based Development – When it Works?

- **You are just starting up:** If you are working on your minimum viable product, then this style is perfect for you. It offers maximum development speed with minimum formality since there are no pull requests.
- **You need to iterate quickly:** If are still in the exploration phase and you need to be able to change your product as fast as possible, this workflow won't slow you down.
- **You work mostly with senior developers:** If your team consists mainly of senior developers, then you should trust them and let them do their job.



## 2.24 Trunk-based Development – When it Doesn't Work?

- **You run an open-source project:** If you are running an open-source project and need very strict control over changes, then Git flow is the better option.
- **You have a lot of junior developers:** If you hire mostly junior developers, then it's a better idea to tightly control what they are doing.
- You are a large enterprise, have established product, or manage large teams



## 2.25 GitHub Flow

- The GitHub Flow is a lightweight workflow created by GitHub in 2011
- It respects the following 6 principles:
  - ◇ Anything in the master branch is deployable
  - ◇ To work on something new, create a branch off from master and given a name
  - ◇ Commit to that branch locally and regularly push your work to the same named branch on the server
  - ◇ When you need feedback or help, or you think the branch is ready for merging, open a pull request
  - ◇ After someone else has reviewed and signed off on the feature, you can merge it into master
  - ◇ Once it is merged and pushed to master, you can and should deploy immediately



## 2.26 GitHub Flow – Pros and Cons

- Pros
  - ◇ A simpler alternative to Git Flow
  - ◇ It is ideal when it needs to maintain a single version in production
- Cons
  - ◇ The production code can become unstable most easily
  - ◇ Not adequate when there's a need for the release plans
  - ◇ It isn't recommended when multiple versions in production are needed



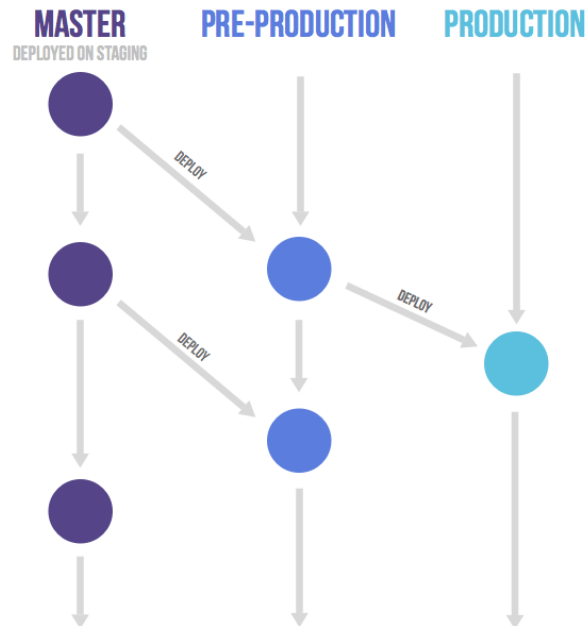
## 2.27 GitLab Flow

- The GitLab Flow was created by GitLab in 2014.
- It combines feature-driven development and feature branches with issue tracking.
- The major difference between GitLab Flow and GitHub Flow is the environment branches in GitLab Flow (e.g. staging and production)
- The GitLab Flow has the following principles:
  - ◇ Use feature branches, no direct commits on master
  - ◇ Perform code reviews before merges into master, not afterward.
  - ◇ Deployments are automatic, based on branches or tags.
  - ◇ Tags are set by the user, not by CI.
  - ◇ Releases are based on tags.
  - ◇ Pushed commits are never rebased.
  - ◇ Everyone starts from master and targets master.
  - ◇ Fix bugs in master first and release branches afterward.





## 2.28 GitLab Flow – Environment Branches



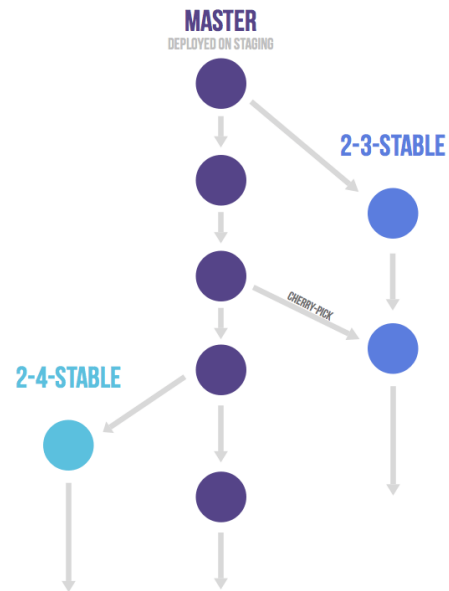
- Suppose you have a staging environment, a pre-production environment, and a production environment. In this case, deploy the master branch to staging. To deploy to pre-production, create a merge request from the master branch to the pre-production branch. Go live by merging the pre-production branch into the production branch.
- This workflow, where commits only flow downstream, ensures that everything is tested in all environments.



- If you need to commit with a hotfix, it is common to develop it on a feature branch and merge it into master with a merge request. In this case, do not delete the feature branch yet. If master passes automatic testing, you then merge the feature branch into the other branches.



## 2.29 GitLab Flow – Release Branches



- You only need to work with release branches if you need to release software to the outside world.
- Each branch contains a minor version, for example, 2-3-stable, 2-4-stable, etc.
- Create stable branches using master as a starting point, and branch as late as possible.
- By doing this, you minimize the length of time during which you have to apply bug fixes to multiple branches.
- After announcing a release branch, only add serious bug fixes to the branch.



### 2.30 GitLab Flow – Release Branches (Contd.)

- If possible, first merge these bug fixes into master, and then cherry-pick them into the release branch.
- If you start by merging into the release branch, you might forget to cherry-pick them into master, and then you'd encounter the same bug in subsequent releases.
- Merging into master and then cherry-picking into release is called an “upstream first” policy, which is also practiced by Google and Red Hat.
- Every time you include a bug fix in a release branch, increase the patch version by setting a new tag.
- Some projects also have a stable branch that points to the same commit as the latest released branch. In this flow, it is not common to have a production branch (or Git flow master branch).



## 2.31 GitLab Flow – Pros and Cons

- Pros
  - ◇ The git history is cleaner
  - ◇ It is ideal when there will be a single version in production
- Cons
  - ◇ It is more complex than GitHub Flow
  - ◇ When there's a need to maintain multiple versions in the production, it can be as complex as Git Flow.



## 2.32 Summary

- Git Flow is a branching model for git
- It is a set of extensions for git.
- The branching model supports features, releases, and hotfixes.
- GitHub Flow and GitLab Flow are the popular alternatives to Git Flow.

## Chapter 3 - Twelve-factor Applications [OPTIONAL]

---

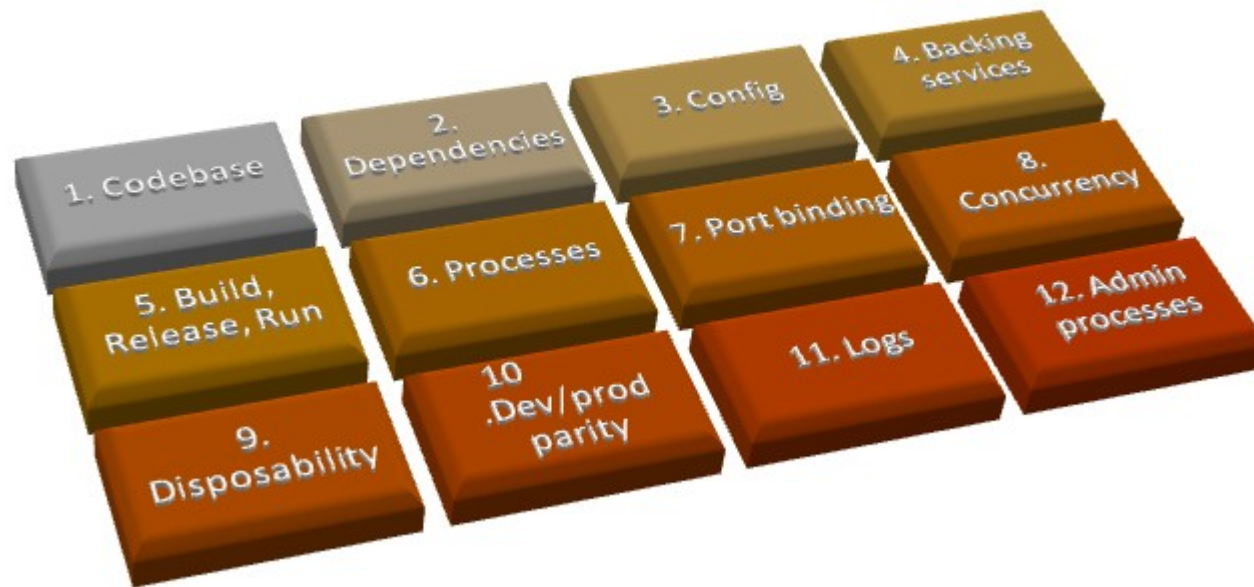
### *Objectives*

Key objectives of this chapter

- Twelve-factor Application Microservices
- Categorizing the Twelve Factors
- Kubernetes and the Twelve Factors



## 3.1 Twelve-factor Applications







## **3.2 Twelve Factors, Microservices, and App Modernization**

- Heroku, a platform as a service (PaaS) provider, established general principles for creating useful web apps known as the Twelve-Factor Application
- Applying 12-factor to microservices requires modification of the original PaaS definitions
- The goal of combining microservices, twelve-factor app and app modernization is a general purpose reference architecture enabling continuous delivery



### 3.3 The Twelve Factors

- I. Codebase - One codebase tracked in revision control, many deploys
- II. Dependencies - Explicitly declare and isolate dependencies
- III. Config - Store config in the environment
- IV. Backing services - Treat backing services as attached resources
- V. Build, release, run - Strictly separate build and run stages
- VI. Processes - Execute the app as one or more stateless processes
- VII. Port binding - Export services via port binding
- VIII. Concurrency - Scale out via the process model
- IX. Disposability - Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity - Keep development, staging, and production as similar as possible
- XI. Logs - Treat logs as event streams
- XII. Admin processes - Run admin/management tasks as one-off processes



## 3.4 Categorizing the 12 Factors

- Code
  - ◇ Codebase
  - ◇ Build, Release, Run
  - ◇ Dev/prod parity
- Deploy
  - ◇ Dependencies
  - ◇ Config
  - ◇ Processes
  - ◇ Backing services
  - ◇ Port Binding
- Operate
  - ◇ Concurrency
  - ◇ Disposability
  - ◇ Logs
  - ◇ Admin Processes



### **3.5 12-Factor Microservice Codebase**

- One codebase per service, tracked in revision control; many deploys
- The Twelve-Factor App recommends one codebase per app. In a microservices architecture, the correct approach is one codebase per service.
- This codebase should be in version control, either distributed, e.g. git, or centralized, e.g. SVN.



### 3.6 12-Factor Microservice Dependencies

- Explicitly declare and isolate dependencies
- As suggested in The Twelve-Factor App, regardless of what platform your application is running on, use the dependency manager included with your language or framework.
- Do not assume that the tool, library or application your code depends on will be there.
- How you install an operating system or platform dependencies depends on the platform:
- In noncontainerized environments, use a configuration management tool (Chef, Puppet, Salt, Ansible) to install system dependencies.
- In a containerized environment, do this in the Dockerfile.



### 3.7 12-Factor Microservice Config

- Store configuration in the environment
- Anything that varies between deployments can be considered configuration.
- All configuration data should be stored in a separate place from the code, and read in by the code at runtime, e.g. when you deploy code to an environment, you copy the correct configuration files into the codebase at that time.
- The Twelve-Factor App guidelines recommend storing all configuration in the environment, rather than committing it to the source code repository.
  - ◇ Use non-version controlled .env files for local development. Docker supports the loading of these files at runtime.
  - ◇ Keep all .env files in a secure storage system, such as Hashicorp Vault, to keep the files available to the development teams, but not committed to Git.
  - ◇ Use an environment variable for anything that can change at runtime, and for any secrets that should not be committed to the shared repository.
  - ◇ Once you have deployed your application to a delivery platform, use the delivery platform's mechanism for managing environment variables



### 3.8 12-Factor Microservice Backing Services

- Treat backing services as attached resources
- The Twelve-Factor App guidelines define a backing service as “any service the app consumes over the network as part of its normal operation.”
- Anything external to a service is treated as an attached resource, including other services. This ensures that every service is completely portable and loosely coupled to the other resources in the system.
- Strict separation increases flexibility during development – developers only need to run the service(s) they are modifying, not others.
- A database, cache, queueing system, etc. These should all be referenced by a simple endpoint (URL) and credentials, if necessary.



### **3.9 12-Factor Microservice Build, Release, Run**

- Strictly separate build and run stages
- To support strict separation of build, release, and run stages, as recommended by The Twelve-Factor App, use a continuous integration/continuous delivery (CI/CD) tool to automate builds.
- Docker images make it easy to separate the build and run stages. Ideally, images are created from every commit and treated as deployment artifacts.





### **3.10 12-Factor Microservice Processes**

- Execute the app in one or more stateless processes
- For microservices, the application needs to be stateless.
- Stateless services scale a service horizontally by simply adding more instances of that service.
- Store any stateful data, or data that needs to be shared between instances, in a backing service



### 3.11 12-Factor Microservice Port Binding

- The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service.
- The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.
- In a local development environment, the developer visits a service URL like `http://localhost:5000/` to access the service exported by their app.
- In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.
- This is typically implemented by using dependency declaration to add a webserver library to the app, such as Tornado for Python, Thin for Ruby, or Jetty for Java and other JVM-based languages.
- This happens entirely in user space, that is, within the app's code.
- The contract with the execution environment is binding to a port to serve requests.
- Nearly any kind of server software can be run via a process binding to a port and awaiting incoming requests. Examples include ejabberd (speaking XMPP), and Redis (speaking the Redis protocol).
- The port-binding approach means that one app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config



for the consuming app.



### **3.12 12-Factor Microservice Concurrency**

- Scale-out via the process model
- The Unix and Mainframe process models are predecessors to a true microservices architecture, allowing specialization and resource sharing for different tasks within a monolithic application.
- For microservices architecture, we horizontally scale each service independently, to the extent supported by the underlying infrastructure.
- Docker or other containerized services, provide service concurrency.



### 3.13 12-Factor Microservice Disposability

- Maximize robustness with fast startup and graceful shutdown
- Instances of a service need to be disposable so they can be started, stopped, and redeployed quickly, and with no loss of data.
- Services deployed in Docker containers satisfy this requirement automatically, as it's an inherent feature of containers that they can be stopped and started instantly.
- Storing state or session data in queues or other backing services ensures that a request is handled seamlessly in the event of a container crash.
- Backing stores support crash-only design.



### **3.14 12-Factor Microservice Dev/Prod Parity**

- Keep development, staging, and production as similar as possible
- Keep all of your environments – development, staging, production, and so on – as identical as possible, to reduce the risk that bugs show up only in some environments.
- Containers enable you to run exactly the same execution environment all the way from local development through production.
- Differences in the underlying data can still result in runtime changes in application behavior



### **3.15 12-Factor Microservice Logs**

- Treat logs as event streams
- Use a log-management solution in a microservice for routing or storing logs.
- Define logging strategy as part of the architecture standards, so all services generate logs in a similar fashion
- Log strategy should be part of a larger Application Performance Management (APM) or Digital Performance Management (DPM) solution tied to the Everything as a Service model (XaaS)



### **3.16 12-Factor Microservice Admin Processes**

- Run admin and management tasks as one-off processes
- In a production environment, run administrative and maintenance tasks separately from the app.
- Containers make this very easy, as you can spin up a container just to run a task and then shut it down.
- Examples include doing data cleanup, running analytics for a presentation, or turning on and off features for A/B testing.





### **3.17 Kubernetes and the Twelve Factors - 1 Codebase**

- Kubernetes makes heavy use of declarative constructs.
- All parts of a Kubernetes application are described with text-based representations in YAML or JSON.
- The referenced containers are themselves described in source code as a Dockerfile
- Because everything from the image to the container deployment behavior is encapsulated in text, you are able to easily source control all the things, typically using git.



### 3.18 Kubernetes and the Twelve Factors - 2 Dependencies

- A microservice is only as reliable as its most unreliable dependency.
- Kubernetes includes *readinessProbes* and *livenessProbes* that enable you to do ongoing dependency checking.
- The *readinessProbe* allows you to validate whether you have backing services that are healthy and you're able to accept requests.
- The *livenessProbe* allows you to confirm that your microservice is healthy on its own.
- If either probe fails over a given window of time and threshold attempts, the Pod will be restarted.



### 3.19 Kubernetes and the Twelve Factors - 3 Config

- The Config factor requires storing configuration sources in your process environment table (e.g. ENV VARs).
- Kubernetes provides *ConfigMaps* and *Secrets* that can be managed in source repositories
  - ◇ *Secrets* should never be source controlled without an additional layer of encryption
- Containers can retrieve the config details at runtime.



### 3.20 Kubernetes and the Twelve Factors - 4 Backing Services

- When you have network dependencies, we treat that dependency as a “Backing Service”.
- At any time, a backing service could be attached or detached and our microservice must be able to respond appropriately.
- For example, you have an application that interacts with a web server, you should isolate all interaction to that web server with some connection details (either dynamic service discovery or via *Config* in a Kubernetes *Secret*). Then consider whether your network requests implement fault tolerance such that if the backing service fails at runtime, your microservice does not trigger a cascading failure. That service may also be running in a separate container or somewhere off-cluster. Your microservice should not care as all interactions then occur through APIs to interact with the database.



### **3.21 Kubernetes and the Twelve Factors - 5 Build, Release, Run**

- Once you commit the code, a build occurs and the container image is built and published to an image registry.
- If you're using Helm, your Kubernetes application may also be packaged and published into a Helm registry as well.
- These “releases” are then re-used and deployed across multiple environments to ensure that an unexpected change is not introduced somewhere in the process (by re-building the binary or image for each environment).



