**Traversing the DOM**

Traversing the DOM

The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object. The properties used to transverse the DOM or to reach/get a particular DOM object are listed below:

Topmost nodes

The document object is the root of every node in the DOM. The topmost tree nodes are associated with document properties:

* document.documentElement property - refers the DOM node of the <html> tag.
* document.head property - refers the DOM node of the <head> tag.
* document.body property - refers the DOM node of the <body> tag.

Parent Nodes

The parent of any node is the node that is one level above in the DOM tree. There are two properties to get the parent — parentNode and parentElement. The parentNode is most commonly used for traversing the DOM.

**Example:**

<body>

<div id="main">

<p id="para">This is a note!</p>

</div>

<script>

let elmt **=** document.querySelector('#para');

document.write(elmt.parentNode**+** "<br>"); // outputs: [object HTMLDivElement]

</script>

</body>

The parentElement property returns the “element” parent, whereas parentNode returns “any node” parent. With the one exception of document.documentElement:

alert( document.documentElement.parentNode ); // document

alert( document.documentElement.parentElement ); // null

Child Nodes

The children of a node are the nodes that are one level below it. The properties are listed below:

* childNodes property - returns a list of child nodes, including text nodes.
* firstChild property - give access to the first child node.
* lastChild property - give access to the last child node.

**Example:**

<body>

<div id="myDiv">

<p>This is a paragraph - first child</p>

<div> this is a div elemt - last child</div>

</div>

<script>

let elmt **=** document.querySelector('#myDiv');

document.write("<br> Child nodes of div element: <br>");

**for** (let i **=** 0; i **<** elmt.childNodes.length; i**++**) {

document.write(elmt.childNodes[i] **+** "<br>");

}

document.write("<br> First child of div element: <br>" **+**elmt.firstChild) ;

document.write("<br> Last child of div element: <br>" **+**elmt.lastChild) ;

</script>

</body>

The output will be:

This is a paragraph - first child

this is a div elemt - last child

Child nodes of div element:

[object Text]

[object HTMLParagraphElement]

[object Text]

[object HTMLDivElement]

[object Text]

First child of div element:

[object Text]

Last child of div element:

[object Text]

Sibling Nodes

Siblings are nodes that are children of the same parent. The siblings of a node are any node on the same tree level in the DOM.

* previousElementSibling property - give access to the previous sibling Node, i.e. the node that immediately precedes the specified node.
* nextElementSibling property - give access to the next sibling vode, i.e. the node that immediately precedes or follows the specified node.

**Example:**

<body>

<ul >

<li>list item 1</li>

<li class="list">list item 2</li>

<li>list item 3</li>

</ul>

<script>

const secondListItem **=** document.querySelector('.list');

document.write(secondListItem.previousElementSibling.textContent) ; // outputs: "list item 1"

document.write(secondListItem.nextElementSibling.textContent); //outputs: "list item 3"

</script>

</body>

**DOM Manipulation**

DOM Manipulation

We can add, remove, replace, and copy any element into a DOM tree. DOM Manipulation methods are listed below:

Create Elements

The createElement() method is used to create a new element and attach it to the DOM tree.

**Example:**

var elmt **=** document.createElement('div');

elmt.innerHTML **=** '<p>Hello World!</p>';

Replace Child Elements

The replaceChild() method is used to remove an element from the DOM tree and insert a new one in its place.

**Example:**

<body>

<div>

This is a div element.

<div>

<script>

// selecting the <div> element

var elmt **=** document.querySelector('div');

//creating **new** <p> element and adding content inside it.

var newElmt = document.createElement('p');

newElmt.innerHTML **=** '<b>The div element is replaced with p element</b>';

//replacing the <div> element with the <p> element

elmt.parentNode.replaceChild(newElmt, elmt);

</script>

</body>

Remove Child Elements

The removeChild() method is used to remove an element from the DOM tree.

**Example:** Here, first we select the element to remove, then walk up the tree to its parent and remove the child element from there.

var elmt **=** document.querySelector('div'); // select the first returned <div> element

elmt.parentNode.removeChild(elmt);

Append a Node

The appendChild() method is used to add a node to the end of the list of child nodes of a specified parent node.

**Example:** Here, we add three list items to the <ul> element

<body>

<ul id="animals">

</ul>

<script>

**function** **createAnimalList**(name) {

let li **=** document.createElement('li');

li.textContent **=** name;

**return** li;

}

// get the ul #animals

const list **=** document.querySelector('#animals');

// add animals to the list

list.appendChild(createAnimalList('Lion'));

list.appendChild(createAnimalList('Tiger'));

list.appendChild(createAnimalList('Wolf'));

</script>

</body>

Insert a Node before another

The insertBefore() method is used to insert a node before another node as a child node of a parent node.

**Example:** Here, we insert the new <li> element before the first child of <ul> element.

<body>

<ul id="animal">

<li>Lion</li>

<li>Tigerr</li>

</ul>

<script>

let animal **=** document.getElementById('animal');

// create a **new** li node

let li = document.createElement('li');

li.textContent **=** 'Wolf';

// insert a **new** node before the first list item

animal.insertBefore(li, animal.firstElementChild);

</script>

</body>

Insert a Node After another

JavaScript DOM provides the insertBefore() method that allows you to insert a new after an existing node as a child node. However, it hasn’t supported the insertAfter() method yet.

So, we can insert a new node after an existing node as a child node, by selecting a parent node of the existing node and call the insertBefore() method on the parent node to insert a new node before that immediate sibling node.

**Example:** Here, we inserts the new <li> element after the first child of <ul> element.

<body>

<ul id="animal">

<li>Lion</li>

<li>Tiger</li>

</ul>

<script>

let animal **=** document.getElementById('animal');

// create a **new** li node

let li = document.createElement('li');

li.textContent **=** 'Wolf';

// insert a **new** node before the first list item

animal.insertBefore(li, animal.firstElementChild.nextSibling);

</script>

</body>

Get or Set text of a Node

The textContent property is used to get and set a text content inside a particular node.

**Example:**

<body>

<div id = "content">

This is div element.

</div>

<script>

// Getting a text content

let content **=** document.getElementById('content');

alert("Getting a text content inside div element: " **+** content.textContent);

//setting a text content

content.textContent **=** 'New text content in the div element';

</script>

</body>

Get or Set HTML of Element

The innerHTML property to get the text and inner HTML elements inside a the particular element and the setting will replace the existing text and inner tags inside an element with the new content.