### 3.22 Kubernetes and the Twelve Factors - 6 Processes

- In Kubernetes, a container image runs as a container process within a Pod.
- Kubernetes (and containers in general) provide a facade to provide better isolation of the container process from other containers running on the same host.
- Using a process model enables easier management for scaling and failure recover (e.g. restarts).
- Typically, the process should be stateless to support scaling the workload out through replication.
- For any state used by the application, you should use a persistent data store that all instances of your application process will discover via your Config.
- In Kubernetes-based applications where multiple copies of pods are running, requests can go to any pod, hence the microservice cannot assume sticky sessions.



### 3.23 Kubernetes and the Twelve Factors - 7 Port Binding

- You can use Kubernetes Service objects to declare the network endpoints of your microservices and to resolve the network endpoints of other services in the cluster or off-cluster.
- Without containers, whenever you deployed a new service (or new version), you would have to perform some amount of collision avoidance for ports that are already in use on each host.
- Container isolation allows you to run every process (including multiple versions of the same microservice) on the same port (by using network namespaces in the Linux kernel) on a single host.



### 3.24 Kubernetes and the Twelve Factors - 8 Concurrency

- Kubernetes allows you to scale the stateless application at runtime with various kinds of lifecycle controllers.
- The desired number of replicas are defined in the declarative model and can be changed at runtime.
- Kubernetes defines many lifecycle controllers for concurrency including *ReplicationControllers*, *ReplicaSets*, *Deployments*, *StatefulSets*, *Jobs*, and *DaemonSets*.
- Kubernetes supports autoscaling based on compute resource thresholds around CPU and memory or other external metrics.
- The *Horizontal Pod Autoscaler (HPA)* allows you to automatically scale the number of pods within a Deployment or *ReplicaSet*.





### **3.25 Kubernetes and the Twelve Factors - 9 Disposability**

- Within Kubernetes, you focus on the simple unit of deployment of Pods which can be created and destroyed as needed—no single Pod is all that valuable.
- When you achieve disposability, you can start up fast and the microservices can die at any time with no impact on user experience.
- With the livenessProbes and readinessProbes, Kubernetes will actually destroy Pods that are not healthy over a given window of time.



### **3.26 Kubernetes and the Twelve Factors - 10 Dev/Prod Parity**

- Containers (and to a large extent Kubernetes) standardize how you deliver your application and its running dependencies, meaning that you're able to deploy everything the same way everywhere.
- For example, if you're using MySQL in a highly available configuration in production, you can deploy the same architecture of MySQL in your dev cluster.
- By establishing parity of production architectures in earlier dev environments, you can typically avoid unforeseen differences that are important to how the application runs (or more importantly how it fails).



### **3.27 Kubernetes and the Twelve Factors - 11 Logs**

- For containers, you will typically write all logs to stdout and stderr file descriptors.
- The important design point is that a container should not attempt to manage internal files for log output, but instead delegate to the container orchestration system around it to collect logs and handle analysis and archival.
- Often in Kubernetes, you'll configure Log collection as one of the common services to manage Kubernetes.
- For example, you can enable an Elasticsearch-Logstash-Kibana (ELK) stack within the cluster.



### **3.28 Kubernetes and the Twelve Factors - 12 Admin Processes**

- Within Kubernetes, the Job controller allows you to create Pods that are run once or on a schedule to perform various activities.
- A Job might implement business logic, but because Kubernetes mounts API tokens into the Pod, you can also use them for interacting with the Kubernetes orchestrator as well.
- By isolating these kinds of administrative tasks, you can further simplify the behavior of your microservice.



### 3.29 Summary

- The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).
- The twelve-factor methodology is highly useful when creating microservices architecture based applications.

## Chapter 4 - Microservice Development

---

### *Objectives*

Key objectives of this chapter

- Principles of Microservices Architecture Design
- Domain-Driven Design
- Designing for Failure
- Managing Distributed Services with Docker & Kubernetes (OpenShift)
- Microservices and their relationship to the front-end
- Single Page Applications (SPA)
- Example of Microservices Architecture



## 4.1 What are Microservices?

- Componentized services which work well together.
- Independent processes for each microservice
- Communication through APIs, rather than using databases directly
- Architecture patterns tied to observable, distributed, secure, scalable, and agile componentized development
- A form of SOA. Typical SOA-based applications are considered monoliths and don't lend themselves well to the world of containerized applications



## 4.2 Microservices vs Classic SOA

SOA	Microservices
XML	JSON
Monolithic	Component
Heavy	Lightweight
HTTP/SOAP	HTTP/REST





### 4.3 Principles of Microservices Architecture Design

- Modeled around business domain
  - ◇ Domain-driven design that can help you find stable, reusable boundaries
- Culture of automation
  - ◇ Moving parts means Automation is key
- Hide implementation details
  - ◇ One of the pitfalls that distributed systems can often fall into is tightly coupling their services together, robbing them of their autonomy. Working out how to hide internal implementation details is one step on the way to avoiding this problem.
- Decentralize all the things
  - ◇ To achieve many of the benefits of these architectures, autonomy is key. And to achieve this you often need to push power out of the center, but organizationally and architecturally.
- Deploy independently
  - ◇ The most important characteristics your microservices need.
- Consumer first
  - ◇ Microservices need to work together and to make that work we need to make them easy to consume



- Isolate failure
  - ◇ Embracing a microservice architecture doesn't automatically make your systems more stable, in fact, the opposite can be the case. Understanding how to isolate a failure in your system is key to ensure that your system doesn't become a fragile edifice.
- Highly observable
  - ◇ With many moving parts, understanding what is happening in your system can be much more challenging.



## 4.4 Domain-Driven Design

- Initially introduced and made popular by programmer Eric Evans in his 2004 book, Domain-Driven Design: Tackling Complexity in the Heart of Software
- The domain-driven design applies to the development of software.
- It aims to ease the creation of complex applications by connecting the related pieces of the software into an ever-evolving model.
- The domain-driven design focuses on three core principles:
  - ◇ Focus on the core domain and domain logic.
  - ◇ Base complex designs on models of the domain.
  - ◇ Constantly collaborate with domain experts, in order to improve the application model and resolve any emerging domain-related issues.



## 4.5 Domain-Driven Design - Benefits

- **Code reusability** - it is achieved by creating APIs in the application code using protocols, such as REST.
- **Evolution** - When it comes to refactoring, or modifying your software according to new requirements, the domain-centric approach works better than the data-centric one.
  - ◇ Teams adhering to the domain-centric standpoint tend to work on the application code and the database together
- **Complexity growth** - DDD brings additional maintenance overhead at the beginning but pays off greatly over time.
  - ◇ At some point, the domain-centric method overtakes the data-centric approach in terms of complexity
  - ◇ It becomes easier to maintain and evolve a system adhering to its principles.
  - ◇ DDD is worth the investment because the problem domain itself is more important than the data it produces.
  - ◇ The investments we make in the modeling of that domain have better ROI.



## 4.6 Microservices and Domain-Driven Design

- Microservices are built around business capabilities
- Each component runs as a separate application, clustered to as many nodes as required





## 4.7 Designing for failure

- Microservices architecture based components are designed for failure.
- Any service can fail, anytime
- The client application has to respond as gracefully as possible
- It's important to be able to detect the failures quickly and, if possible, automatically restore service
- Microservices applications put a lot of emphasis on real-time monitoring
- Netflix's Chaos Monkey induces failures of services during the working day to test the application's resilience and monitoring



## **4.8 Microservices Architecture – Pros**

- Multiple developers and teams can deliver relatively independently of each other
- Can be written in different programming languages
- Can be managed by different teams
- Can use different data storage technologies AKA backing services
- Independently deployable by fully automated deployment machinery
- Typically maps to smaller Agile team structure and notions of self-organizing



## **4.9 Microservices Architecture – Cons**

- Distributed transactions require orchestration services
- Difficult testing – interactions between the services. May require mock testing, if service doesn't exist
- Robust error handling and recovery is required
- Sophisticated real-time monitoring required





## 4.10 Docker and Microservices

- Docker containers drives the adoption of microservices and vice versa
- Docker puts significant emphasis on the "Unix philosophy" of shipping containers, i.e. "do one thing, and do it well".
- Docker documentation itself says:
  - ◇ Run only one process per container. In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers
- Such principles at Docker's core philosophy makes it much closer to the microservice architecture than a conventional, large monolithic architecture.
- When aiming for "doing one thing" it doesn't make sense to containerize the entire, huge, enterprise application as a single Docker container.
- You would want to first modularize the application into loosely coupled components that communicate via standard protocols, which in essence, is what the microservices architecture delivers.
- Docker containers and microservices architecture are two ends of the road that leads to the same ultimate goal of continuous delivery



## 4.11 Microservice Deployment with Docker – Workflow

- A typical workflow for microservice deployment with Docker would involve following
  - ◇ The code is checked into the source code repository. If this is the project's first check-in, it is done along with Dockerfile for the project
  - ◇ The preceding check-in triggers the build engine, such as Maven/MSBuild, to build the service from the source code and run unit/integration tests
  - ◇ If tests are successful, the Docker image is created and pushed to a Docker registry.
  - ◇ If QA tests pass as well, the container is promoted to deploy and start in production.



## 4.12 Writing Dockerfile

```
FROM openjdk:8
RUN mkdir -p /opt/my/service
ADD target/myservice-0.0.1-SNAPSHOT.jar /opt/my/service/
EXPOSE 8080
CMD ["java", "-jar", "/opt/my/service/myservice-0.0.1-SNAPSHOT.jar"]
```

- The above Dockerfile does following
  - ◇ Uses openjdk:8 image from the Docker hub
  - ◇ Creates a directory for storing custom service files
  - ◇ Copies myservice\*.jar file into the container
  - ◇ Makes port 8080 so it can be accessed from outside the container
  - ◇ Executes the custom service by using java command line tool.



## 4.13 Kubernetes

- Kubernetes offers container orchestration
- Kubernetes provides a platform for automating deployment, scaling, and operations of application containers across clusters of hosts
- Kubernetes supports a range of container tools, including Docker
- Microservices require a reliable way to find and communicate with each other.
- Microservices in containers and clusters can make things more complex as we now have multiple networking namespaces to bear in mind.
- Communication and discovery requires traversing of container IP space and host networking.
- Kubernetes benefits from getting its ancestry from the clustering tools used by Google for the past decade. Many of the lessons learned from running and networking two billion containers per week have been distilled into Kubernetes

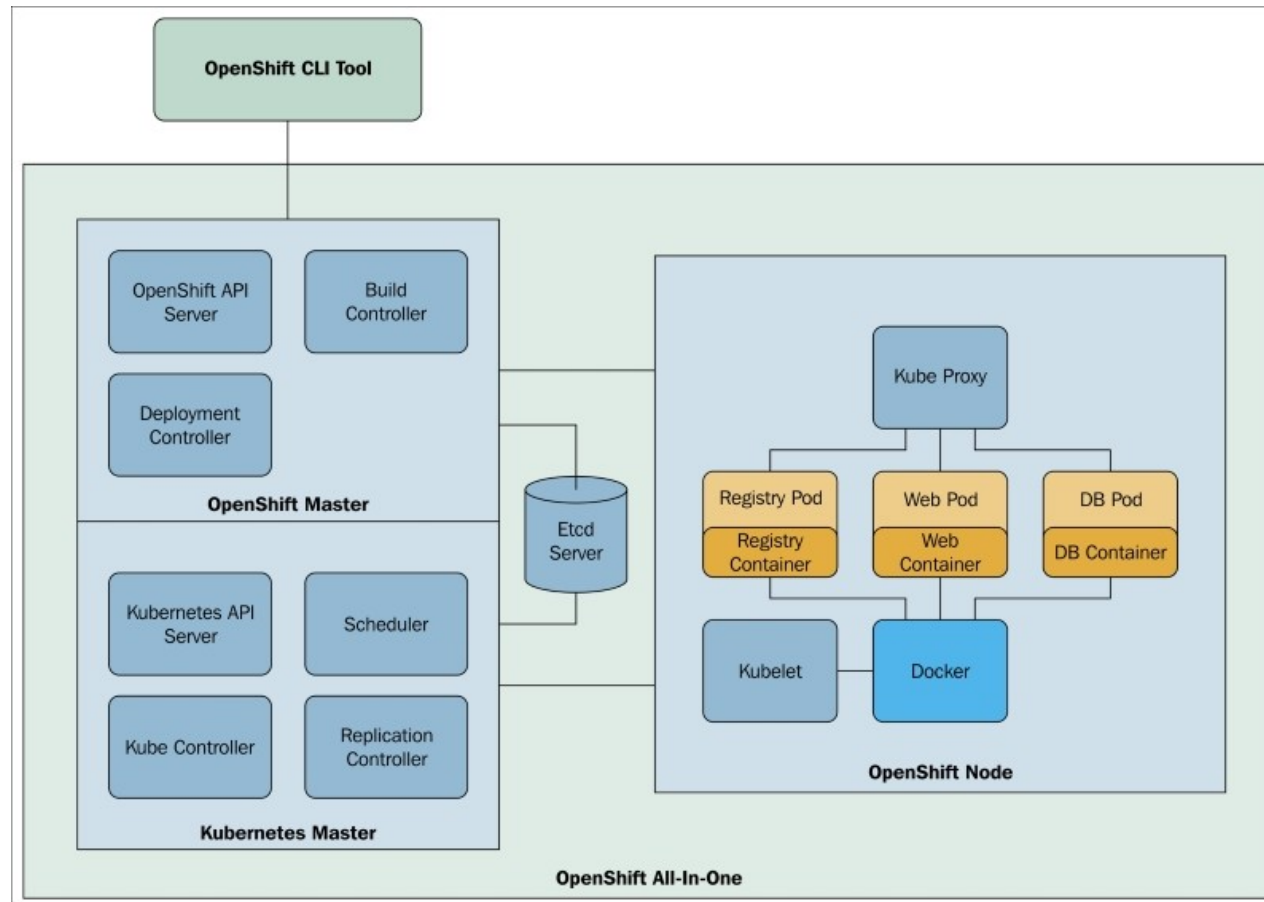


## 4.14 What is OpenShift

- OpenShift is a computer software product from Red Hat
- It provides a container-based software deployment and management solution
- In short, it's a container orchestration software
- It is a supported distribution of Kubernetes using Docker containers and DevOps tools



## 4.15 OpenShift Architecture





## 4.16 Microservices and Various Applications

- There are various types of applications, such as:
  - ◇ Web applications
  - ◇ Rich client applications
  - ◇ Rich internet applications
  - ◇ Mobile applications
  - ◇ Services applications
- Almost all these application types can benefit from Microservices architecture.



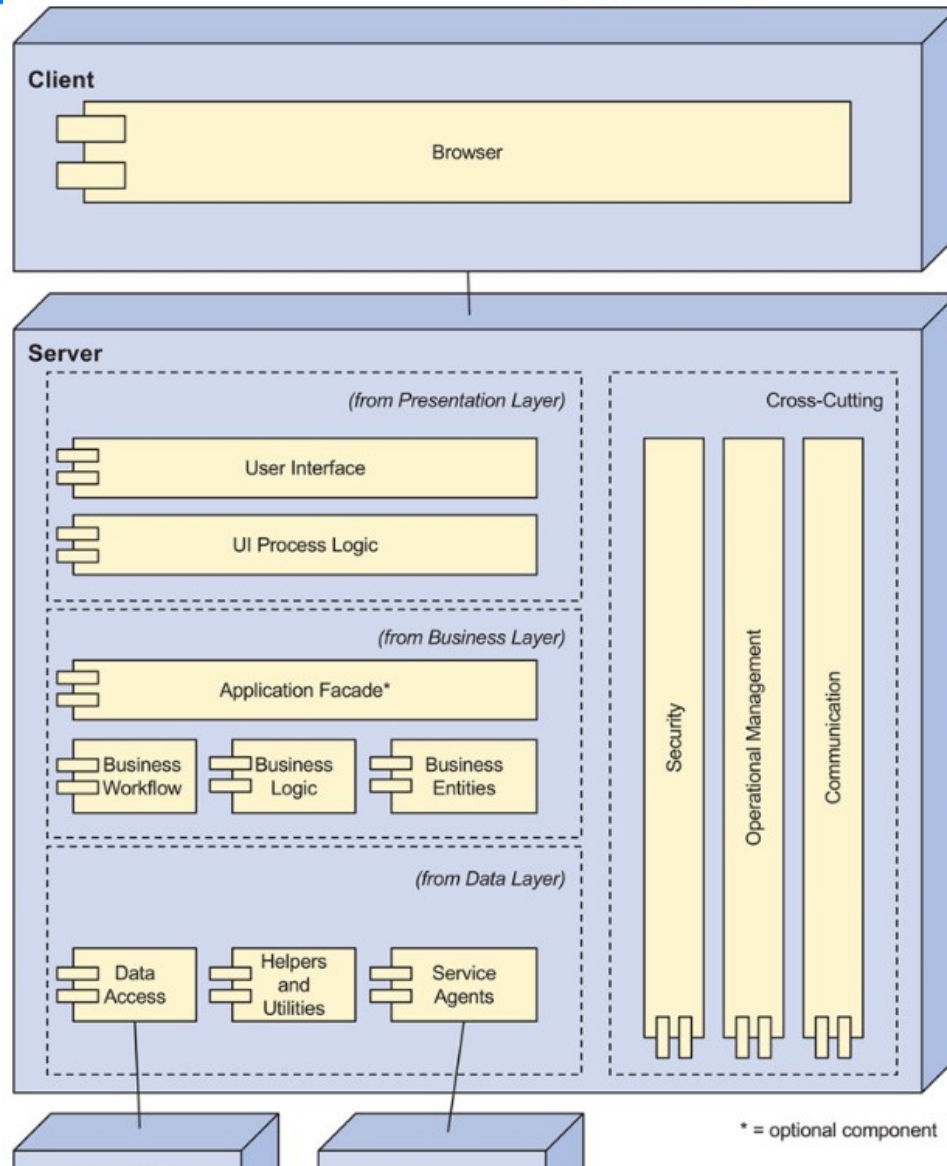
## 4.17 Web Applications

- This web application is typically initiated from a web browser that communicates with a server using the HTTP protocol.
- The bulk of the application resides on the server, and its architecture is typically composed of three layers:
  - ◇ presentation - contains modules that are responsible for managing user interaction
  - ◇ business - contains modules that handle aspects related to the business logic
  - ◇ data layers - contains modules that manage data that is stored either locally or remotely.
- In addition, certain functionality that is common to modules across the layers is organized as cross-cutting concerns.
- This cross-cutting functionality includes aspects related to security, logging, and exception management.





## **4.18 Web Applications – Reference Architecture**







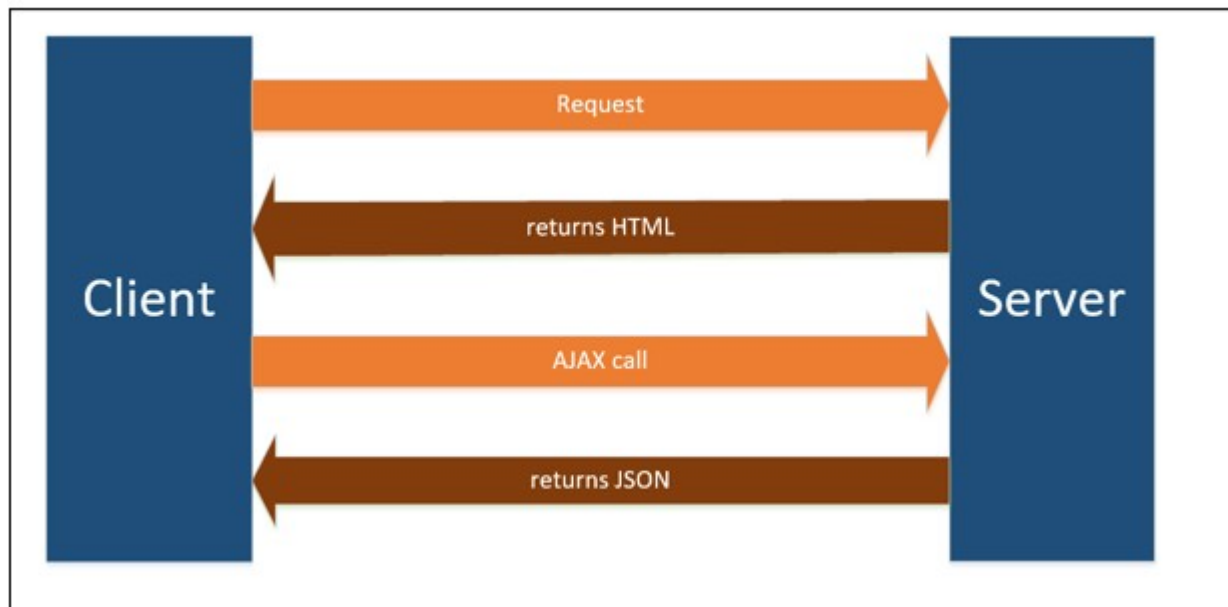
### **4.19 Web Applications – When to use?**

- You should consider using this type of application when:
  - ◇ You do not require a rich user interface.
  - ◇ You do not want to deploy the application by installing anything on the client machine
  - ◇ You require portability of the user interface.
  - ◇ Your application needs to be accessible over the Internet.
  - ◇ You want to use a minimum of client-side resources.



## 4.20 Single Page Applications

- The SPAs are web applications that have a single web page and all views render dynamically when the user interacts with the application.
- SPAs use AJAX to call backend data through services and most of the work is done on the client side:





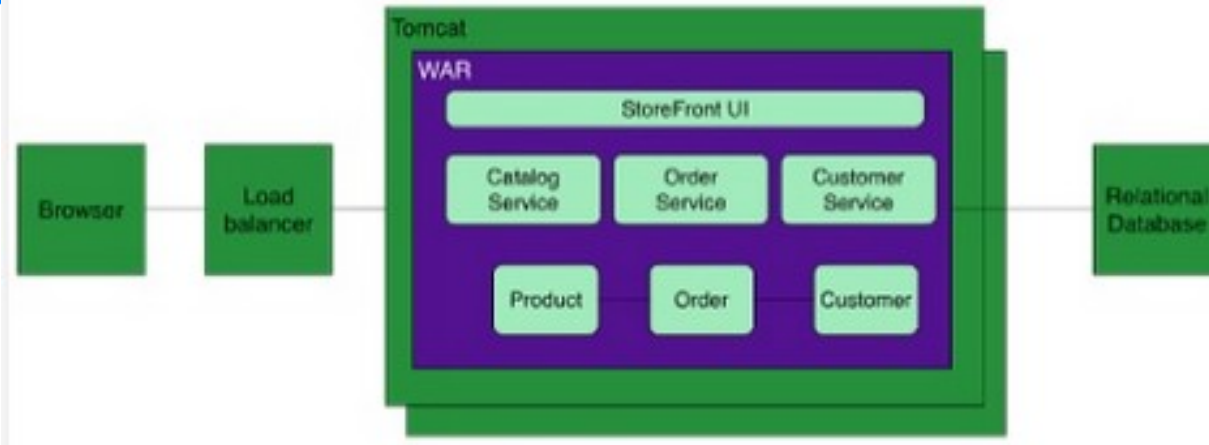
### **4.21 Single Page Applications – Benefits**

- SPAs are responsive in nature, as most of the resources, including CSS, JavaScript, and Images, are only loaded once throughout the application lifespan
- A SPA reduces the size of response by making AJAX requests to the server and receives a JSON response
- SPAs make it easy to scale and cache resources



## 4.22 Traditional Enterprise Application Architecture

- Classical architecture
- Typical 3 layers:
  - ◇ client-side UI (Browser, HTML + JS)
  - ◇ a database (RDBMS, NoSQL ...)
  - ◇ server-side application (Java, .NET, PHP, ...)
- Any changes to the system involve building and deploying a new version of the application. Changes are expensive.
- Scaling requires scaling of the entire application, rather than parts of it that require greater resource.
- Long release cycles.

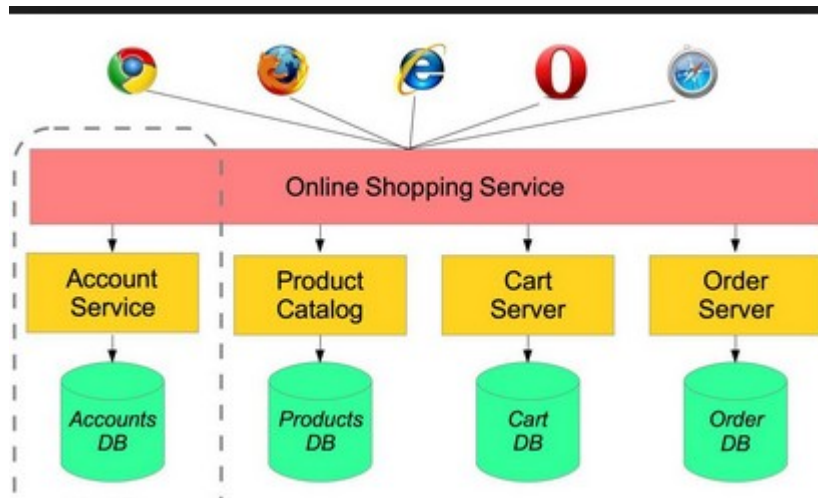






## 4.23 Sample Microservices Architecture

- Applications naturally start as Monoliths, they scale and evolve to Microservice architecture
- Applications are decomposed to components – smaller independent service applications.
- Components are loosely coupled.





## 4.24 Serverless & Event-driven Microservice – AWS Lambda

- AWS Lambda is a compute service that lets you run code without provisioning or managing servers.
- AWS Lambda executes your code only when needed and scaled automatically, from a few requests per day to thousands per second.
- You pay only for the compute time you consume - there is no charge when your code is not running.
- With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration.
- AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging.
  - ◇ All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js, Java, C# and Python).



## 4.25 Summary

- Small, autonomous services which work well together.
- Microservices are designed for failure
- Docker containers play a very important role in implementing Microservices architecture
- Kubernetes provides a container orchestration solution. It has built-in functionality for implementing fault-tolerance and load balancing.
- OpenShift makes configuration and management of Docker containers and Kubernetes much easier.

## Chapter 5 - Introduction to Gradle

---

### *Objectives*

Key objectives of this chapter

- What is Gradle
- Why Groovy
- Build Script
- Task Dependencies
- Plugins
- Dependency Management
- Gradle Command-line Arguments



## 5.1 What is Gradle

- Gradle is a flexible general purpose build tool like ANT
- Provides powerful dependency management
- Full support for your existing Maven or Ivy repository infrastructure
- Ant tasks are also supported
- Groovy and Kotlin languages are supported for writing scripts
- It is free and an open source project, and licensed under the Apache Software License (ASL)
- Gradle can increase productivity, from single project builds to huge enterprise multi-project builds



## 5.2 Why Groovy?

- ANT uses declarative XML based style
- Gradle uses a DSL (domain specific language) based on Groovy which makes it more powerful
  - ◇ Also supports Kotlin
- Gradle's main focus is on Java projects, but it's still a general purpose build tool
- Groovy offers transparency for Java people
- Groovy's base syntax is the same as Java's



## 5.3 Build Script

- build.gradle file
  - ◇ Analogous to ANT's build.xml file
- Written in Groovy
- Composed of tasks
  - ◇ Similar to ANT targets
- Tasks are composed of actions
  - ◇ doFirst, doLast



## 5.4 Sample Build Script

```
task hello1 {
    doLast {
        println 'Hello World!'
    }
}

task hello2 {
    doFirst {
        print 'This is '
    }
    doLast {
        println 'a test!'
    }
}
```





## 5.5 Task Dependencies

- A task can depend on one or more tasks
- If a dependent task already exists earlier in the build script, then the following syntax can be used

```
task <task_name>(dependsOn: dependentTask)
```

- If not, then the dependent task name must be enclosed in quotes like this:

```
task <task_name>(dependsOn: 'dependentTask')
```



## 5.6 Plugins

- Plugins can be defined in the build script to make more tasks available to Gradle
- Examples
  - ◇ Java plugin – provides tasks for compiling Java code, copying resources, creating a JAR file, running unit tests and generating JavaDocs
  - ◇ Application plugin – provides a task to create an executable JVM application. Packages application as a TAR and/or ZIP file and includes operating system specific start scripts.
- Add plugins to build.gradle file using the following syntax:

```
plugins {  
    id 'java'  
    id 'application'  
}
```



## 5.7 Dependency Management

- A project can make use of additional libraries that are not already part of current project
- Java projects can connect to various repositories, such as Maven Central and JCenter
- Repositories can be defined like this:

```
repositories {  
    jcenter()  
}
```

- Dependencies can be defined like this:

```
dependencies {  
    testImplementation 'junit:junit:4.12'  
}
```



## 5.8 Gradle Command-Line Arguments

- `gradle tasks`: displays the tasks defined in the build script, including plugin provided tasks
- `gradle -q`: suppresses the log messages and runs the default tasks
- `gradle build`: compiles code, generates JAR file and runs the unit tests
- `gradle clean`: cleans the build directory
- `gradle test`: runs the unit tests



## 5.9 Summary

- Gradle is an open source, general purpose build tool
- Gradle offers more flexibility since it uses Groovy as a DSL language, which is closer to Java, instead of using XML based syntax
- Gradle allows dependency management for packages / libraries

## Chapter 6 - Introduction to Spring Boot

---

### *Objectives*

Key objectives of this chapter

- Overview of Spring Boot
- Using Spring Boot for building microservices
- Examples of using Spring Boot



## 6.1 What is Spring Boot?

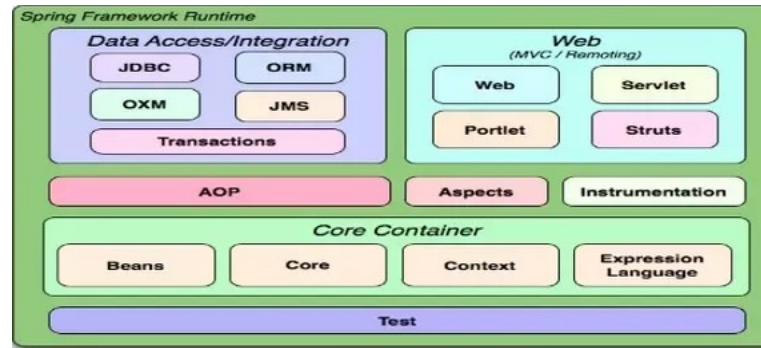
- Spring Boot (<http://projects.spring.io/spring-boot/>) is a project within the Spring IO Platform (<https://spring.io/platform>)
- Developed in response to Spring Platform Request SPR-9888 "Improved support for 'containerless' web application architectures"
- Inspired by the DropWizard Java framework (<http://www.dropwizard.io/>)
- The main focus of Spring Boot is on facilitating a fast-path creation of stand-alone web applications packaged as executable JAR files with minimum configuration
- An excellent choice for creating microservices
- Released for general availability in April 2014
  - ◇ (<https://spring.io/blog/2014/04/01/spring-boot-1-0-ga-released>)
- Version 2 was released in Spring of 2017



## 6.2 Spring Framework

- Simply put, the Spring framework provides comprehensive support for developing Java applications.
- It lets you utilize features like Dependency Injection and comes with out of the box modules like:
  - ◇ Spring JDBC
  - ◇ Spring MVC
  - ◇ Spring Security
  - ◇ Spring AOP
  - ◇ Spring ORM
  - ◇ Spring Test
- These modules can drastically reduce the development time of an application.







### 6.3 How is Spring Boot Related to Spring Framework?

- Unlike Spring, Spring Boot is not a framework. Instead, it's an extension of Spring Framework.
- It is a way to easily create stand-alone applications with minimal or zero configurations.
- It provides boilerplate code and annotation configuration to quick start new spring projects.
- It provides a set of starter pom or gradle build files which one can use to add required dependencies and also facilitate autoconfiguration.
- It provides 'starter' dependencies to simplify build and application configuration
- It provides Embedded HTTP servers like Tomcat, Jetty, etc. to develop and test our web applications very easily.
- Spring Boot provides lots of plugins to develop and test Spring Boot Applications very easily using build tools like Maven and Gradle



## 6.4 Spring Boot 2

- Spring Boot is currently at version 2.2.6
- Spring Boot 2 extends Spring Framework 5
- Spring Boot 2 requires Java 8 minimum
- Breaking changes in security configuration
  - ◇ No more default username and password
  - ◇ Everything is now secured by default, including static resources and endpoints
- Support for reactive programming
- Actuator endpoints are now independent of Spring MVC
  - ◇ Custom endpoints can be created
  - ◇ Predefined endpoints are no longer configurable
- Spring Boot DAO API are now in line with JPA APIs
  - ◇ .findOne is now called .findById
  - ◇ .findById returns an Optional<T> instead of T



## 6.5 Spring Boot Main Features

- Spring Boot offers web developers the following features:
  - ◇ Ability to create WAR-less stand-alone web applications that you can run from command line
  - ◇ Embedded web container that is bootstrapped from the `public static void main` method of your web application module:
    - Tomcat Servlet container is default; you have options to plug in Jetty or Undertow containers instead
  - ◇ No, or minimal configuration
    - Spring Boot relies on Spring MVC annotations (more on annotations later ...) for configuration
  - ◇ Built-in production-ready features for run-time metrics collection, health checks, and externalized configuration



## 6.6 Spring Boot on the PaaS

- Spring boot is designed to be conveniently run in a container. All major cloud provider can run Docker and Kubernetes
- Older projects might use PaaS (Platform as a Service)
- PaaS offer robust, scalable, and cost-efficient run-time environments that can be used with minimum operational involvement
- Spring Boot is being financed by Pivotal Software Inc. which also, in partnership with VMware, sponsors Cloud Foundry
  - ◇ While Cloud Foundry is available for free, Pivotal offers a commercial version of it called Pivotal Web Services



## 6.7 Understanding Java Annotations

- Annotations in Java are syntactic metadata that is baked right into Java source code; you can annotate Java classes, methods, variables and parameters
- Basically, an annotation is a kind of label that is processed during a compilation stage by annotation processors when the code or configuration associated with the annotation is injected in the resulting Java class file, or some additional operations associated with the annotation are performed
- An annotation name is prefixed with an '@' character
- **Note:** Should you require to change annotation-based configuration in your application, you would need to do it at source level and then re-build your application from scratch



## 6.8 Spring MVC Annotations

- Spring Boot leverages Spring MVC (Model/View/Controller) annotations instead of XML-based configuration
  - ◇ Additional REST annotations, like *@RestController* have been added with Spring 4.0
- The main Spring MVC annotations are:
  - ◇ **@Controller** / **@RestController** – annotate a Java class as an HTTP end-point
  - ◇ **@RequestMapping** – annotate a method to configure with URL path / HTTP verb the method responds to
  - ◇ **@RequestParam** - named request parameter extracted from client HTTP request



## 6.9 Example of Spring MVC-based RESTful Web Service

- The following code snippet shows some of the more important artifacts of Spring MVC annotations (with REST-related Spring 4.0 extensions) that you can apply to your Java class
- **Note:** Additional steps to provision and configure a web container to run this module on are required

```
// ... required imports are omitted
@RestController
public class EchoController {
    @RequestMapping(path="/echoservice", method=RequestMethod.GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
}
```

- The above annotated Java class, when deployed as a Spring MVC module, will echo back any *echo* message send with an HTTP GET request to this URL:

`http(s)://<Deployment-specific>/echoservice?id=hey`





## 6.10 Spring Booting Your RESTful Web Service

- Spring Boot allows you to build production-grade web application without the hassle of provisioning, setting up, and configuring the web container
- A Spring Boot application is a regular executable JAR file that includes the required infrastructure components and your compiled class that must have the **public static void main** method which is called by the Java VM on application submission
  - ◇ The **public static void main** method references the **SpringApplication.run** method that loads and activates Spring annotation processors, provisions the default Tomcat Servlet container and deploys the Spring Boot-annotated modules on it



## 6.11 Spring Boot Skeletal Application Example

- The following code is a complete Spring Boot application based on the Spring MVC REST controller from a couple of slides back

```
// ... required imports are omitted
@SpringBootApplication
@RestController
public class EchoController {
    @RequestMapping(path="/echoservice", method=RequestMethod.GET)
    public String echoback(@RequestParam(value="id") String echo) {
        return echo;
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(EchoController.class, args);
    }
}
```

- You run Spring Boot applications from command line as a regular executable JAR file:

```
java -jar Your_Spring_Boot_App.jar
```

- The default port the Spring Boot embedded web container starts listening on is 8080
  - ◇ To change the default port, you need to pass a **server.port** System property or specify it in the Spring's **application.properties** file



## 6.12 Converting a Spring Boot Application to a WAR File

- In some scenarios, users want to have a Spring Boot runnable JAR file converted into a WAR file
- For those situations, Spring Boot provides two plug-ins:
  - ◇ **spring-boot-gradle-plugin** for the Gradle build system (<https://gradle.org/>)
  - ◇ **spring-boot-maven-plugin** for the Maven build system (<https://maven.apache.org/>)
- For more details on Spring Boot JAR to WAR conversion, visit <https://spring.io/guides/gs/convert-jar-to-war/>



## 6.13 Externalized Configuration

- Lets you deploy the same Spring Boot artifact (jar file) in different environments—configuration is drawn from environment variables
- Properties are pulled from (note - this is an incomplete list)
  - ◇ OS Environment variable `SPRING_APPLICATION_JSON`
  - ◇ Java System properties
  - ◇ OS Environment variables
  - ◇ Application property files - `application.properties` or `application.yml`
    - in a `/config` folder in current directory
    - in the current directory
    - in a classpath `/config` package
    - in the classpath root
    - In a folder pointed to by `'spring.config.location'` on the command line.