**innerHTML vs textContent** - The innerHTML property returns the text and inner HTML elements. The textContent property returns only the text Content.

**Example:**

<body>

<div id="myBdy">

<p id = "para">This is Paragraph.</p>

<button onclick="myFunction()" >Try it</button>

<p id="demo"></p>

</div>

<script>

**function** **myFunction**() {

// get HTML of Element

var x **=** document.getElementById("para").innerHTML;

document.getElementById("demo").innerHTML **=** x;

}

// You can understand the difference between innerHTML and textContent property clearly from the output of the

// below two alert boxes

alert ("textcontent property:" **+** document.getElementById("myBdy").textContent);

alert ("innerHTML property:" **+** document.getElementById("myBdy").innerHTML);

</script>

</body>

Clone a Node

The cloneNode() method is used to clone an element. The cloneNode() method accepts an optional parameter. If the parameter value is true, then the original node and all of its descendants are cloned. If the parameter value is false, only the original node will be cloned.

**Example:** Here, the parameter value for cloneNode() method is true. So it clones the target node and all of its descendants.

<body>

<ul id="animal">

<li>Lion</li>

<li>Tiger</li>

<li>Wolf</li>

</ul>

<script>

let list **=** document.querySelector('#animal');

let clonedList **=** list.cloneNode(true);

clonedList.id **=** 'cloned animal';

document.body.appendChild(clonedList);

</script>

</body>

Managing Attributes

* getAttribute(attribute\_name) method - Used to get the value of an attribute on a specified element
* setAttribute(attribute\_name, attribute\_value) method - Used to set a value of an attribute on a specified element,
* removeAttribute(attribute\_name) method - Used to remove an attribute with a specified name from an element
* hasAttribute(attribute\_name) method - Used to check an element has a specified attribute or not.

**Selecting elements from the DOM**

Selecting elements from the DOM

JavaScript is used to get or modify the content or value of the HTML elements on the page. To perform any action on the HTML element, we need to select the target HTML element.

The ways for selecting the elements on a page are listed below:

Selecting Elements by ID

The getElementById() method is used to select an element based on its unique ID. The getElementById() method will return the element as an object if the matching element was found, or null if no matching element was found in the document.

**Example:** In the example below, getElementById()is a method, while innerHTML is a property. It selects the element with the id = "demo" attribute and changes the content to "Paragraph Changed".

<body>

<p id="demo">This is a paragraph.</p>

<script>

document.getElementById("demo").innerHTML **=** "Paragraph Changed";

</script>

</body>

Selecting Elements by Class Name

The getElementsByClassName() method used to select all the elements having specific class names. This method returns an array-like object of all child elements which have all of the given class names.

**Example:** selects the list of elements which have class="test" attribute and changes the background color as yellow.

<body>

<div class="test"> This is a div element with class="test". </div>

<p>

<h1 class= "test"> This is a h1 element with class="test". </h1>

This is a paragraph.

</p>

<p class="test">This is a p element with class="test".</p>

<script>

var matches **=** document.getElementsByClassName("test");

**for**(var elem in matches) {

matches[elem].style.background **=** "yellow";

}

</script>

</body>

Selecting Elements by Tag Name

The getElementsByTagName() method used to select HTML elements by tag name. This method also returns an array-like object of all child elements with the given tag name.

**Example:** selects all the <p> element and changes the background color as red.

<body>

<h1> Heading </h1>

<p>This is a paragraph of text.</p>

<div>This is another paragraph of text.</div>

<p>This is one more paragraph of text.</p>

<script>

var matches **=** document.getElementsByTagName("p");

**for**(var elem in matches) {

matches[elem].style.background **=** "red";

}

</script>

</body>

Selecting Elements with CSS Selectors

We can use querySelector() and querySelectorAll() methods to select elements that matches the specified CSS selector. The querySelector() finds the first element that matches a CSS selector whereas the querySelectorAll() finds all elements that match a CSS selector.

**Example:**

<body>

<p id = "para">This is a paragraph</p>

<ul>

<li>Apple</li>

<li>Orange</li>

<li>Mango</li>

</ul>

<script>

var matches **=** document.querySelectorAll("ul li");

**for**(var elem of matches) {

document.write(elem.innerHTML **+** "<br>"); //outputs: "Apple Orange Mango"

}

document.write(document.querySelector('#para').textContent); //outputs: "This is a paragraph"

</script>

</body>

**Events & Listeners**

JavaScript Events

Events occur when user interaction takes place on a web page, such as when the end-user clicked a link or button, pressed a key on the keyboard, moved the mouse pointer, submits a form, etc. The browser also triggers the events, such as the page load and unload events.

When an event occurs, we use a JavaScript event handler (or an event listener) to detect them and perform a specific task.

Some of the commonly used JavaScript Events are listed below:

* **onclick** - This is a click event occurs when a user clicks the element on a web page.
* **ondblclick** - This is a click event occurs when a user double clicks the element on a web page.
* **onload** - This is a load event occurs when the browser has finished loading the page.
* **onunload** - This is a load event occurs when a user closes the document.
* **onresize** - This is a load event occurs when the browser window is minimized or maximized.
* **onmouseover** - This is a mouse event occurs when the user moves the mouse over an HTML element.
* **onmouseout** - This is a mouse event occurs when the user moves the mouse away from an HTML element.
* **onkeydown** - This is a keyboard event occurs when the user presses down a key on the keyboard.
* **onkeyup** - This is a keyboard event occurs when the user releases a key on the keyboard.
* **onfocus** - This is a form-input event occurs when the user gives focus to an element on a web page.
* **onblur** - This is a form-input event occurs when the user takes the focus away from a form element or a window.
* **onchange** - This is a form-input event that occurs when a user changes the value of a form element.
* **onsubmit** - This is a form-input event that occurs when the user submits a form on a web page.

**Example:**

<body>

<button onclick= onclickEvent()>Click me!!</button>

<p id= "para" onmouseover = onmouseoverEvent() onmouseout = onmouseoutEvent() >This is a Paragraph....</p>

<script>

**function** **onclickEvent**(){

alert('Hello, You clicked the button');

}

**function** **onmouseoverEvent**(){

document.getElementById("para").style.backgroundColor **=** "green";

}

**function** **onmouseoutEvent**(){

document.getElementById("para").style.backgroundColor **=** "yellow";

}

</script>

</body>

JavaScript EventListener

An event listener is a function in JavaScript that waits for an event to occur. The addEventListener() function is an inbuilt function in JavaScript used to handle the event.