## 6.14 Starters

- Spring Boot auto-configures based on what it finds on the classpath
- So, the set of modules is determined by what's in the 'pom.xml'
- The project provides a number of 'starter' artifacts that pull in the correct dependencies for a given technology
- Some examples:
  - ◇ spring-boot-starter-web
  - ◇ spring-boot-starter-jdbc
  - ◇ spring-boot-starter-amqp
- You just need to include these artifacts in the 'pom.xml' to configure those technologies.



## 6.15 Maven - The 'pom.xml' File

- Assuming you're building with Apache Maven, the 'pom.xml' file describes all the artifacts and build tools that go into producing a delivered artifact.
- Generally, use the 'starter parent':

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.6.RELEASE</version>
</parent>
```

- ◊ It defines all the dependency management and core dependencies for a Spring Boot application



## 6.16 Maven - The 'pom.xml' File

- Add dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    ...
  </dependency>
</dependencies>
```



## 6.17 Spring Boot Maven Plugin

- The Maven plugin can package the project as an executable jar file, that includes all the dependencies

```
<build>
  <plugins>
    <plugin>
      <groupId>
        org.springframework.boot
      </groupId>
      <artifactId>
        spring-boot-maven-plugin
      </artifactId>
    </plugin>
  </plugins>
</build>
```





## 6.18 Gradle - The 'build.gradle' File

- To define the dependencies, use the dependencies block as shown below:

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    testImplementation('org.springframework.boot:spring-boot-starter-  
test')  
    ...  
}
```

- It defines the dependency management and core dependencies for a Spring Boot application



## 6.19 Spring Boot Maven Plugin

- Spring provides a standalone Spring Boot Gradle plugin which adds some tasks and configurations to ease the work with Spring Boot based projects.
- Assuming you're building with Gradle, the 'build.gradle' file describes all the artifacts and build tools that go into producing a delivered artifact.

```
plugins {  
    id 'org.springframework.boot' version '2.2.6.RELEASE'  
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'  
    id 'java'  
}
```



## 6.20 HOWTO: Create a Spring Boot Application

- Create a new Maven project in the IDE of your choice
- In 'pom.xml',
  - ◇ add the Spring Boot parent project (see above)
  - ◇ Add dependencies to the 'starter' projects that describe the features you want (e.g. Spring MVC, JDBC, etc)
  - ◇ Add the 'spring-boot-maven-plugin' to the build plugins
- Add a default 'application.properties' or 'application.yml' file in 'src/main/resources'
  - ◇ List of properties is at:
    - <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>
    - At least include 'spring.application.name'



## 6.21 HOWTO: Create a Spring Boot Application

- Create a main class in `src/main/java/<package>`
- e.g. in `'com.mycom.app'`,

```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

- Build with `'mvn install'`
- From the IDE, run the main class
- From command line, run the jar file




## 6.22 Spring Initializr

- Spring also offers an Initializr website which can be used to create a new Spring Boot project
  - ◇ <https://start.spring.io/>
- Use it to configure whether you want to use Maven or Gradle, the Spring Boot version, language, project metadata, Java version, packaging format and dependencies.
- Preview the Gradle or Maven build file using the **EXPLORE** button.
- When done, click **GENERATE** button to download a zip file.
- Import the zip file into your favorite IDE.



## 6.23 Spring Initializr



**Project**  
☒ Maven Project  
☐ Gradle Project

**Language**  
☒ Java ☐ Kotlin  
☐ Groovy

**Spring Boot**  
☐ 2.3.0 M4 ☐ 2.3.0 (SNAPSHOT)  
☐ 2.2.7 (SNAPSHOT) ☒ 2.2.6  
☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

**Project Metadata**  

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 14 ☐ 11 ☒ 8

**Dependencies** ADD ... ⌘ + B  

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...



## 6.24 Summary

- Spring Boot eliminates many of the headaches related to provisioning, setting up and configuring a web server by offering a framework for running WAR-less web applications using embedded web containers
- Spring Boot favors annotation-based configuration over XML-based one
  - ◇ Any change to such a configuration (e.g. a change to the path a REST-enabled method responds to), would require changes at source level and recompilation of the project
- Spring Boot leverages much of the work done in Spring MVC

## Chapter 7 - Spring REST Services

---

### *Objectives*

Key objectives of this chapter

- A Basic introduction to REST-style services
- Using Spring MVC to implement REST services
- Combining the JAX-RS standard with Spring





## 7.1 Many Flavors of Services

- Web Services come in all shapes and sizes
  - ◇ XML-based services (SOAP, XML-RPC, RSS / ATOM Feeds)
  - ◇ HTTP services (REST, JSON, standard GET / POST)
  - ◇ Other services (FTP, SMTP)
- While SOAP is the most common style of service, increasingly organizations are utilizing REST for certain scenarios
  - ◇ REpresentational State Transfer (REST), first introduced by Roy Fielding (co-founder of the Apache Software Foundation and co-author of HTTP and URI RFCs)
  - ◇ REST emphasizes the importance of resources, expressed as URIs
  - ◇ Used extensively by Amazon, Google, Yahoo, Flickr, and others



## 7.2 Understanding REST

- REST applies the traditional, well-known architecture of the Web to Web Services
  - ◇ Everything is a resource
  - ◇ Each URI is treated as a distinct resource and is addressable and accessible using an application or Web browser
  - ◇ URIs can be bookmarked and even cached
- Leverages HTTP for working with resources
  - ◇ GET – Retrieve a representation of a resource. Does not modify the server state. A GET should have no side effects on the server side.
  - ◇ DELETE – Remove a representation of a resource
  - ◇ POST – Create or update a representation of a resource
  - ◇ PUT – Update a representation of a resource



## 7.3 RESTful Services

- A RESTful Web service services as the interface to one or more resource collections.
- There are three essential elements to any RESTful service
  - ◇ Resource Address – expressed as a URI
  - ◇ Representation Format – a known MIME type such as TXT or XML, common data formats include JSON, RSS / ATOM, and plain text
  - ◇ Resource Operations – a list of supported HTTP methods (GET, POST, PUT, DELETE)



## 7.4 REST Resource Examples

- GET /checkflightstatus/AA1215 HTTP/1.1
  - ◇ Check the flight status for American Airlines flight #1215
- POST /checkflightstatus HTTP/1.1
  - ◇ Upload a new flight status by sending an XML document that conforms to a previously defined XML Schema
  - ◇ Response is a “201 Created” and a new URI

201 Created

Content-Location: /checkflightstatus/AA1215

- PUT /checkflightstatus/AA1215 HTTP/1.1
  - ◇ Update an existing resource representation
- DELETE /checkflightstatus/AA1215 HTTP/1.1
  - ◇ Delete the resource representation



## 7.5 @RestController Annotation

- Spring 4 (and by extension, Spring Boot) added the @RestController annotation
- It combines @Controller and @ResponseBody
- So now you just need one annotation on a REST controller class.



## 7.6 Implementing JAX-RS Services and Spring

- JAX-RS is the official Java specification for defining REST services in Java
  - ◇ Similar to Spring MVC this is done mainly with JAX-RS annotations although these annotations are different
- Spring MVC is NOT a JAX-RS implementation
- It is possible though to use standard JAX-RS code and implementation to define the REST service itself and simply inject Spring components into the JAX-RS service class
  - ◇ This would provide for more "standard" code for the REST service but allow the other features of Spring to be used as well
- The best way to do this would be to use the `SpringBeanAutowiringSupport` class from within a `@PostConstruct` method of the JAX-RS class which contains `@Autowired` Spring components

```
public class QuoteService {  
    @Autowired  
    private QuoteGenerator generator;  
    @PostConstruct  
    public void initSpringComponents() {  
        SpringBeanAutowiringSupport.  
            processInjectionBasedOnCurrentContext(this);  
    }  
}
```



## 7.7 JAX-RS Annotations

- JAX-RS uses different annotations from Spring MVC
- `@Path` annotation for linking classes or methods to the request path that will map to them

```
@Path("/quotes")
```

```
public class QuoteService {
```

- Separate annotations for mapping to the various HTTP methods

```
@GET @POST @PUT @DELETE
```

- You can use a `@PathParam` annotation with a method parameter to extract data from the URL

```
@Path("/thisResource/{resourceId}")
```

```
public String getResourceById(@PathParam("resourceId") String id)  
{ ... }
```



## 7.8 Java Clients Using RestTemplate

- The most common types of clients for REST services are JavaScript and AJAX clients
- It is perhaps common also to need to have Java code communicate with REST services as a client
- Although REST service requests and responses are fairly simple using basic Java APIs like the `java.net` package would be difficult and low-level
- JAX-RS 2.1 provides a client API, but before then did not
- Spring provides the `RestTemplate` class which has a number of convenience methods for sending requests to REST services
  - ◇ These REST services do not need to be implemented using Spring MVC or JAX-RS
- To use the `RestTemplate` class in a Java client you would still need to include the Spring MVC module in the application





## 7.9 RestTemplate Methods

- There are many different methods and these are generally provided based on the HTTP method that will be sent
  - ◇ The parameters and return type differ depending on if the request body will contain data from an object and if the response body will contain data coming back
  - ◇ All take a String for URL and a variable argument Object... parameter for parameters in the URL

- DELETE methods are probably the simplest

```
void delete(String url, Object... urlVariables)
```

- GET takes no request body but returns an object for the response body

```
<T> getForObject(String url, Class<T> responseType, Object...  
urlVariables)
```

- PUT takes a request body but returns nothing

```
void put(String url, Object request, Object... urlVariables)
```

- POST takes a request body and can return a response body

```
<T> postForObject(String url, Object request, Class<T> responseType,  
Object... uriVariables)
```



## 7.10 Summary

- Spring MVC provides basic support for implementing REST services
- Spring could also be used behind standard JAX-RS REST services

## Chapter 8 - Introduction to Continuous Integration, Continuous Delivery and Jenkins-CI

---

### *Objectives*

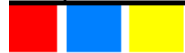
Key objectives of this chapter

- Agile Development
- Continuous Integration
- Continuous Delivery
- History of Jenkins

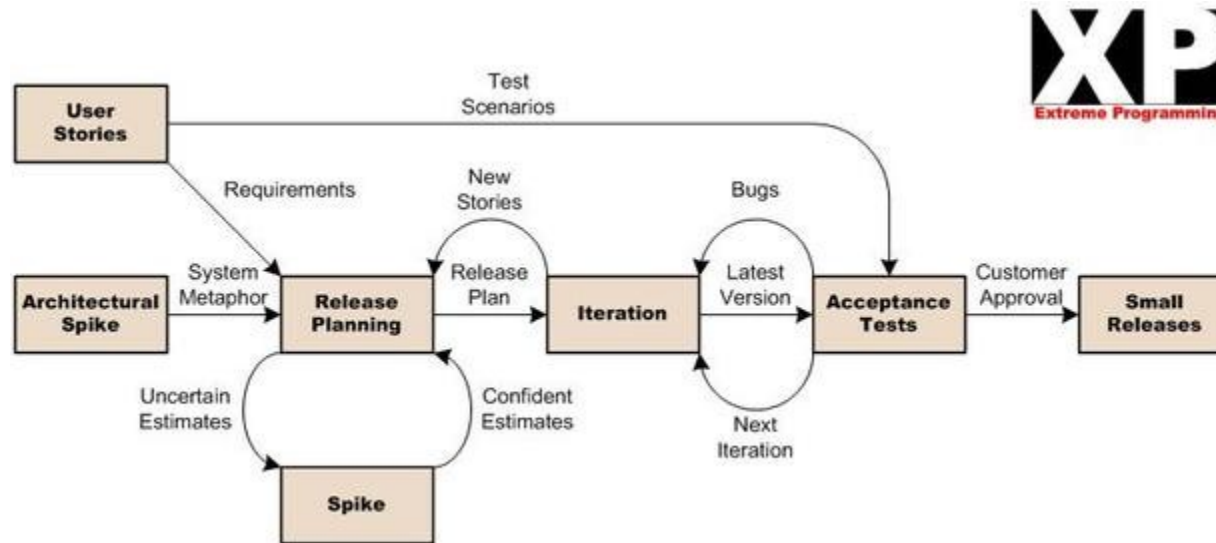


## 8.1 Foundation of Agile AppDev

- Agile Development begins with Extreme Programming (XP)
  - ◇ Invented/Promoted by Kent Beck and associates
- Beck describes XP as:
  - ◇ A philosophy of software development based on the values of communication, feedback, simplicity, courage, and respect
  - ◇ A body of practices proven useful in improving software development.
  - ◇ A set of complementary principles, intellectual techniques for translating the values into practice, useful when there isn't a practice handy for your problem.
  - ◇ A community that shares these values and many of the same practices.



## 8.2 XP Flow





## 8.3 Extreme Programming

- As time went on, people took inspiration from XP and developed other approaches with much the same goals
  - ◇ Speed
  - ◇ Include the Customer Early
  - ◇ Don't wait until you understand the entire problem - you'll understand it while you fix it
  - ◇ Deliver usable value at every iteration
  - ◇ Make reasonable use of modeling and development tools



## 8.4 Agile Development

- Two themes emerge from XP and Agile Development
  - ◇ Trust
    - Vendor, customer, managers, developers all need to trust each other
  - ◇ Automation
    - If we're going to go fast, we need tools that make it easy to produce software
- Trust is a social problem – solution is all about different approaches to process, requirements gathering, project management, etc.
- Automation is about tools – and Continuous Integration is a primary tool of Agile Development



## 8.5 What is Continuous Integration

- “Integrate and test changes after no more than a couple of hours.”
  - Beck & Andres “Extreme Programming Explained”
    - ◇ Integrate and build the complete product, atomic build product
    - ◇ If a website, build the website packaging, e.g. .NET, J2EE EAR
    - ◇ If a Javascript development, package the scripts and any configuration files





## 8.6 What is Continuous Integration (cont'd)

- Purposes
  - ◇ You find out quickly about integration problems
  - ◇ Immediately evident if a code change “breaks the build”
  - ◇ Prevents a long drawn-out integration step at the end of code changes
  - ◇ Should be complete enough that eventual first deployment of the system is “no big deal”.