The Syntax of addEventListener() function: element.addEventListener(event, function, useCapture)

Where,

* event - Specifies the name of the event.
* function - Specifies the function to run when the event occurs
* useCapture - It is an optional parameter takes a boolean value. If the parameter value is true then the event handler is executed in the capturing phase. If the parameter value is false then the event handler is executed in the bubbling phase.

The removeEventListener() method used to remove an event handler that has been attached with the addEventListener() method.

**Example:**

<body>

<h2>JavaScript addEventListener()</h2>

<p id="myBtn">This is a paragraph.</p>

<p id="demo"></p>

<script>

var x **=** document.getElementById("myBtn");

x.addEventListener("mouseover", myFunction);

x.addEventListener("mouseout", mySecondFunction);

**function** **myFunction**() {

document.getElementById("demo").innerHTML **+=** "Moused over!<br>";

x.style.backgroundColor **=** "green";

}

**function** **mySecondFunction**() {

document.getElementById("demo").innerHTML **+=** "Moused out!<br>";

x.style.backgroundColor **=** "white";

}

</script>

</body>

**Bubbling, capturing**

Event Bubbling

In Event Bubbling, the event propagates from the target element to its parents, then all its ancestors that are on the way to top. Bubbling follows the **Bottom to Top** approach.

**Example:** Event Bubbling works for all handlers, regardless of how they registered with the addEventListener(). When we click on any element, event propagates or bubbles back up the DOM tree, from the target element up to the Window, visiting all of the ancestors of the target element one by one. (a-> p -> div)

<body>

<div onclick="alert('Bubbling: ' + this.tagName)">DIV

<p onclick="alert('Bubbling: ' + this.tagName)">P

<a href="#" onclick="alert('Bubbling: ' + this.tagName)">Click Me!!</a>

</p>

</div>

</body>

If we click on the <a> element in the above example, it results in the alert pop-ups in below order:

1. alert pop-ups saying 'Bubbling: a'
2. alert pop-ups saying 'Bubbling: p'
3. alert pop-ups saying 'Bubbling: div'

Event Capturing

In Event Capturing, the event propagates from the top element to the target element. Capturing follows the **Top to Bottom** approach.

**Example:** Event capturing only works with event handlers registered with the addEventListener() method when the third argument is set to true. When we click on the any element, the event capturing propagates the element from top element to target element (div -> p -> a).

<body>

<div id="wrap">DIV

<p class="hint">P<br>

<a href="#">Click Me!!</a>

</p>

</div>

<script>

**function** **showTagName**() {

alert("Capturing: "**+** this.tagName);

}

var elems **=** document.querySelectorAll("div, p, a");

**for**(let elem of elems) {

elem.addEventListener("click", showTagName, true);

}

</script>

If we click on the <a> element in the above example, it results in the alert pop-ups in below order:

1. alert pop-ups saying 'Capturing: div'
2. alert pop-ups saying 'Capturing: p'
3. alert pop-ups saying 'Capturing: a'

**Event Target**

Event Target is the target element that has generatd the event in DOM. The event.target is used to access the target element.

**Stopping the Event Propagation**

* event.stopPropagation() method

It used to stop the event to travel to the bottom to top i.e. Event Bubbling. If you want to stop the event flow from event target to top element in DOM, we use event.stopPropagation() method.

* event.stopImmediatePropagation() method

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute. The event.stopPropagation() stops event bubbling but all other handlers will run. To stop the bubbling and prevent handlers on the current element from running, we use event.stopImmediatePropagation() method.

**let and const keywords**

let and const keywords

In ES6, const and let keywords allow developers to declare variables in the block scope, which means those variables exist only within the corresponding block.

Variables declared with the let and const keyword can have **Block Scope**.

let Keyword

The variables declared inside a block {} have Block Scope, so it can not be accessed from outside the block.

**Example:**

{

let y **=** 2;

}

// y can NOT be used here

var x **=** 10;

// Here x is 10

{

let x **=** 2;

// Here x is 2

}

// Here x is 10

const Keyword

const is block-scoped, much like variables defined using the let keyword. The value of a const variable can't be changed through reassignment, and it can't be redeclared.

**Example:** We cannot change the primitive value of constant variable.

const a **=** 12;

a **=** a**+**11; //error

a**=**"hello"; //error

**Example:** We can change the properties of a constant object but we can't reassign the constant object.

//const object

const person **=** { name: "Johnson" , age: "23", gender: "male"};

// change a property

person.age **=** "21";

console.log(person); // prints " {name: "Johnson", age: "21", gender: "male"} " in the console.

//reassigning const object

person = { name: "Mercy" , age: "23", gender: "female"}; //error

**Template Literal**

ES6 Template Literals

Template literals are a new feature introduced in ECMAScript 2015/ ES6. It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions.

Template literals are strings that enclosed within the backtick character(`). Template literals allow for embedded expressions (placeholders), which are indicated by the dollar sign and curly braces ($(expression}). These expressions can be used to evaluate code.

**Syntax**: var str = `string value`

Multiline strings

We use the escape character, represented as \n, to give a new line for creating a multiline string. In Template Literals, there is no need to add \n at the end when the string enclosed within the backtick (`) character. Template literals used to create the new line by literally moving a string to a new line without using any character.

**Example:**

console.log(`I am the first line

I am the second line

I am the third line`);

**Output:**

I am the first line

I am the second line

I am the third line

String Interpolation

In JavaScript, the template literals give support for string interpolation. Template literals perform string interpolation using embedded expressions, ${}, which are replaced with the value of the code within it.

**Example:**

**function** **sayHello**(){

**return** "Hello World!!!"

}

var x **=** 10;

var y **=** 20;

document.write(`${sayHello()}, The product of the variables ${x} **and** ${y} are

${x**\***y}`);

**Output on the console:**

Hello World!!!, The product of the variables 10 and 20 are 200.

Tagged templates:

Tagged templates are one feature provided by ES6. Tagged templates allow us to parse template literals with a function.