## 8.7 What is Continuous Integration (cont'd)

- Usually, CI goes along with Test-First design and automated QA
  - ◇ Run the unit-test suite and smoke testing to ensure the build isn't broken
- Clearly, automatic build is a prerequisite
  - ◇ So we need a build system – Maven for instance
- Can be synchronous or asynchronous
  - ◇ Asynchronous – Integration happens automatically on code committal
  - ◇ Synchronous – Trigger the build manually after a development session

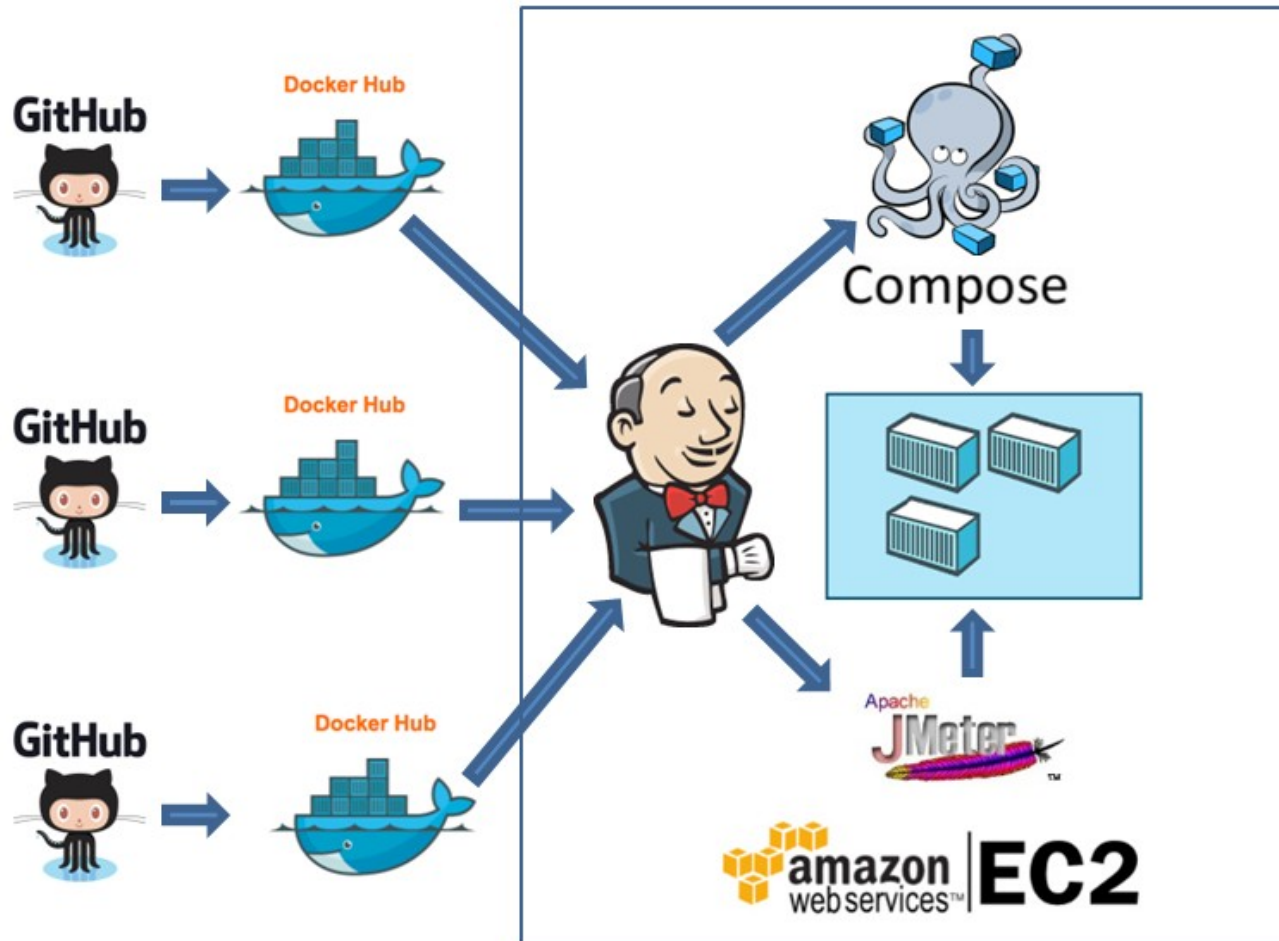


## 8.8 What is Continuous Integration (cont'd)

- Side Effects
  - ◇ Generate development reports
  - ◇ Install to QA, User Test, etc
  - ◇ Always have an install-able artifact
- It's a great time to generate development metrics
  - ◇ E.g. Code Coverage, Standards Compliance, Static Analysis



## 8.9 Typical Setup for Continuous Integration



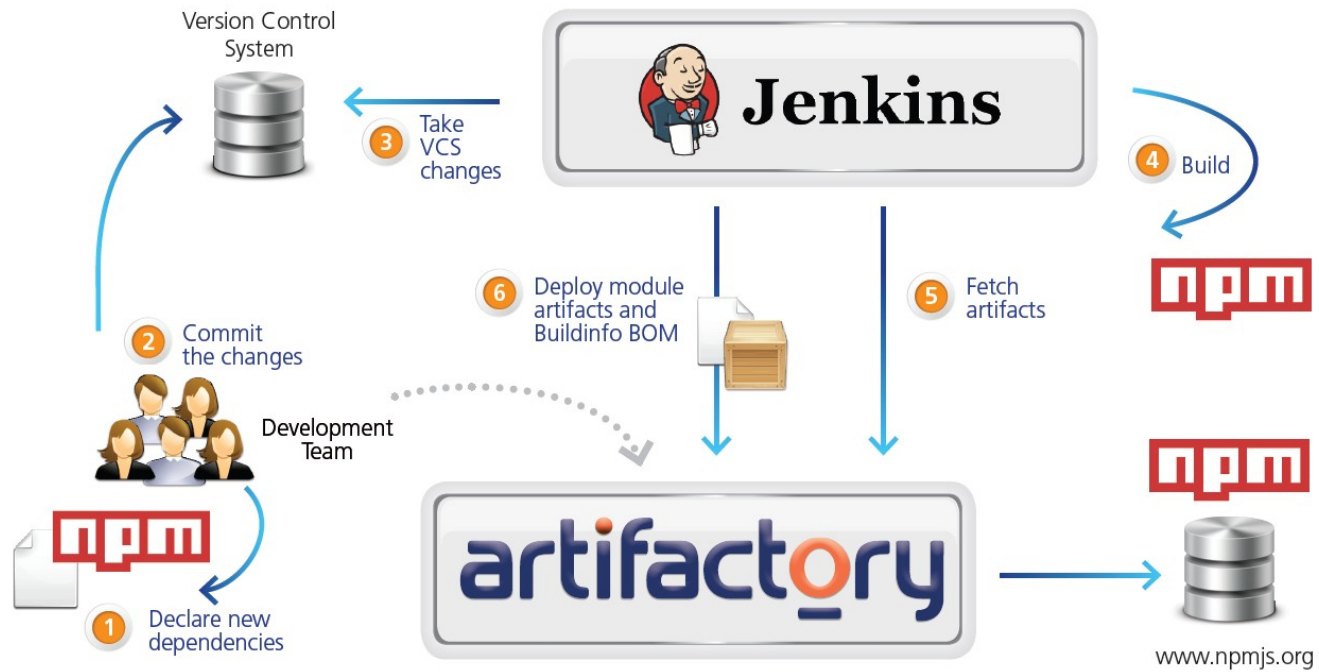


## 8.10 Setup Notes for Continuous Integration

- Notes:
  - ◇ CI system gets the code directly from version control
  - ◇ Build is independent of any local artifacts that are on the developer's machine
    - Controlled links to corporate repository and Maven Central
    - Goal is to ensure that the package can be built from the corporate repository
  - ◇ Jenkins can have connections to a deployment environment
    - Production Staging
    - User Acceptance Testing
    - QA
    - Load Test
    - Etc...
  - ◇ It turns out that if we use Maven, we ABSOLUTELY NEED a local repository manager
    - More info to follow...



## 8.11 CI with Artifact Management





## 8.12 What is Continuous Delivery?

- Using the definition from DevOps thought leader Martin Fowler
- Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.
- You're doing continuous delivery when:
  - ◇ Your software is deployable throughout its lifecycle
  - ◇ Your team prioritizes keeping the software deployable over working on new features
  - ◇ Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them



### 8.13 Why Continuous Delivery?

- ◇ CD extends CI into readiness for deployment and operations
- ◇ Agile/XP speeds up the development process
  - Include the Customer or Voice of the Customer
  - Ensure releasable artifact after every iteration
- ◇ Nonetheless, Agile/XP releases fit into the standard Software Development Life Cycle
  - A release engineering and deployment process follows the development process.
  - Reflects traditional split between development and operations





## 8.14 DevOps and Continuous Delivery

- ◇ Managing deployments and development across horizontally-scaled environments is complex
  - We have to deploy more than one server/image/container
  - Requires tight control of the process
  - Requires automation of development, deployment and monitoring
  - Along with the technologies of virtualization, cloud, etc.
- ◇ This is the central theme of DevOps - Integrated provision of services using appropriate technology and processes
- ◇ Deploy software as fast as we create it, while maintaining quality, traceability and accountability



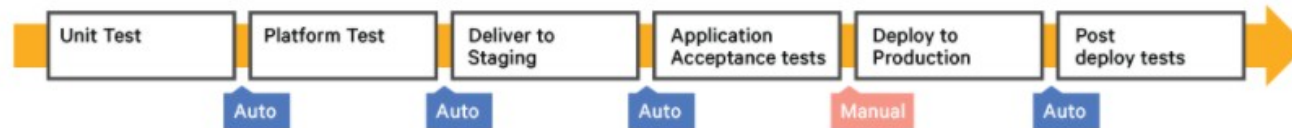
## 8.15 Continuous Delivery Challenges

- ◇ More than one department/group involved, not just Developers
  - QA, Compliance, Business Customers, etc
- ◇ CI job takes a few minutes, CD process could extend over days
  - Could also include human input, manual tests, acceptance tests, etc
- ◇ Extended cycle means we have multiple process instances "in flight"
  - "Version 6" part-way through User Acceptance Testing, while work continues on "Version 7"
- ◇ Multiple resources and test/deployment environments involved
  - "Version 6" goes on to Performance Validation
  - "Version 7" moves into User Acceptance Testing
  - Meanwhile, development gets going on "Version 8"

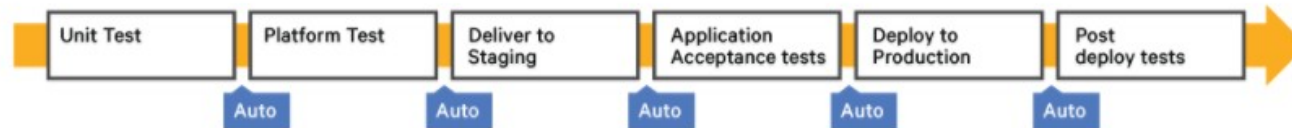


## 8.16 Continuous Delivery vs Continuous Deployment

### Continuous Delivery



### Continuous Deployment





## 8.17 Jenkins Continuous Integration

- Originally developed at Sun by Kohsuke Kawaguchi
  - ◇ Originally “Hudson” on java.net circa 2005
  - ◇ Jenkins forked in November 2010
  - ◇ Hudson is still live, part of Eclipse Foundation
  - ◇ But Jenkins seems to be far more active



## 8.18 Jenkins Features

- Executes jobs based on a number of triggers
  - ◇ Change in a version control system
  - ◇ Time
  - ◇ Manual Trigger
- A Job consists of some instructions
  - ◇ Run a script
  - ◇ Execute a Maven project or Ant File
  - ◇ Run an operating system command
- User Interface can gather reports
  - ◇ Each job has a dashboard showing recent executions
- Jenkins 2 adds the "Pipeline"
  - ◇ Orchestrate Continuous Delivery processes more easily



## 8.19 Running Jenkins

- You can run Jenkins Standalone or inside a web container
- You can setup distributed instances that cooperate on building software
- Can setup jobs in place of what might have been script commands.



## 8.20 Summary

- Continuous Integration is a powerful tool for agile software development
- Jenkins is currently the dominant Continuous Integration tool

## Chapter 9 - Installing and Running Jenkins

---

### *Objectives*

Key objectives of this chapter

- Downloading and Installing Jenkins.
- Running Jenkins as a Stand-Alone Application.
- Running Jenkins on an Application Server.
- Installing Jenkins as a Windows Service.
- Build Steps
- Reporting on Test Results
- Working with Maven Build Jobs





## 9.1 Downloading and Installing Jenkins

- Download Jenkins from the Jenkins website (<http://jenkins.io>)
  - Jenkins is a dynamic project, and new releases come out at a regular rate.
- Jenkins distribution is bundled in Java web application (a WAR file).
- For Windows users, there is a graphical Windows installation MSI package
- There are also native installers for several \*nix variations (Ubuntu, Red Hat, FreeBSD, Mac OSX, etc), as well as Docker images





## 9.2 Running Jenkins as a Stand-Alone Application

- Jenkins comes bundled as a WAR file that you can run directly with 'java -jar'.
- Jenkins uses a lightweight, embedded Servlet engine to allow you to run the server out of the box.
- Flexible to install plug-ins and upgrades on the fly.
- To run Jenkins using the embedded Servlet container, just go to the command line and type the following:

A screenshot of a Windows Command Prompt window. The title bar says "C:\> Command Prompt". The command prompt shows the directory "c:\Software" and the command "java -jar jenkins.war" being entered.

```
C:\> Command Prompt
c:\Software>java -jar jenkins.war
```

- The Jenkins web application will now be available on port 8080.
- Jenkins can be accessed directly using the server URL (<http://localhost:8080>).
- To stop Jenkins, just press Ctrl-C.
- Note: Requires a Java run-time environment - JDK 7 or higher, JDK 8 recommended.
  - ◇ Note - some Linux distributions ship with GCJ, which doesn't work with Jenkins. Install OpenJDK or an Oracle JDK instead.



### 9.3 Running Jenkins as a Stand-Alone Application (cont'd)

- Useful Options:
- --httpPort
  - By default, Jenkins will run on the 8080 port.
  - Jenkins can be start on different port using the --httpPort option:
  - `java -jar jenkins.war --httpPort=8081`
- --logfile
  - By default, Jenkins writes its logfile into the current directory.
  - Option to redirect your messages to other file:
  - `java -jar jenkins.war --logfile=C:/Software/log/jenkins.log`
- These options can also be set at JENKINS\_HOME/jenkins.xml config file.



## 9.4 Running Jenkins on an Application Server

- Jenkins distribution WAR file can be easily deploy to standard Java application server such as Apache Tomcat, Jetty, or GlassFish.
- Jenkins will be executed in its own web application context (typically “jenkins”).
  - URL : <http://localhost:8080/jenkins>.



## 9.5 The Jenkins Home Folder

- ◇ When run either as stand-alone or in an application server, Jenkins will create a "home" folder called '.jenkins'
- ◇ On Unix/Linux, this will be under the user that runs either Jenkins or the server
  - e.g. if Bob Smith runs Jenkins on his Mac,
    - /Users/smithb/.jenkins
  - e.g. Claire Jones runs Jenkins on Windows 8
    - C:\Users\claire\.jenkins



## 9.6 Installing Jenkins as a Windows Service

- ◇ In production, we want Jenkins to run as a Windows Service
- ◇ Jenkins will automatically start whenever the server reboots
- ◇ Can be managed using the standard Windows administration tools.
- ◇ Recent versions of Jenkins ship with an 'MSI'-style installer for Windows
  - Oddly, it's packed in a '.zip' file. Unpack the zip file to get the installer.
  - The installation includes a bundled Java Runtime Environment
- ◇ Typically installs to 'C:\Program Files (x86)\Jenkins', but you can override it.
- ◇ The same folder is used as the 'Jenkins Home' folder.
- ◇ Start/Stop the service through the service control panel.
- ◇ URL: <http://localhost:8080>
  - This can be configured in 'jenkins.xml' in the installation folder



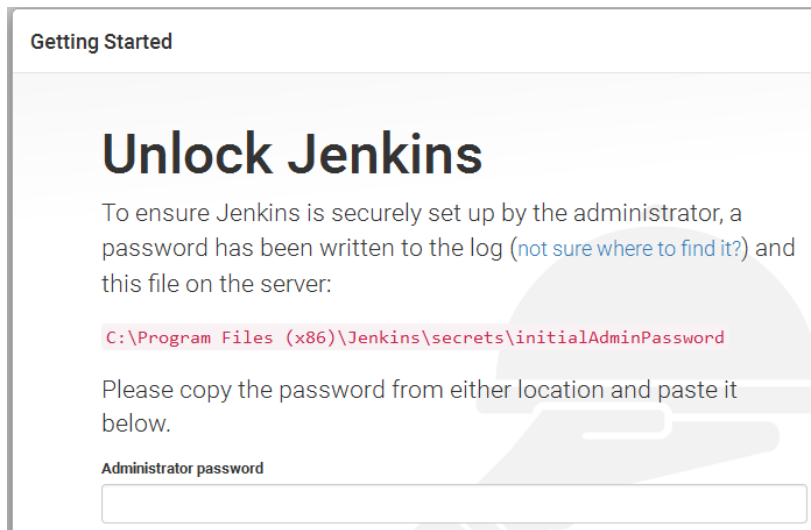
## 9.7 Initial Configuration

- ◇ When you first run Jenkins, it runs an installation wizard that forces you to:
  - Select a set of plugins
  - Setup an administrative user



## 9.8 Configuration Wizard

- ◇ When you access through 'http://localhost;8080' (i.e. Windows installer or embedded mode)



- ◇ It will ask for a one-time password, which you can get from the server console or the file indicated





## 9.9 Configuration Wizard (cont'd)

- ◇ Click on either "Install suggested Plugins" or "Select plugins to install"
  - Usually OK to start with the suggested plugins, unless you know your configuration needs customization

### Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

#### Install suggested plugins

Install plugins the Jenkins community finds most useful.

#### Select plugins to install

Select and install plugins most suitable for your needs.



## 9.10 Configuration Wizard (cont'd)

- ◇ As of Jenkins 2, Jenkins defaults to secure mode.
- ◇ Create an Administrative user

### Create First Admin User

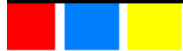
Username:	<input type="text" value="wasadmin"/>
Password:	<input type="password" value="....."/>
Confirm password:	<input type="password" value="....."/>
Full name:	<input type="text" value="Web Age Administrator"/>
E-mail address:	<input type="text" value="admin@example.com"/>

- ◇ That user can then do further configuration through the web interface
- ◇ We'll cover security setup and secure-http in a later module.



## 9.11 Configuring Tools

- ◇ Jenkins doesn't actually "build" anything - it uses tools
- ◇ e.g. Java, Maven, Ant, gulp, grunt, etc
- ◇ We might have multiple versions installed
  - A given job might only work with a particular version of Maven for instance
  - You might want to build/test against more than one version of Java
  - etc...
- ◇ Install locations can vary across different hosts
- ◇ Add tools through the web interface:
  - **Manage Jenkins --> Global Tool Configuration**
  - In general, need to supply the install location for each tool, along with an identifier



## 9.12 Configuring Tools - Best Practices

- ◇ Provide a tool identifier for general use
  - e.g. 'M3-Default' or 'M3-Latest'
  - Keep this one updated - most jobs should use it
- ◇ If you have jobs that need specific versions, then add the specific version

**Maven**

Maven installations

Maven	
Name	<input type="text" value="M3-Latest"/>
MAVEN_HOME	<input type="text" value="C:\Software\apache-maven-3.3.9"/>
<input type="checkbox"/> Install automatically	<a href="#">?</a>
<input type="button" value="Add Maven"/>	
<input type="button" value="Delete Maven"/>	

List of Maven installations on this system



## 9.13 Logging in Jenkins

- ◇ Jenkins uses the 'java.util.logging' package
  - Look in the web server's logging system
- ◇ Jenkins also sets up its own logger and keeps the data in memory
- ◇ Access the logs through **Manage Jenkins** → **System Log**
- ◇ For build job problems, read through the build's console output






## 9.14 Custom Log Recorders

- ◇ The system log records log messages to a memory-based ring buffer
- ◇ Default is to log everything
- ◇ You can create your own log recorders

Name  

Loggers

Logger	Log level	
<input type="text" value="jenkins.*"/>	<input type="text" value="SEVERE"/> 	 

List of loggers and the log levels to record



## 9.15 Summary

- ◇ Jenkins is a JEE Web Application
- ◇ Can run as an embedded web server, in a web server installation, or as a Windows Service
- ◇ Defaults to secure mode - some initial setup is required.
- ◇ You need to setup tools - Java, Maven, Ant, etc.