The following is a tagged template:

tagFunction`Hello ${firstName} ${lastName}!`

Putting a template literal after an expression triggers a function call, similar to how a parameter list triggers a function call.

The above code is equivalent to the following function call:

tagFunction(['Hello ', ' ', '!'], firstName, lastName)

Thus, the name before the content in backticks is the name of a function to call, the tag function. The first argument of the tag function contains an array having string values, like ['Hello ', ' ', '!'] in the above tag Function. The remaining arguments are substitutions delimited by ${}, such as firstName and lastName in the above tag Function.

**Example:**

**function** **printAll**(literalArray, operator1, operator2, result){

console.log(literalArray);

console.log(operator1);

console.log(operator2);

console.log(result);

}

a **=** 3;

b **=** 4;

printAll `Addition: ${a} **+** ${b} **=** ${a**+**b}`;

**Output on the console:**

(4) ["Addition: ", " + ", " = ", "", raw: Array(4)]

3

4

7

Raw Strings

The template literal raw method allows the accessing of raw strings as they were entered.

The String.raw() is a built-in function which accepts a string literal argument and returns raw string. It returns the strings without any interpretation of backslashed characters.

**Example:**

var a **=**3;

var b **=**4;

var myString **=** **String**.raw`sum:\n ${a**+**b}`;

console.log(myString);

**Output on the console:**

sum:\n 7

The raw is a special built-in property which can be used on the first function argument of tagged templates. This allows us to access the raw strings as they were entered.

**Example:**

**function** **indexTest**(literalArray){

console.log(literalArray.raw[0]);

console.log(literalArray[0]);

}

indexTest `"finding \n errors"`;

**Output on the console:**

"finding \n errors"

"finding

errors"

**Arrow functions**

### **Arrow Functions**

Arrow functions, introduced in ES6, provides a concise way to write functions in JavaScript. They save developers time and simplify function scope.

* **One-line arrow functions** have implict return and defined without curly braces.

let func = (arg1, arg2, ...argN) => expression;

**Example:**

var hello **=** () **=>** "Hello World!";

hello(); // output: "Hello World!"

* **Multiline arrow functions** have multiple statements inside the function, enclosed with curly braces and need an explicit return to return something.

let func = (arg1, arg2, ...argN) =>{

stament1;

...

statementn;

return value;

}

**Example:**

var sum **=** (a, b) **=>** {

let result **=** a **+** b;

**return** "Sum is" **+** result;

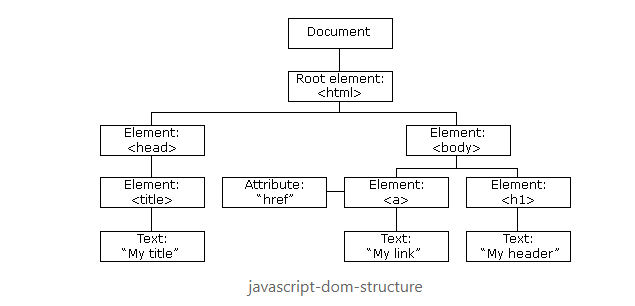
};

sum(5, 3); //output: "sum is 8"

**DOM Structure**

Document Object Model (DOM)

The Document Object Model (DOM) is a programming API for HTML and XML documents. It enables Javascript to access and manipulate the elements and styles of a website. The browser creates a tree-like hierarchical representation of the HTML document, that tree-like structure is known as **DOM Structure** or a **DOM tree**.



Each HTML element in the DOM tree is an object. The positions of the elements in the DOM tree are nodes.The tags are element nodes. Attributes in the elements are attribute nodes. The text inside elements is a text node. It may not have children and is always a leaf of the tree. The root of the DOM tree is a <html> element, which is known as a **document object**.

**Example:**

Below, we have a simple HTML Document:

<!DOCTYPE HTML>

<html>

<head>

<title>Title goes here</title>

</head>

<body>

<p> DOM Structure </p>

</body>

</html>

The DOM structure for the above HTML document looks like:

HTML (root)

|

|---HEAD

| |

| |----TITLE

| |

| |-----#text - "Title goes here"

|

|---BODY

|

|----P

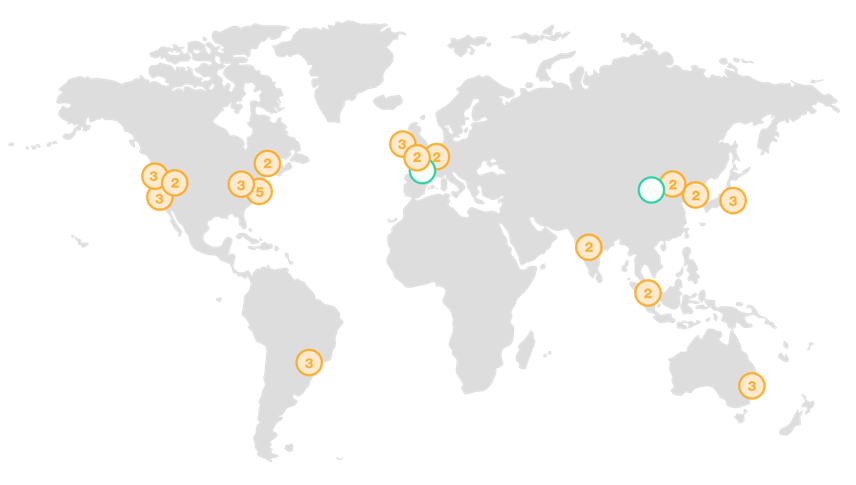
|

|----- #text - "DOM Structure"

**AWS Regions and Availability Zones**

## AWS Regions & Availability Zones

AWS now spans **77 Availability Zones** within **24 geographic regions** around the world, and has announced plans for nine more Availability Zones and three more AWS Regions in Indonesia, Japan, and Spain.



### **AWS Regions**

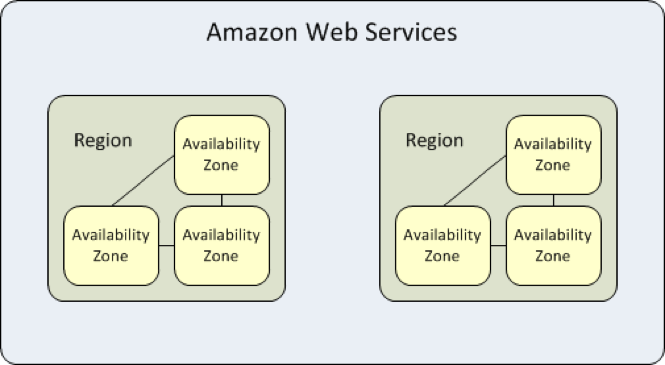
An **AWS Region** is a geographical location with a collection of availability zones mapped to physical data centers in that region. Every region is physically isolated from and independent of every other region in terms of location, power, water supply, etc.

This level of isolation is critical for workloads with compliance and data sovereignty requirements where guarantees must be made that user data does not leave a particular geographic region. The presence of AWS regions worldwide is also important for workloads that are **latency-sensitive** and need to be located near users in a particular geographic area.

Inside each region, you will find two or more **availability zones** with each zone hosted in separate data centers from another zone.

### **AWS Availability Zones**

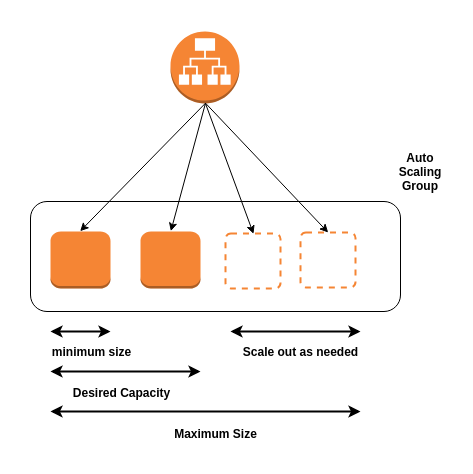
An availability zone is a logical data center in a region available for use by any AWS customer. Each zone in a region has redundant and separate power, networking and connectivity to reduce the likelihood of two zones failing simultaneously.



**EC2 Autoscaling**

## EC2 Autoscaling

Amazon EC2 Auto Scaling helps you maintain application availability and allows you to automatically add or remove EC2 instances according to conditions you define.



### **Benefits**

1. Improve Fault Tolerance
2. Increase Application Availability
3. Lower Costs

##### **Improve Fault Tolerance**

Amazon EC2 Auto Scaling can detect when an instance is unhealthy, terminate it, and replace it with a new one.

##### **Increase Application Availability**

Amazon EC2 Auto Scaling ensures that your application always has the right amount of compute, and also proactively provisions capacity with Predictive Scaling.

##### **Lower Costs**

Amazon EC2 Auto Scaling adds instances only when needed, and can scale across purchase options to optimize performance and cost.

### **Auto-Scaling Groups**

**Groups** are collections of EC2 instances with similar charcteristics. Using the auto scaling groups you can increase the number of instances to improve your application performance and also you can decrease the number of instances depending on the load to reduce your cost. The auto-scaling group also maintains a fixed number of instances even if an instance becomes unhealthy.

### **Launch Configuration**

The launch configuration is a template used by auto scaling group to launch EC2 instances. You can specify the Amazon Machine Image (AMI), instances type, key pair, and security groups etc.. while creating the launch configuration.

### **Scaling Plans**

Scaling plans tells Auto Scaling when and how to scale. Amazon EC2 auto-scaling provides several ways for you to scale the auto scaling group.

* Schedules Scaling
* Dynamic Scaling
* Predictive Scaling

### **How to Configure Autoscaling**

In order to configure autoscaling, you must create an AMI and launch template. You can find [full autoscaling documentation here](https://docs.aws.amazon.com/autoscaling/ec2/userguide/GettingStartedTutorial.html#gs-tutorial-next-steps). The basic steps are as follows:

1. Create a launch template
2. Create an Auto Scaling group
3. Verify your Auto Scaling group
4. Customize Auto Scaling plan.
5. (Optional) Delete your scaling infrastructure

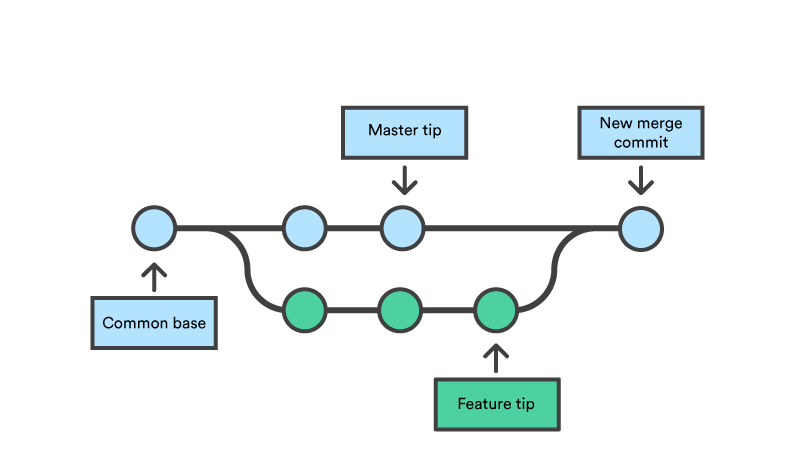
**Git flow & Branching strategies**

## Git Branching

Branching is a feature available in most modern version control systems. Instead of copying files from directory to directory, Git stores a **branch** as a reference to commit. The branch itself represents the HEAD of a series of commits.

The default branch name in Git is master, which commonly represents the official, working version of your project. As you start making commits, the master branch points to the last commit you made. Everytime you commit, the master branch pointer moves forward automatically. Think of a branch as a timeline of versions of a project as it progresses.

Branching is a strategy that allows developers to take a snapshot of the master branch and test a new feature without corrupting the project in production. If the tests are successful, that feature can be **merged** back to the master branch and pushed to production.



### **Create a login branch**

Imagine that you want to create a login functionality feature on your MyApp project.

1. To create a new branch called login, within your MyApp directory run:

$ git branch login

1. To checkout to the newly created branch, run:

$ git checkout login

Now you will be able to create and edit files that will only exist on this branch until you merge it with master.

**A note on git checkout vs. git switch:** Please keep in mind that both checkout and switch will change branches to the branch you specify following the command. There are some limitations to each command which you can explore in further detail [here](https://www.atlassian.com/git/tutorials/using-branches/git-checkout).

1. Create an html document called login.html by running touch login.html within the login branch.



1. Fill out your login.html page with a simple template such as the following:

<html>

<head>

<title>My App</title>

</head>

<body>

This is my App!