## Chapter 10 - Job Types in Jenkins

---

### *Objectives*

Key objectives of this chapter

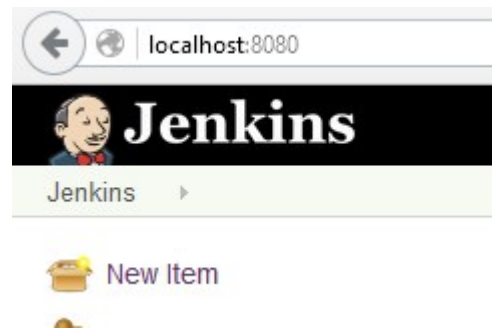
- Different types of Jenkins job.
- Configuring Source Code Management.
- Working with Subversion
- Build Triggers.
- Schedule Build Jobs.
- Maven Build Steps





## 10.1 Introduction

- ◇ Traditional Continuous Integration tasks are called "Jobs"
- ◇ More complicated tasks can be defined with the Pipeline, which we'll see later.
- ◇ Create a new job by clicking 'New Item' from the main menu





## 10.2 Different types of Jenkins Items

- ◇ Jenkins supports several different types of items at the top level
  - Freestyle software project:
    - Freestyle projects are general purpose and allow to configure any sort of build job.
    - Highly flexible and very configurable.
  - Maven project:
    - Installed as a plugin
    - Jenkins understands Maven pom files and project structures.
    - Reduce the work needed to do to set up the project.
  - Monitor an external job:
    - Monitoring the non-interactive execution of processes, such as cron jobs.




### 10.3 Different types of Jenkins Items (cont'd)

- Multi-configuration job:
  - Run same build job in many different configurations.
  - Powerful feature, useful for testing an application in many different environments.
- Folder:
  - aka the CloudBees Folders Plugin
  - Lets you group jobs in folders
  - Creates separate namespaces for jobs
- Pipeline
  - Runs an orchestration script that can have multiple steps and span restarts
- Multibranch Pipeline
  - Creates a pipeline for more than one branch in an SCM repository




## 10.4 Different types of Jenkins Items (cont'd)


**Enter an item name**  
  
» Required field




**Freestyle project**  
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.



**Maven project**  
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.



**Pipeline**  
Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.



**External Job**  
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.



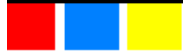
## **10.5 Configuring Source Code Management(SCM)**

- Monitors version control system, and checks out the latest changes as they occur.
- Compiles and tests the most recent version of the code.
- Simply check out and build the latest version of the source code on a regular basis.
- SCM configuration options in Jenkins are identical across all sorts of build jobs.
- Jenkins supports Git and Subversion out of the box (default plugins)
- Integrates with a large number of other version control systems via plugins.



## 10.6 Working with Subversion

- Simply provide the corresponding Subversion URL
  - Supported protocols http, svn, or file.
- Jenkins will check that the URL is valid as soon as you enter it.
- If authentication needed, Jenkins will prompt you for the corresponding credentials automatically, and store them for any other build jobs that access this repository.
- Fine-tune Jenkins to obtain the latest source code from your Subversion repository by selecting an appropriate value in the Check-out Strategy drop-down list.
- Choose check-out Strategy as “Use ‘svn update’ as much as possible, with ‘svn revert’ before update”
  - No local files are modified, though it will not remove any new files that have been created during the build process.
  - You might want other options, depending on the load on your svn server.



## 10.7 Working with Subversion (cont'd)

### Source Code Management

☐ CVS

☐ None

☒ Subversion

Modules

Repository URL

 Repository URL is required.

Local module directory (optional)

Check-out Strategy

Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser

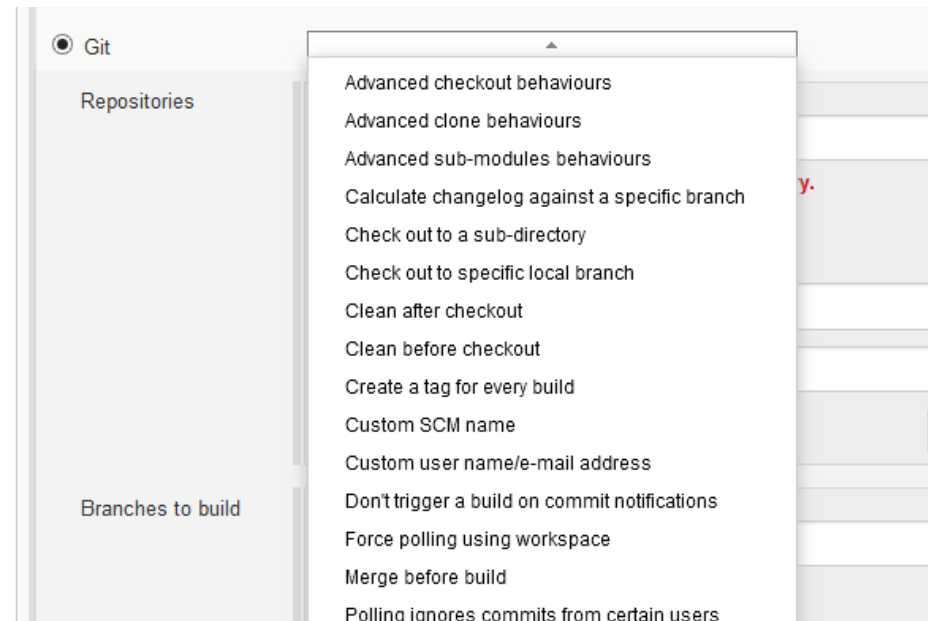


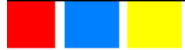


## 10.8 Working with Git

- Git support is in a plugin, but is installed by default if you select "Install suggested plugins" at initial setup.
- Any type of remote repository can be used:
  - ◇ Could be https, ssh, or local
- Jenkins will check that the URL is valid as soon as you enter it.
- You can store ssh credentials or http credentials
- There are a wide variety of other "Additional Checkout Behaviors" that are possible

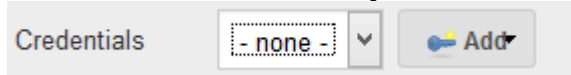






## 10.9 Storing Credentials

- ◇ Usually, SCM systems require login
- ◇ The values we use for login are called "Credentials" and can be managed centrally
- ◇ As a convenience, you can add them directly from the job configuration screen



- ◇ Click 'Add' and then select 'Jenkins Credential Provider'



## 10.10 Service Accounts

- As a general rule, you don't want users to put their own credentials into Jenkins jobs or other setups
- The correct thing to do is create accounts for Jenkins
  - ◇ i.e. "Service Accounts"
- Jenkins then has an identity that it uses to access things
- Administration personnel maintain the credentials in Jenkins for the service account
- No users can see or download the stored credentials



## 10.11 Storing Credentials (cont'd)

- ◇ You can store several types of credentials
  - User name/Password
  - SSH Key
  - Secret file
  - Secret text
  - X.509 Certificate
  - Other plugins are available



## 10.12 Build Triggers

- In a Freestyle build, there are three basic ways a build job can be triggered
  - Start a build job once another build job has completed
  - Kick off builds at periodic intervals
  - Poll the SCM for changes

A screenshot of the Jenkins 'Build Triggers' configuration panel. The panel has a title 'Build Triggers' in bold. Below the title, there are several checkboxes with corresponding labels. The first checkbox is checked and labeled 'Build whenever a SNAPSHOT dependency is built'. The second checkbox is unchecked and labeled 'Schedule build when some upstream has no successful builds'. The third checkbox is unchecked and labeled 'Trigger builds remotely (e.g., from scripts)'. The fourth checkbox is unchecked and labeled 'Build after other projects are built'. The fifth checkbox is unchecked and labeled 'Build periodically'. The sixth checkbox is unchecked and labeled 'Build when a change is pushed to GitHub'. The seventh checkbox is unchecked and labeled 'Poll SCM'.

**Build Triggers**

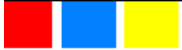
- ☒ Build whenever a SNAPSHOT dependency is built
- ☐ Schedule build when some upstream has no successful builds
- ☐ Trigger builds remotely (e.g., from scripts)
- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ Build when a change is pushed to GitHub
- ☐ Poll SCM



### 10.13 Schedule Build Jobs

- Build job at regular intervals.
- For all scheduling tasks, Jenkins uses a cron-style syntax, consisting of five fields separated by white space in the following format:
  - ◇ MINUTE : Minutes within the hour (0–59)
  - ◇ HOUR : The hour of the day (0–23)
  - ◇ DOM : The day of the month (1–31)
  - ◇ MONTH : The month (1–12)
  - ◇ DOW : The day of the week (0–7) where 0 and 7 are Sunday.
- There are also a few short-cuts:
  - ◇ “\*” represents all possible values for a field. For example, “\* \* \* \* \*” means “once a minute.”
  - ◇ You can define ranges using the “M–N” notation. For example “1-5” in the DOW field would mean “Monday to Friday.”
  - ◇ You can use the slash notation to defined skips through a range. For example, “\*/5” in the MINUTE field would mean “every five minutes.”
  - ◇ A comma-separated list indicates a list of valid values. For example, “15,45” in the MINUTE field would mean “at 15 and 45 minutes past every hour.”





## 10.14 Polling the SCM

- Poll SVN or Git server at regular intervals if any changes have been committed.
- The more frequent the polling is, the faster the build jobs will start, and the more accurate.
  - ◇ But, there's more load on the SCM server
- In Jenkins, SCM polling is easy to configure, and uses the same cron syntax we discussed previously.





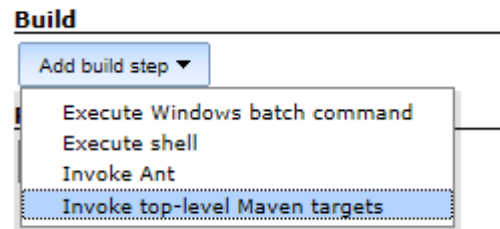
## 10.15 Polling vs Triggers

- Many SCMs can have "hooks" added
  - ◇ For instance a script can be run when changes are pushed to a repository
- You could use a hook to trigger a build
  - ◇ Typical way is to use "curl" to initiate a build
  - ◇ In the project configuration, enable "Trigger builds remotely"
    - Jenkins will show you a URL that triggers the build
- Triggering may be more work to setup
- Polling may be more overhead on the SCM repository
  - ◇ Especially if you have a large number of Jenkins jobs
- You probably need to experiment!

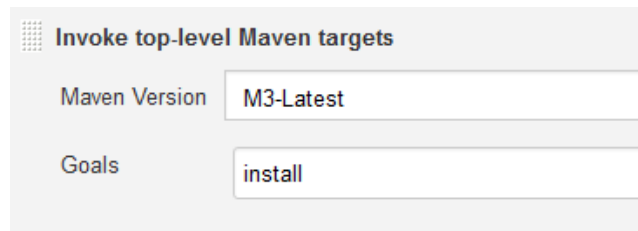


## 10.16 Maven Build Steps

- Jenkins has excellent Maven support, and Maven build steps are easy to configure and very flexible.
- Select “Invoke top-level Maven targets” from the build step lists.



- Select a version of Maven to run (if you have multiple versions installed)
- Enter the Maven goals you want to run. Jenkins freestyle build jobs work fine with both Maven 2 and Maven 3.
- The optional POM field lets you override the default location of the Maven pom.xml file.





## 10.17 Summary

- Learned how Jenkins checks out the source code of a project.
- How to set-up and schedule a job.
- Setup build for Maven project.

## Chapter 11 - Continuous Delivery and the Jenkins Pipeline

---

### *Objectives*

Key objectives of this chapter

- Continuous Delivery
- The Jenkins Pipeline
- A Brief Introduction to Groovy
- The JenkinsFile
- Pipeline Jobs

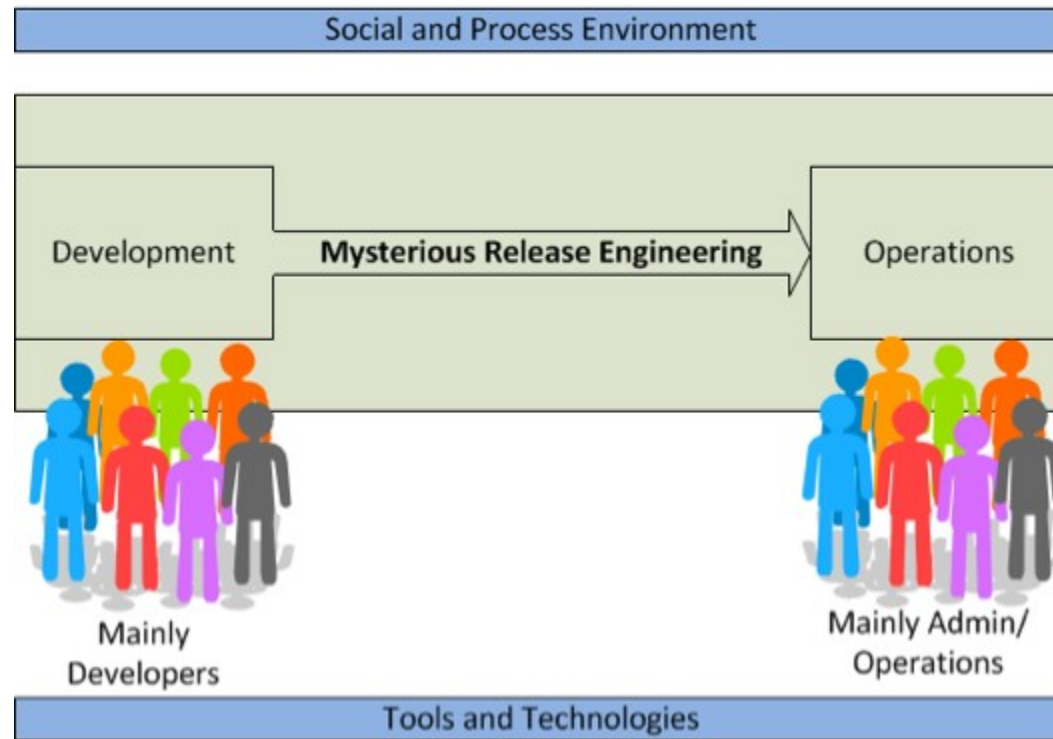


## 11.1 Continuous Delivery

- ◇ CD extends CI into deployment and operations
- ◇ Agile/XP speeds up the development process
  - Include the Customer or Voice of the Customer
  - Ensure releasable artifact after every iteration
- ◇ Nonetheless, Agile/XP releases fit into the standard Software Development Life Cycle
  - A release engineering and deployment process follows the development process.
  - Reflects traditional split between development and operations



## 11.2 Continuous Delivery (cont'd)





### 11.3 DevOps and Continuous Delivery

- ◇ Managing deployments and development across horizontally-scaled environments is difficult
  - Requires tight control of the process
  - Requires automation of development, deployment and monitoring
  - Along with the technologies of virtualization, cloud, etc.
- ◇ This is the central theme of DevOps - Integrated provision of services using appropriate technology and processes
- ◇ Deploy software as fast as we create it, while maintaining quality, trace-ability and accountability



## 11.4 Continuous Delivery Challenges

- ◇ More than one department/group involved, not just Developers
  - QA, Compliance, Business Customers, etc
- ◇ CI job takes a few minutes, CD process could extend over days
  - Could also include human input, manual tests, acceptance tests, etc
- ◇ Extended cycle means we have multiple process instances "in flight"
  - "Version 6" part-way through User Acceptance Testing, while work continues on "Version 7"
- ◇ Multiple resources and test/deployment environments involved
  - "Version 6" goes on to Performance Validation
  - "Version 7" moves into User Acceptance Testing
  - Meanwhile, development gets going on "Version 8"





## 11.5 Continuous Delivery with Jenkins

- ◇ Jenkins has always been useful for this
  - A job completed successfully can trigger another job
  - Jobs can be run on different machines
  - Parameterized jobs can gather information from users
  - Plugins can manage deployment
- ◇ Workflow processing has been difficult though
  - Builds in-process don't persist across restart
  - No logic in linking jobs
  - No unified user interface
  - Job definitions are not in Version Control
- ◇ Solution: The Pipeline Plugin



## 11.6 The Pipeline Plugin

- ◇ Represents a workflow
- ◇ More like a business process in SOA
  - whereas a "Job" is more like a "Service Operation"
- ◇ Define a process and then execute an "instance"
- ◇ Completion of the process can span more than one machine
- ◇ Process has "State"
  - "where are we in the process"
  - Process variables
- ◇ State is stored persistently
  - Also distributed when the build uses a different machine



## 11.7 The Pipeline Plugin (cont'd)

- ◇ Steps can be executed conditionally
- ◇ Can have explicit parallelism
  - And can span machines
- ◇ Definition of Pipeline can be stored in Version Control
  - Including all the steps that would make traditional "jobs"
- ◇ User Interface
  - lets us see multiple instances
  - See processes that span distributed installations
  - Interact with the process (input and output)



## 11.8 Defining a Pipeline

- ◇ The pipeline is defined by a script written in the Groovy programming language
  - <http://groovy-lang.org>
- ◇ The script defines "steps" that the Pipeline plugin executes
- ◇ Steps:
  - built-in steps like "checkout" , "sh" or "bat"
  - Calls to plugins
  - Calls to existing jobs



## 11.9 A Pipeline Example

```
node {  
    stage 'Checkout'  
    checkout scm  
  
    stage 'Build'  
    bat 'nuget restore SolutionName.sln'  
    bat "\"${tool 'MSBuild'}\" SolutionName.sln"  
  
    stage 'Archive'  
    archive 'ProjectName/bin/Release/**'  
  
}
```



### 11.10 Pipeline Example (cont'd)

- ◇ 'node' indicates a section that should be run on an Agent machine
- ◇ 'stage' defines a pipeline stage
  - for presentation in the UI
- ◇ 'checkout' invokes the SCM plugin to checkout a copy of the project on the node in question
  - Note: this file would be stored in version control and checked-out by the Jenkins master. 'checkout' is checking out the same project on the execution node
- ◇ 'bat' runs a Windows batch file
- ◇ 'archive' stores the build results



## 11.11 Parallel Execution

- ◊ Sections of the pipeline can be executed in parallel

```
def labels = ['precise', 'trusty'] // labels for Jenkins node
types we will build on
def builders = [:]
for (x in labels) {
    def label = x // Need to bind the label variable before the
closure - can't do 'for (label in labels)'

    // Create a map to pass in to the 'parallel' step so we can
fire all the builds at once
    builders[label] = {
        node(label) {
            // build steps that should happen on all nodes go here
        }
    }
}

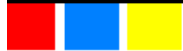
parallel builders
```



## 11.12 Creating a Pipeline

- ◇ Various ways
  - From the UI, like a Job
    - Enter pipeline script into the web interface
    - This is nice because of syntax help available
  - In a "Jenkinsfile"
    - Stored in version control
    - Could be in the same repository as a project, or in its own repository
  - Commonly-used scripts and functions can be stored in a "Global Library"
    - Jenkins runs an instance of 'Git' to let you push global libraries that are then available to all Pipeline scripts.
  - Libraries can be stored in the project and loaded as needed.





## 11.13 Invoking the Pipeline

- ◇ The pipeline shows up just like a regular Jenkins job

All		+		
S	W	Name ↓	Last Success	Last Failure
		<a href="#">ReleasePipeline</a>	8 hr 18 min - <a href="#">#3</a>	8 hr 22 min - <a href="#">#2</a>
		<a href="#">SimpleGreeting</a>	2 days 9 hr - <a href="#">#4</a>	N/A

Icon: [S](#) [M](#) [L](#)

[Legend](#) [RSS for all](#) [RSS for fai](#)

- It can be invoked manually, at a predefined interval, or based on SCM polling
  - ◇ Just like a regular job

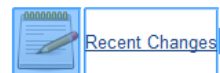


## 11.14 Interacting with the Pipeline

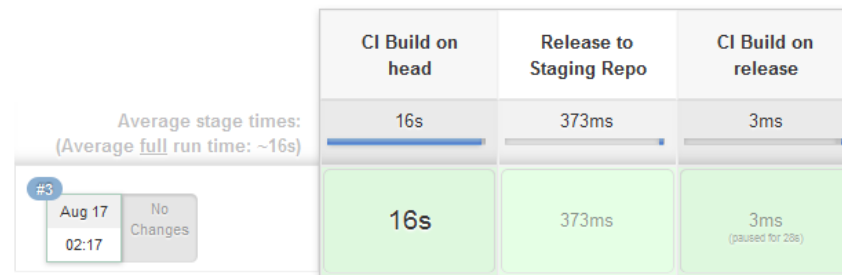
- ◇ Stages in the pipeline appear on the user interface

### Pipeline ReleasePipeline

[add description](#)



### Stage View



- ◇ In case of input required or output, these appear on the UI



### 11.15 Pipeline vs Traditional Jobs

- ◇ In the long term, we expect that use of traditional (config.xml) jobs will be replaced by Pipeline jobs
- ◇ 'config.xml' is not stored in version control
  - It is possible to manually store it, but Jenkins doesn't know anything about versioning the config.xml
- ◇ Libraries allow the same process to be applied to multiple jobs
- ◇ Plugins like the "BitBucket Branch Source Plugin" and the "GitHub Branch Source Plugin" let administrators basically apply the same job setup to all the repositories in a given organization
  - Under this practice, you get a CI job/jobs automatically when the repository is created



## 11.16 Conclusion

- ◇ The Pipeline plugin turns Jenkins into an Orchestration system
- ◇ Continuous Delivery processes may span extended time periods
- ◇ Pipeline definition is in the form of a Groovy script
- ◇ The Jenkins UI lets you interact with the Pipeline

## Chapter 12 - Groovy DSL [OPTIONAL]

---

### *Objectives*

Key objectives of this chapter

- Basic Groovy Syntax
- Defining Functions
- Defining Classes
- Classes vs Scripts
- Defining Shared Libraries
- Using Shared Libraries



## 12.1 What is Groovy

- Groovy is an object oriented language
- It is based on Java platform.
- First released in 2007.
- It is distributed via the Apache License.



## 12.2 Groovy in Jenkins

- Groovy can be executed in Jenkins in various ways
  - ◇ Script Console (<http://localhost:8080/script>)
  - ◇ Execute system groovy script (Build step)
    - Runs in Jenkins' JVM.
    - Jenkins' system is available to the scripts
  - ◇ Execute groovy script (Build step)
    - Runs outside Jenkins' JVM.
    - Jenkins' system isn't available to scripts
  - ◇ Groovy PostBuild Plugin
- Pipeline Job Command



## 12.3 Comments in Groovy

- Single-line comment
  - ◇ //
- Multi-line comment
  - ◇ /\*
  - ◇ \*/





## 12.4 Data Types

- Built-in types
  - ◇ byte
  - ◇ short
  - ◇ int
  - ◇ long
  - ◇ float
  - ◇ double
  - ◇ char
  - ◇ Boolean
  - ◇ String
- Object types
  - ◇ java.lang.Byte
  - ◇ java.lang.Short
  - ◇ java.lang.Long
  - ◇ java.lang.Float



- ◇ `java.lang.Double`
- ◇ `java.math.BigInteger`
- ◇ `java.math.BigDecimal`



## 12.5 Identifiers

- Identifiers are used to define variables, functions, or other user defined objects.
- Identifiers start with a letter, a dollar, or an underscore.
- They cannot start with a number.
- e.g.
  - ◇ `customer123name`, `_customer123Name`, `customer_name123`



## 12.6 Variables

- Case-sensitive
- Variable declarations
  - ◇ `String message = "Hello World";`
  - ◇ `int a = 5;`
- Printing variables
  - ◇ `println message;`
  - ◇ `println(message);`
  - ◇ `print message;`
  - ◇ `print(message);`



## 12.7 def

- def keyword can be used to define an identifier
- When using def, the actual type holder is **Object** (so you can assign any object to variables defined with def, and return any kind of object if a method is declared returning def)
- e.g.
  - ◇ `def x = 5;`



## 12.8 String Interpolation

- String interpolation requires double quotes.

```
String name = "Bob";  
String message = "Hi, ${name}";
```



## 12.9 Operators

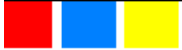
- Arithmetic
  - ◇ +, -, \*, /, %, ++, --
- Relational
  - ◇ ==, !=, <, <=, >, >=
- Logical
  - ◇ &&, ||, !
- Bitwise
  - ◇ &, |, ^, ~
- Assignment
  - ◇ +=, -=, \*=, /=, %=
- Range
  - ◇ ..
  - ◇ e.g. `def range = 5..10`
  - ◇ `println(range); //prints [5,6,7,8,9,10]`
  - ◇ `println(range.get(2)); //prints 7`



## 12.10 Ranges

- Range examples
  - ◇ 1..5 (inclusive range)
  - ◇ 1..<5 (exclusive range)
  - ◇ 5..1 (descending order)
  - ◇ 'a'..'f' (character range)
  - ◇ 'f'..'a' (descending order)
- Methods
  - ◇ contains(val)
  - ◇ get(pos)
  - ◇ size()
  - ◇ subList(fromIndex, toIndex)





## 12.11 Conditional Statements

- **if**

```
if (a < 5) {  
    println("A is less than 5");  
}
```

- **if / else**

```
if(a < 5) {  
    println("A is less than 5");  
}  
else if(a == 5) {  
    println("A is equal to 5");  
}  
else {  
    println("A is greater than 5");  
}
```

- **switch**

```
switch(a) {  
    case 1:  
        println("A is 1");  
        break;  
    case 2:
```



```
        println("A is 2");  
        break;  
    default:  
        println("Unknown");  
        break;  
}
```



## 12.12 Loops

- **for**

```
for(int a = 0; a < 10; a++) {  
    println(a);  
}
```

- **for-in**

```
String[] teams = ["Flames", "Maples", "Oilers" ];  
for(String team in teams) {  
    println(team);  
}
```

- **while**

```
while(a < 5) {  
    println(a);  
    a++;  
}
```

- **Additional keywords**

- ◊ **break**

```
while(a < 5) {  
    println(a);  
    a++;  
}
```



```
    if(a == 3) {  
        break;  
    }
```

```
}
```

◊ **continue**

```
while(a < 5) {  
    println(a);  
    a++;  
    if(a == 3) {  
        continue;  
    }  
}
```



## 12.13 Lists

- List examples
  - ◇ `[]` // an empty list
  - ◇ `[1, 2, 3, 4, 5]`
  - ◇ `[1, 2, [3, 4], 5]` // a nested list
  - ◇ `['apple', 'banana', 'cherry']` // a list of strings
  - ◇ `[1, 'apple', 2, 'banana']` // heterogeneous list
- Methods
  - ◇ `list.add(val);`
  - ◇ `list.remove(val);`
  - ◇ `list.contains(val);`
  - ◇ `list.get(3);`
  - ◇ `list.isEmpty();`
  - ◇ `list.minus(anotherList);`
  - ◇ `list.plus(anotherList);`
  - ◇ `list.pop();`



- ◇ `list.reverse();`
- ◇ `list.size();`
- ◇ `list.sort();`



## 12.14 Maps

- Dictionary / table / hash
- Unordered collection of object references
- key / value pair
- e.g.
  - ◇ `[ : ]` // an empty map
  - ◇ `teams = ['Calgary': 'Flames', 'Toronto': 'Maples']`
- Processing all items in a map

```
for(team in teams) {  
    println(team);  
}
```

- Methods
  - ◇ `map.containsKey(key);`
  - ◇ `map.get(key);`
  - ◇ `map.put(key, value);`
  - ◇ `map.size();`



## 12.15 Exception Handling

### ■ try .. catch .. finally

```
try {
    def arr = [1,2,3];
    def item = arr[7];
    println(item);
}
catch (ArrayIndexOutOfBoundsException ex) {
    println(ex.getMessage());
}
catch (Exception ex) {
    println ("Some other exception");
}
finally {
    println("always executed");
}
```





## 12.16 Methods

- A method is defined with a return type or with the **def** keyword.
- e.g.

```
def sum(def a, def b) {  
    return a + b;  
}
```

```
def result = sum(5,6);
```

- **Default parameters**

```
def sum(def a = 5, def b = 6) {  
    return a + b;  
}
```

```
def result1 = sum(5);
```

```
def result2 = sum();
```

- **Return keyword and semi-colon are optional, but recommended**

```
def sum(def a, def b) {  
    a + b  
}
```



## 12.17 Closures

- e.g.

```
def closure = {println "Hello world"};
closure.call();
```

- Parameters in closure

```
def closure = {param -> println "Hello ${param}"};
closure.call("World");
```

- Multiple parameters in closure

```
def sumClosure = {num1, num2 -> return num1 + num2};
closure.call(5,6);
```



## 12.18 this Keyword

- Used for accessing instance-level variable.

```
class Customer {  
    String name;  
    public void SetName(String value) {  
        this.name = value;  
    }  
    public String GetName() {  
        return this.name;  
    }  
}
```



## 12.19 Classes

- A Groovy class is a collection of data and the methods that operate on that data
- A class declares the state (data) and the behavior (methods) of objects defined by that class.
- Getter and setter methods are used to implement encapsulation
- e.g.

```
class Customer {  
    String name;  
    public void SetName(String value) {  
        this.name = value;  
    }  
    public String GetName() {  
        return this.name;  
    }  
}
```

- Instance creation

```
def cus1 = new Customer();
```



## 12.20 Static Methods

- Class level methods
- e.g.

```
class MyClass {  
    static def MyMethod() {  
        println("Static method");  
    }  
}
```



## 12.21 Inheritance

- **extends** keyword is used to inherit the properties of a class.

```
class Person {
    String name;
    public Person() {
        this.name = "";
    }
    public void SetName(String value) {
        this.name = value;
    }
    public String GetName() {
        return this.name;
    }
}

public class Student extends Person {
    public Student() {
        super();
    }
}
```



## 12.22 Abstract Classes

- Abstract classes represent generic concepts
- They cannot be instantiated
- They must be sub-classed
- e.g.

```
abstract class Person {  
    public String name;  
    abstract void DisplayDetails();  
}  
  
public class Customer extends Person {  
    public Customer() {  
        super();  
    }  
    void DisplayDetails() {  
        println "Details...";  
    }  
}
```



## 12.23 Interfaces

- Defines a contract that a class needs to conform to.

```
interface Vehicle {  
    void Start();  
}  
  
class Car implements Vehicle {  
    void Start() {  
        println("Car.Start()");  
    }  
}
```





## 12.24 Generics

- Generalized classes
- Generic for Collections

```
def list = new ArrayList<String>();  
list.add("A");
```

- Generalized Classes

```
public class MyClass<T> {  
    private T localVariable;  
    public T getVariable() {  
        return this.localVariable;  
    }  
    public void set(T value) {  
        this.localVariable = value;  
    }  
}
```



## 12.25 Jenkins Script Console

- Manage Jenkins > Script Console
- Allows execution of adhoc scripts.
- Sample script:

```
def printFile(location) {  
    pub = new File(location)  
    if (pub.exists()){  
        println "Location ${location}"  
        pub.eachLine{line-> println line}  
    }  
    else{  
        println "${location} does not exist"  
    }  
}  
  
printFile("C:/Windows/System32/drivers/etc/hosts")
```



## 12.26 Extending with Shared Libraries

- Parts of Pipelines can be shared between various projects to reduce redundancies and keep code DRY (Don't Repeat Yourself).
- A Shared Library is defined with a name, a source code retrieval method such as by SCM, and optionally a default version.
- Version can be anything understood by the SCM, e.g. branches, tags, and commit hashes.
- Library can be loaded implicitly or explicitly



## 12.27 Directory Structure

```
(root)
+- src                # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy # for org.foo.Bar class
+- vars
|   +- foo.groovy        # for global 'foo' variable
|   +- foo.txt           # help for 'foo' variable
+- resources           # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json  # static helper data for org.foo.Bar
```

- Can be defined at various levels
- src directory should look like standard Java source directory structure. This directory is added to the classpath when executing Pipelines.
- The vars directory hosts scripts that define global variables accessible from Pipeline. The basename of each \*.groovy file should be a Groovy identifier, conventionally camelCased. The matching \*.txt can contain documentation.
- The resources directory allows usage from an external library to load associated non-Groovy files.



## 12.28 Sample Groovy Code

```
package com.abcinc;

def checkout() {
    node {
        stage 'Checkout'
        git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }
}
```



## 12.29 Defining Shared Libraries

- Shared libraries can be defined at various levels:
  - ◇ Global Shared Libraries
    - Manage Jenkins > Configure System > Global Pipeline Libraries
    - Folder-level Shared Libraries
    - Automatic Shared Libraries

The screenshot shows the Jenkins 'Library' configuration page. The 'Name' field is set to 'my-shared-library'. The 'Default version' field is empty. The 'Load implicitly' checkbox is unchecked. The 'Allow default version to be overridden' checkbox is checked. Under the 'Retrieval method' section, 'Modern SCM' is selected. Under the 'Source Code Management' section, 'Git' is selected. The 'Project Repository' field contains 'git://git.example.com/pipeline-library.git'. The 'Credentials' dropdown is set to 'none', with an 'Add' button next to it. The 'Ignore on push notifications' checkbox is unchecked. The 'Repository browser' dropdown is set to '(Auto)'.

- e.g. GitHub Organization Folder



## 12.30 Using Shared Libraries

- Shared libraries can be utilized in various places
  - ◇ Pipeline
  - ◇ Execute system Groovy script
  - ◇ Execute Groovy script
  - ◇ Execute PostBuild script
- Loading libraries explicitly

```
@Library('my-shared-lib') _  
@Library('my-shared-lib@master') _  
@Library(['my-shared-lib', 'another-shared-lib']) _
```

- Conventionally the annotation goes on an **import** statement

```
@Library('my-shared-lib')  
import com.abcinc.utils;
```



### 12.31 Same Shared Library Usage Code

```
@Library('my-shared-lib')
import com.abcinc.utils;

def u = new utils();
u.checkout();
```





## 12.32 Defining Global Variables

```
// vars/acme.groovy
def setName(value) {
    name = value;
}
def getName() {
    return name;
}

// src/com/abcinc/sample.groovy
def myFunction() {
    name = "Bob";
}
```



## 12.33 Summary

- Groovy's syntax is similar to Java
- Semi-colon is optional in end of statement
- **return** keyword is optional in a method
- **def** keyword can be used to declare variables, as method return type, and for method input parameters.
- Groovy is used to define shared libraries
- In Jenkins shared libraries can be defined at various levels
- Shared libraries can be loaded implicitly or explicitly using @Library annotation.