Login

</body>

</html>

1. Save your file and add the changes to staging within the login branch with git add .. Then commit with git commit -m "add login form".
2. Now return to your master branch with git checkout master.
3. To integrate the files that you created and populated in the login branch, run the following commands to merge the login branch, add the files, commit changes, then push to the remote repository.

$ git merge login -m "merge login form"

$ git add .

$ git commit -m "add login form"

$ git pull

$ git push

### **References**

* [Git Branching](https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell)

Day – 3

**Containers vs VMs**

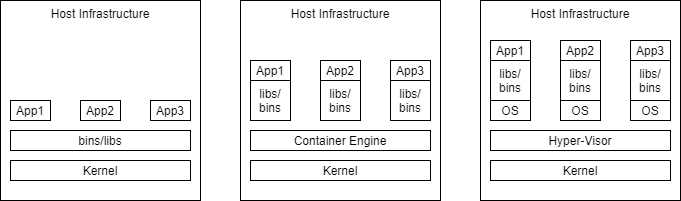
# Containers vs. Virtual Machines

## References

* [Containers explained in relation to VMs from F5 Dev Central](https://www.youtube.com/watch?v=wuhxSLapDe0)
* Docker for the Virtualization Admin Book- [View here](https://goto.docker.com/rs/929-FJL-178/images/docker-for-the-virtualization-admin.pdf)
* [IBM on VMs vs. Containers](https://www.ibm.com/cloud/learn/containerization#toc-virtualiza-jttBZ0x-)
* [IBM on Virtualization](https://www.ibm.com/cloud/learn/virtualization-a-complete-guide)

## Overview

Virtual machines and containers provide the ability to isolate processes from one another and provide some kind of virtualization, so that the processes can run in their own sandbox environment on the same host machine. However, virtual machines and containers do so in differing ways.



**Note**, virtualization means to provide a virtualized view of some set of resources. For example, a process may have a virtualized view of a file system where it believes that it is manipulating the root of the system. Meanwhile, the directory that the process manipulates is many layers away from the root.

## Virtual Machines

Virtual Machines simulate a physical server so that multiple "servers" can run on a single machine.

They virtualize the entire supporting OS. Thus, using virtual machines your host computer could be windows based and you could still run a Linux OS and an application on top of it.

Virtual machines are enabled by hypervisors, software that coordinates between multiple vms and interfaces with the underlying infrastructure. (Some hypervisors interact directly with the hardware, while others run on top of the underlying OS.)

### **Pros**

* near total isolation
* Provides virtualization
  + virtualizing the entire OS
* Ensures an application runs reliably regardless of Host

### **Cons**

* considered "bulky", expensive in the context of resources

## Containers

Containers bundle together applications with their supporting libraries and dependencies, allowing them to run isolated from one another. However, containers still share the underlying OS kernel and are, therefore, much lighter weight than virtual machines.

Containers provide their processes a virtualized view of the underlying resources. Processes within a container only see a particular version of their environment and have limited access to resources- isolating them from the rest of the processes running on a given host.

Containerization is enabled by an engine running on the host i.e. the Docker Engine.

### **Pros**

* considered "light weight", because they don't require spinning up a whole guest OS
* they can enable layers of isolation or partial isolation-- depending on how they are implemented
* provide a virtualized view of certain resources.
* Package an application in an isolated environment
* Ensure an application runs reliably regardless of Host

### **Cons**

* having layers of isolation
  + if you have need of very strict and complete isolation the ability to have layers can be a con

## Summary

In most cases containers are preferable to virtual machines since they provide isolation and virtualization without the cost of having to spin up an entire OS on top of the host. They also enable you to vary the layers of isolation, occasionally sharing necessary resources between processes in differing containers.

However, when a developer desires near total isolation then they may prefer to run their set of processes/application in a virtual machine.

**Containerization**

# Contanerization

## Reference

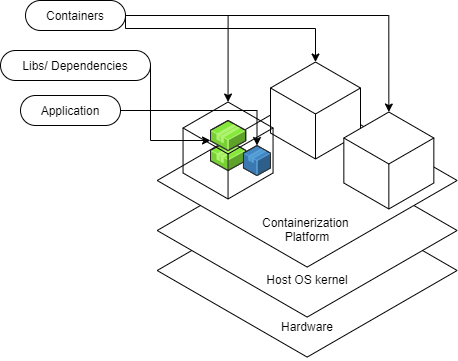
* [History of Containerization](https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016)
* [Video Explanation from VMWare](https://www.youtube.com/watch?v=EnJ7qX9fkcU)
* Docker for the Virtualization Admin Book
* [IBM learning Containerization](https://www.ibm.com/cloud/learn/containerization)
* [Namespaces Explained from NDC by Michael Kerrisk](https://app.revature.com/https:/www.youtube.com/watch?v=0kJPa-1FuoI&list=RDCMUCTdw38Cw6jcm0atBPA39a0Q&start_radio=1)
* [Windows Containers](https://app.revature.com/https:/docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container)
* [Windows Containers Under the hood](https://medium.com/@justen.walker/a-short-introduction-to-windows-containers-db5adc0db536)
* [Windows Containers a History](https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/april/containers-bringing-docker-to-windows-developers-with-windows-server-containers)

## Introduction

A **container** is a mechanism for packaging an application(or limited number of processes) with its dependencies so that it runs in its own isolated sandbox.

Containerization helps to ensure the application or set of processes can run reliably regardless of the host environment. The container shouldn't be able to modify or interact with it anything it doesn't need and, on the whole, changes in the container should not effect the Host or other containers (and vice versa).

The following diagram illustrates the structure of containers- an application in a container is isolated to just what it needs.



## Foundations

### **Linux**

LinuX Containers, which provide the foundation for most modern container systems (including Docker), began with two key kernel tools in Linux: cgroups and namespaces

* Control groups (cgroups) is a kernel feature that allows you to exercise control over the resources used for a particular process-- via monitoring and limiting
  + Things like limiting the amount of memory for a process, determining the priority of a process- how much CPU time, and stopping/starting that process or set of processes.
* Namespaces
  + method of encapsulating a global resource
  + Different types:
    - mount: set of mount points exposed (First namespace type introduced)
    - PID: set of process IDs exposed, form a hierarchy
    - Net (network): set of network resources exposed; routing tables; socket port member names etc.
    - UTS: Unix timesharing system- defines visible host and domainname-- not related to DNS
    - User: Isolates the user ID and group ID-- can make a user seem like the super user for its namespace, but in actuality the user has limited privileges in the context of the greater system
    - IPC
    - Cgroup
    - and more
  + Namespaces determine what is visible to a particular process. A process begins as part of a single instance of each particular namespace type.
    - can only see a certain set of other process IDs, network resources, mount points etc.
    - namespaces also provide a layer of virtualization-- a process might see itself as having PID 1. In its namespace this may be "true", but there may be a namespace further up the heirarchy that understands this same process as having PID 312

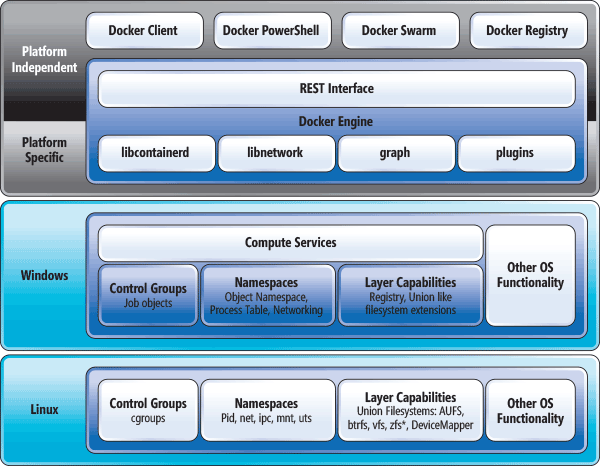
Take the cgroup management of resources + namespaces => the basics of a container

=> Isolation and Virtualization => Containerization

By packaging up an application in a container you ensure that it has all the needed dependencies and can modify/use only what it needs. You also ensure that the application is portable.

### **Windows Containers**

As Linux containers gained popularity, Windows worked to keep up and find it's own solution to support containerization. While Windows met with some challenges since their operating system setup is a bit different from Linux, they were able to ultimately support the same functionality through tools that were analogous to cgroups and namespaces. Though, because of the manner in through which certain resources must be shared, they also came up with another version of containers that's based on the same technology used by virtual machines. As they got things working and were able to break down the problems they faced, in 2016, the question became how to make these features available to users of the Windows OS. The solution became to partner with Docker who by that point was already a huge driving force in the move toward containerization.

As the diagram illustrates the docker engine runs upon the platform specific supports for containers within either the Windows OS or Linux OS.

## Containers

* Built from images (template for the container)
* Run on an engine (on the host OS) i.e. Docker Engine
* Ideally stateless
  + State needed to persist for an application should be stored in a way that is essentially "detachable" from the actual container- otherwise state only persists so long as a container is running- volumes-- solve this
* Virtualization
* Isolation

### **Benefits**

* Secure
  + Isolation and Virtualization keep your containerized apps more secure
* Standardized and thus Portable
  + Think write once run anywhere
* Lightweight
  + shares the host operating system's kernel
* Flexible and Loosely Coupled
* Scalable
  + Easy to spin up and because of this lightweight ease they can be scaled up quickly

**Installing Docker**

# Installing Docker

## References

* [Docker Installation for Windows Home](https://docs.docker.com/docker-for-windows/install-windows-home/)
* [Docker Installation for Mac](https://docs.docker.com/docker-for-mac/install/)

## Background

Docker has a number of components that work in tandem to bring you a standardized way of working with containerization technology.

**Docker Engine**:

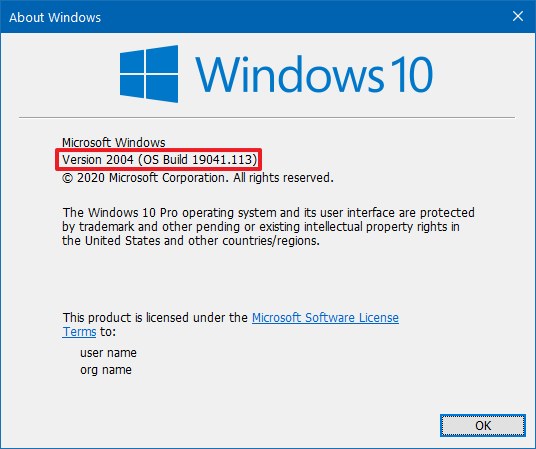
* Docker Deamon
* APIs that allow for interaction with the deamon -- importantly the CLI(Command Line Interface)

**Docker Desktop** is the manner through which you install the Docker Engine on Windows and Mac OS -- it includes:

* the Docker Engine
* Kubernetes
* Docker Compose
* Notary
* Credential Helper

Determine the OS and continue with the installation steps for that environment:

* Windows:
  + Type Windows Key + R
  + Then run the command winver
  + Make a note of Home or Pro OS as well as the version and build.
  + If you are on Windows 10 Home version 2004 build 19041 or later, then see [Docker Installation for Windows Home.](https://app.revature.com/)



* Mac:
  + [Instructions for checking the OS version are here.](https://support.apple.com/en-us/HT201260#:~:text=From%20the%20Apple%20menu%20%EF%A3%BF,version%20number%20to%20see%20it.)
  + If you are using hardware from 2010 or later and version 15.13 or later (Catalina, Mojave, or High Sierra) see [Docker Installation for Mac.](https://app.revature.com/)

COMMENTS

- should focus on installation on Windows10 20H1 with WSL2

- should focus on installation on macOS 10.15 Catalina

- add the basic setup for each environment using either command line or user interface