## Chapter 13 - Securing Jenkins

---

### *Objectives*

Key objectives of this chapter

- Overall view of Jenkins Security
- Authentication Options
- Authorization



## 13.1 Jenkins Security - Overview

- Jenkins is used to build software
  - ◇ Source code is available in the workspace
- Code built with Jenkins is frequently “unreleased” development or test builds
  - ◇ Need to make sure only authorized users can access this software
- Jenkins can be used to execute admin jobs
  - ◇ Prevent access to prevent breaches and denial-of-service
- Jenkins is highly configurable
  - ◇ So we need to prevent accidental configuration changes



## 13.2 Jenkins Security

- ◇ Security breaks down into separate issues:
  - Authentication
  - Authorization
  - Confidentiality



## 13.3 Authentication

- ◇ The task of identifying a user
- ◇ Authentication happens against a user registry
- ◇ Jenkins can use LDAP, operating system, servlet container, or internal registry



## 13.4 Authorization

- ◇ Authorization is the task of deciding what operations a user can perform, once they are authenticated
- ◇ Jenkins allows a few different options for authentication
  - Anyone can do anything
  - Authorized Users can do anything
  - Permissions Matrix
  - Project-Based Permissions Matrix



## 13.5 Confidentiality

- ◇ Making sure that unauthorized parties don't get information they shouldn't have
- ◇ Two aspects in Jenkins
  - Control access to projects (really authentication)
  - Web confidentiality
    - Use secure http
    - Handled by the servlet container or httpd





## 13.6 Activating Security

- ◇ Manage Jenkins --> Configure Global Security
- ◇ Click the “Enable Security” checkbox
  - Jenkins will then display other options



### **Configure Global Security**

☒ Enable security



## 13.7 Configure Authentication

- ◇ Authentication is performed within a “realm”
- ◇ Possibilities
  - Use the servlet container's authentication
    - Different containers can be setup for different authentication techniques
  - Jenkins' own User Database
  - LDAP
  - Unix User/Group Database
    - Using Pluggable Authentication Module (PAM)



## 13.8 Using Jenkins's Internal User Database

- ◇ In the “Configure Global Security” screen, select “Jenkins own user database”

### Security Realm

---


- ☐ Delegate to servlet container
- ☒ Jenkins' own user database
- ☒ Allow users to sign up
- ☐ LDAP
- ☐ Unix user/group database

- ◇ Choose whether users can “sign up” for a user id
  - Otherwise, administrator needs to create users



## 13.9 Creating Users

- ◇ Manage Jenkins -> Manage Users -> Create User



Back to Dashboard

Manage Jenkins

Create User

### Sign up

Username:

Password:

Confirm password:

Full name:

E-mail address:



## 13.10 Authorization

- ◇ Five options

### Authorization

---

- ☒ Anyone can do anything
- ☐ Legacy mode
- ☐ Logged-in users can do anything
- ☐ Matrix-based security
- ☐ Project-based Matrix Authorization Strategy



## 13.11 Authorization

- ◇ Typically we want “Matrix-based security” or “Project-based Matrix Authorization Strategy”
- ◇ Authorization is separate from the authentication system
  - Need to enter userid or group entries separately from the users



## 13.12 Matrix-Based Security

- Each user is granted a set of permissions that apply to various operations

☒ Matrix-based security

User/group	Overall					Credentials				
	Administer	Read	RunScripts	UploadPlugins	ConfigureUpdateCenter	Create	Update	View	Delete	Manag
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

User/group to add:

- Add an entry for the user id, then select the appropriate check boxes



### **13.13 Note – Create the Administrative User**

- ◇ Create an administrative user at the same time you turn on security
  - Grant full permissions to this user
- ◇ If there's no admin user, but you've turned on matrix permissions, then no-one has permissions
  - You're locked out!
- ◇ See notes for recovery from lock-out






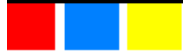
## 13.14 Project-based Matrix Authorization

- ◇ Same global user/permission grid
- ◇ You still need an administrative user
  - Grant full permissions to the admin user
- ◇ Additionally, you can set permissions by project

### Project-based Matrix Authorization Strategy

User/group	Overall					Credentials				
	Administer	Read	RunScripts	UploadPlugins	ConfigureUpdateCenter	Create	Update	View	Delete	ManageDomain
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 admin	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

User/group to add:



## 13.15 Project-Based Authentication

- ◇ You will now have an option in each job's configuration to enable project-based security

☒ Enable project-based security

☐ Block inheritance of global authorization matrix

User/group	Credentials						
	Create	Update	View	Delete	Manage	Domains	Delete
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

User/group to add:

- ◇ Add users to this permission list, for the project
- ◇ Permissions are additive
  - Users with global permissions still have those permissions for the project



## 13.16 Role Based Access Control

- The Role Based Access Control plugin allows administrators to define roles and then map users to roles
- Mappings can be applied to specific resources
- So, for instance, only certain groups could be allowed to access a given folder
- As a user, what you'll see depends on the role that you're in



## 13.17 Conclusion

- ◇ Configure security through the “Configure Global Security” screen
- ◇ There are a variety of authentication and authorization options available.

## Chapter 14 - Jenkins Plugins

---

### *Objectives*

Key objectives of this chapter

- Introduction
- Jenkins Plugins – SCM
- Jenkins Plugins – Build and Test
- Jenkins Plugins – Analyzers
- Jenkins for Teams
- Installing Jenkins Plugins



## 14.1 Introduction

- Jenkins has over 300 plugins.
  - ◇ Software Configuration Management(SCM).
  - ◇ Test Frameworks
  - ◇ Notifiers
  - ◇ Static Analyzer
  - ◇ Builders



## 14.2 Jenkins Plugins - SCM

- Version Control Systems:
  - ◇ Git
  - ◇ Subversion
  - ◇ ClearCase



## 14.3 Jenkins Plugins – Build and Test

- Build Tools:
  - ◇ Ant
  - ◇ Maven
- Test Frameworks:
  - ◇ Junit
  - ◇ Selenium





## 14.4 Jenkins Plugins – Analyzers

- Static Analyzers:
  - ◇ FindBugs
  - ◇ PMD
  - ◇ Sonar
  - ◇ Fortified
  - ◇ CodeScanner
- Code Coverage:
  - ◇ Emma
  - ◇ Cobertura
  - ◇ Clover



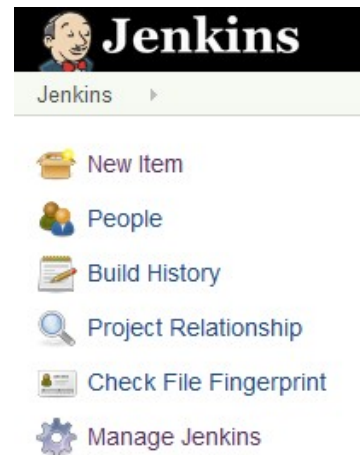
## 14.5 Jenkins for Teams

- Multi Configuration jobs.
- Multi Stage Jobs.
  - ◇ Replaced by the Pipeline plugin
- Folders Plugin
- GitHub Organization Plugin

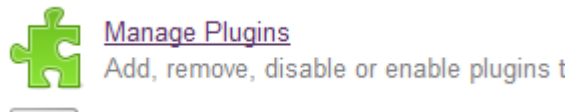


## 14.6 Installing Jenkins Plugins

- For example, installing the Jenkins Emma plugin for code coverage.



- Click **Manage Jenkins**
- Then on click the **Manage Plugins** link:

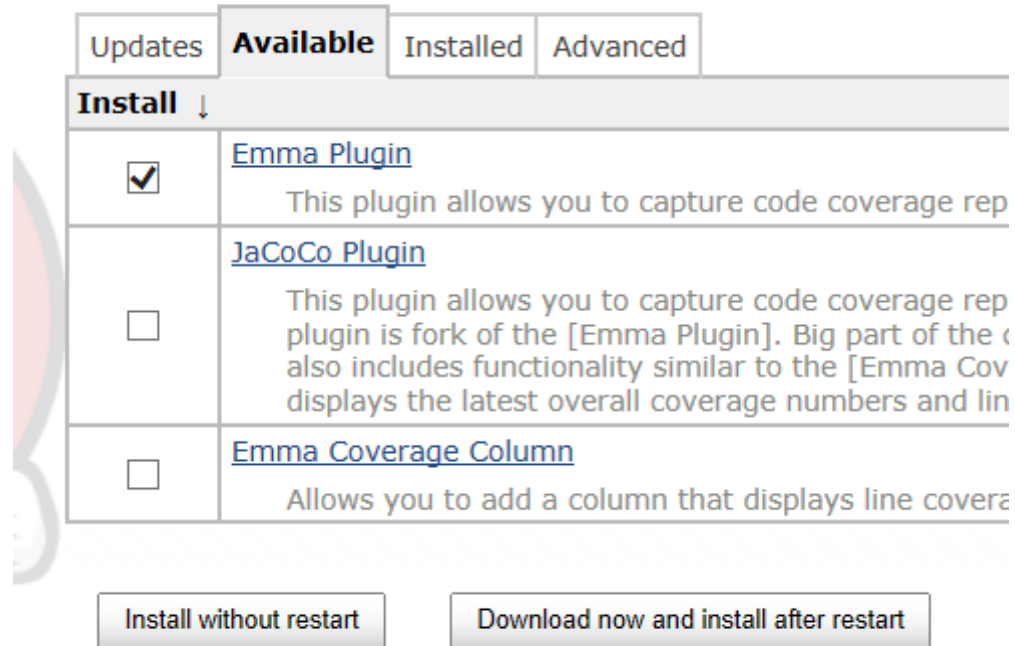




- Select the Available tab:
- For quick access, enter plugin name on the filter input box, located at right top corner.

Filter:  x

- Scroll down the list of plugins to find the Emma Plugin.
- Select the check box next to Emma Plugin.
- Click **Download now and install after restart**



This will download the Emma plugin for Jenkins. The download may take some time to finish. Once the installation is complete, you will get the following:



## Installing Plugins/Upgrades

### Preparation

- Checking internet connectivity
- Checking update center connectivity
- Success

Emma Plugin



Downloaded Successfully. Will be activated during the n



[Go back to the top page](#)

(you can start using the installed plugins right away)



Restart Jenkins when installation is complete and no jobs are running

- Select the Restart Jenkins when installation is complete and no jobs are running.
- You have done the installation
- Note that quite often, the "Install without restart" option works too



## 14.7 Summary

- Jenkins Plugins are where most features of Jenkins are actually implemented
- Different Categories of Plugins are available
- Plugins are installed through the "Manage Jenkins" menu

.

## Chapter 15 - Distributed Builds with Jenkins

---

### *Objectives*

Key objectives of this chapter

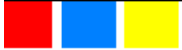
- Setting up Distributed Builds





## 15.1 Distributed Builds - Overview

- Jenkins can act as master/agent setup to distribute builds over multiple machines
- Why?
  - ◇ Additional Capacity
  - ◇ Different build/runtime environments
  - ◇ Integration Test
  - ◇ Managing Machines



## 15.2 Distributed Builds – How?

- ◇ Have another machine
- ◇ Make sure all the required software is installed
- ◇ Version control system
- ◇ Java Development Kit
- ◇ SSH
- ◇ Configure the Jenkins Master
- ◇ If necessary, configure projects



## 15.3 Agent Machines

- ◇ For \*nix, Master contacts agents by SSH
- ◇ Need user created on agent machine
- ◇ Keys/credentials if necessary
- ◇ For Windows, usually just install agent as a Windows Service
- ◇ Needs to have local copies of anything that Jenkins needs
  - Git, SVN, Mercurial, etc
  - Maven
  - Java
  - Jenkins can install some of these automatically



## 15.4 Configure Jenkins Master

- ◇ Create a Node
- ◇ Configure Node for # executors, tool locations, etc



## 15.5 Configure Projects

- ◇ If desired, you can restrict where a project runs
- ◇ Each agent has one or more tags
- ◇ Project can call out a tag
- ◇ Project will only run on agents where the tags match
- ◇ No other config required



## 15.6 Conclusion

- ◇ Distributed builds are readily supported in Jenkins

## Chapter 16 - Scripting

---

### *Objectives*

Key objectives of this chapter

- Introduction to Scripting
- Benefits of Scripting
- The Jenkins Script Console
- Calling Scripts using cURL
- Sample Scripts
- Calling Groovy Scripts from a Jenkins Job
- Jenkins API



## 16.1 Introduction to Scripting

- Usually you create a Jenkins job and define the steps (build, post, etc.) via the GUI
- The GUI is available either OOB or provided by plugins
- Optionally, Groovy scripting can be utilized to create jobs, such as a Pipeline
- Groovy scripting can also be defined as a step (build, post etc.)





## 16.2 Benefits of Scripting

- There are several benefits of scripting
  - ◇ Scripts are reusable
  - ◇ Scripts can be versioned
  - ◇ Scripts can be used for performing complex functions, such as automatic deletion of previous builds etc.



## 16.3 The Jenkins Script Console

- Jenkins features a Groovy script console
- The script console allows to run arbitrary scripts on the Jenkins server or on slave nodes.
- This feature can be accessed in two ways:
  - ◇ Manage Jenkins
  - ◇ <http://server/jenkins/script>
- It is a web-based Groovy shell into the Jenkins runtime.
- Groovy is a very powerful language which offers the ability to do practically anything Java can do



## 16.4 Calling Scripts using cURL

- User can execute Groovy scripts remotely sending POST request to /script url
- e.g.

```
curl -d "script=<your_script>" http://server/script
```

- or

```
curl --data-urlencode "script=$(<./scomscript.groovy>)" http://server/script
```

- If security is configured in Jenkins, then curl can be provided options to authenticate using the curl --user option
- `curl --user 'username:password' -d "script=<your_script>" http://server/script`



## 16.5 Sample Scripts

- Display a simple message

```
PI = 3.14;
message = "Hello, World!";
println("The value of PI is: ${PI}, and the message is: ${message}");
```

- Get job list

```
j = jenkins.model.Jenkins.instance;
for(item in j.getAllItems()) {
    name = item.name;
    url = item.getUrl();
    println("Job name: ${name}, URL: ${url}");
}
```

- Get successful build list

```
println "Jobs with successful builds\n-----"
activeJobs = hudson.model.Hudson.instance.items.findAll{job ->
job.isBuildable() }
```

```
successfulRuns = activeJobs.findAll{job -> job.lastBuild != null
&& job.lastBuild.result == hudson.model.Result.SUCCESS}
```



```
successfulRuns.each{run -> println "Successful Job Name: ${run.name}"}
```

- Various useful scripts are available on following site

<https://github.com/jenkinsci/jenkins-scripts/tree/master/scriptler>



## 16.6 Calling Groovy Scripts from a Jenkins Job

- Groovy scripts can be called from a Jenkins job in two ways
  - ◇ Build step
    - Requires **Groovy** plugin
    - **Add Build Step > Execute system Groovy script**
  - ◇ Post-build step
    - Requires **Groovy Postbuild** plugin
    - **Add post-build action > Groovy Postbuild**



## 16.7 Jenkins API

- Importing Jenkins classes

```
import jenkins.model.*;
```

- Obtaining Jenkins instance

```
ji = Jenkins.instance;
```

- Obtaining a job by name

```
job = ji.getItemByFullName('<job_name>');
```

- Additional useful functions

- ◇ `getJobNames()`

- ◇ `restart()`

- ◇ `doWipeOutWorkspace()`

- ◇ `getBuilds()`

- API documentation is available on following site:

<http://javadoc.jenkins-ci.org/>



## 16.8 Summary

- Scripting allows a powerful way of executing operations in Jenkins that aren't available when using the GUI
- Scripts can be written in the Jenkins Script Console or defined as part of a Jenkins job.



## Chapter 17 - Best Practices for Jenkins

---

### *Objectives*

Key objectives of this chapter

- Best Practices.



## 17.1 Best Practices - Secure Jenkins

- Always secure Jenkins.
  - ◇ Without security, the Jenkins dashboard makes your source code available to anyone
  - ◇ Also, pre-release artifacts
- Even if most of your users are trusted, the setup of projects is sometimes complicated
  - ◇ Securing those projects helps prevent accidents
- See the security chapter



## 17.2 Best Practices - Users

- Jenkins will need to have user credentials on other systems
  - ◇ e.g. to read from SCM, deploy software, etc
- **Don't let users put their real credentials into jobs or pipelines!**
- Create a user account for Jenkins itself
- Don't publish the login credentials for this "server account"
  - ◇ Use the credentials system in Jenkins
  - ◇ You can grant usage of the credentials through Jenkins security subsystem



### 17.3 Best Practices - Backups

- Everything of any interest is stored in the Jenkins Home directory
  - ◇ Back it up regularly
- JENKINS\_HOME/jobs
  - ◇ contains all the job information
  - ◇ You can copy jobs from one Jenkins installation to another
- JENKINS\_HOME/workspace
  - ◇ is where the checkouts from version control go to
  - ◇ Could be skipped in backups if space is a problem
- Archive unused jobs before removing them.
  - ◇ Store a copy of the project's folder under JENKINS\_HOME/jobs
  - ◇ If you need to restore, copy the archived folder back into 'jobs', then 'Manage Jenkins --> Reload config from disk'
  - ◇ Can be done while Jenkins is running



## 17.4 Best Practices - Reproducible Builds

- Ensure a build is reproducible, the build must be a clean build, which is built fully from Source Code Control, plus authorized binaries
- All code including build scripts, release notes, etc. must be checked into Source Code Control.
  - ◇ If it isn't in Version Control, it didn't happen!
- Dependencies in Version Control?
  - ◇ This is a contentious issue
  - ◇ Old-school says "put the dependency jars in version control"
  - ◇ Modern approaches usually use binary repository like Nexus, Artifactory, or Apache Archiva for dependencies
  - ◇ Discuss, set policy and follow it
- Repeatable build is always the goal!



## 17.5 Best Practices - Testing and Reports

- Always configure your job to generate trend reports and automated testing when running a Java build
- Make your build self-testing
  - ◇ Run tests as part of the build process
  - ◇ Provide rapid feedback
    - define a test pipeline with distinct test phases
- Take steps to ensure failures are reported as soon as possible.
  - ◇ For example, it may be appropriate to run a limited set of "sniff tests" before the full suite.
- Integrate with your issue management system as much as possible
- Set up email notifications mapping to ALL developers in the project, so that everyone on the team has his or her pulse on the project's current status.
- Tag, label, or baseline the code-base after a successful build.



## 17.6 Best Practices - Large Systems

- For integration tests, and multiple jobs, allocate a different port for parallel project builds and avoid scheduling all jobs to start at the same time
- If builds conflict, look into the "Locks and Latches" plugin, or the "Throttle Concurrent Builds" plugin
- Multiple jobs running at the same time can cause resource overload.
  - ◇ Try to avoid scheduling all jobs to start at the same time.
    - e.g. Use 'H/5' rather than '\* /5' in the schedule entries.
      - Spreads out the start time
- Write jobs for your maintenance tasks, such as cleanup operations to avoid full disk problems.



## 17.7 Best Practices - Distributed Jenkins

- In larger systems, use distributed builds and disallow build on the master machine. This ensures that the Jenkins master can scale to support many more jobs than if it had to process build jobs directly as well.
- Be aware that Jenkins may use a lot of disk space!
- Make sure your installation process for Jenkins Master and Agents are documented and automated
  - ◇ Either use a configuration management system (Ansible, Puppet, Chef, etc)
  - ◇ Or use a containerized agent (Docker)





## 17.8 Best Practices - Summary

- ◇ Backup!
- ◇ Repeatable builds!
- ◇ Communicate!
- ◇ Use Jenkins' features
  - Testing and Reporting
  - Distributed Jenkins