# Installation

### [**Docker Installation for Windows Home**](https://docs.docker.com/docker-for-windows/install-windows-home/)

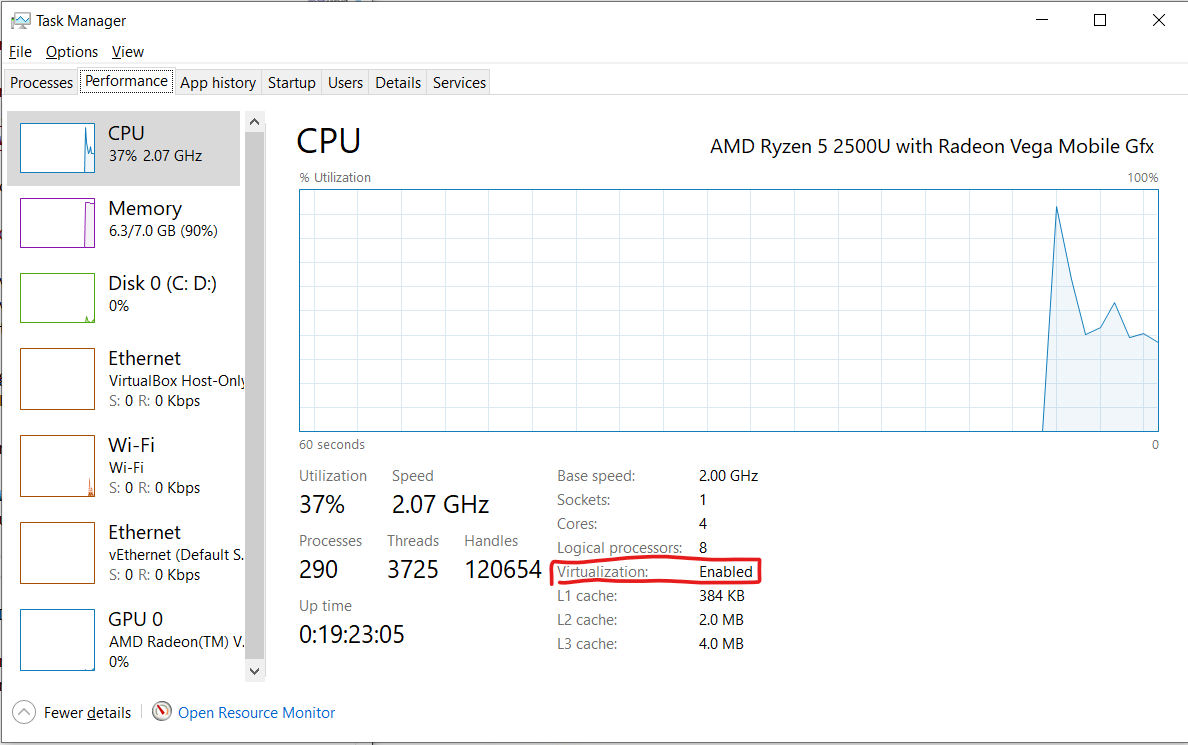
First, let's check our system requirements.

Go to This PC > Properties.

You should have more than 4GB of RAM and you should have a 64 bit processor. (In System Type it should begin 64-bit...)

Second, we must make sure BIOS-level hardware virtualization is enabled.

In order to check for this go to the task manager and click over to the performance tab. You should see the section Virtualization and it should say enabled.

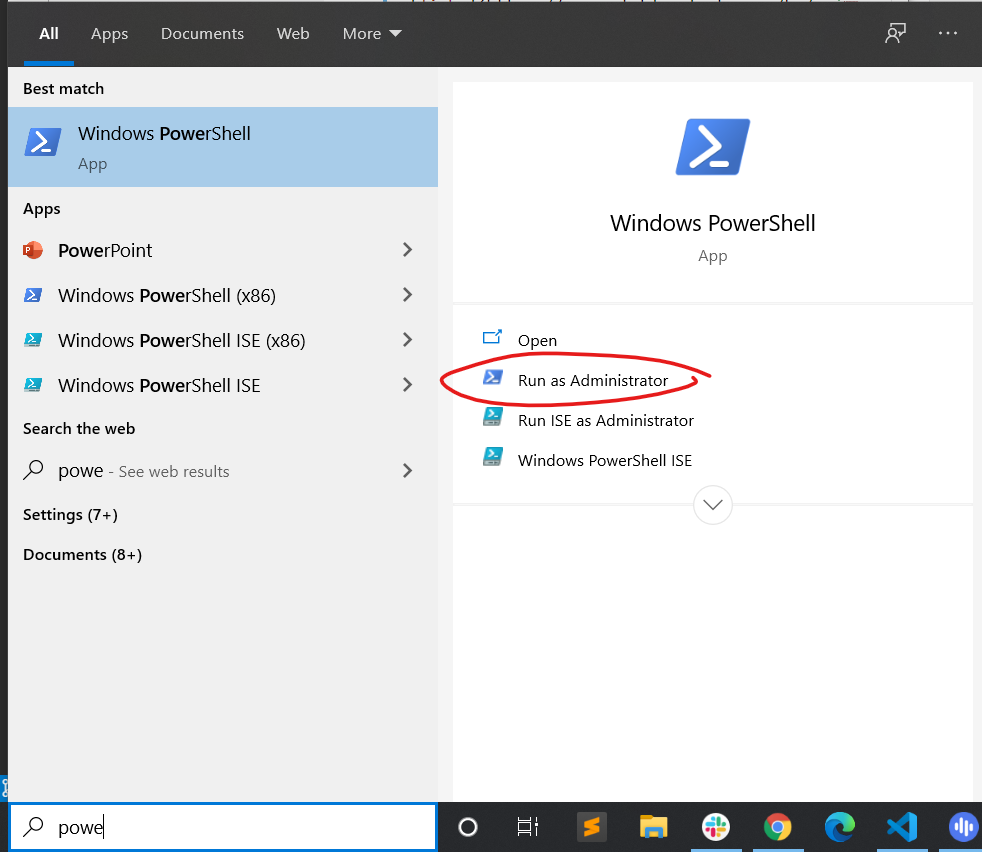


If the task manager page says that virtualization is disabled, then go to your BIOS menu to enable virtualization. This will look slightly different depending on your hardware, but you can access the BIOS menu through the instructions in this [link.](https://support.bluestacks.com/hc/en-us/articles/115003174386-How-can-I-enable-virtualization-VT-on-my-PC-#%E2%80%9C3%E2%80%9D)

If you have a hard time finding out how to enable virtualization from the BIOS menu, then you may need to do a google search of your manufacturer and the phrase "enable virtualization" to locate additional hardware specific instructions.

Note: Docker Desktop allows Windows users to use either Windows Containers (on a Windows OS) or via Hyper-V and virtualization creating Linux containers via a Windows Subsystem for Linux.

Now you'll need to enable WSL 2. (The second version of Windows Subsystem for Linux.) (Remember your build has to be later than 19041.)

Open PowerShell as an admin.

Then run the command.

dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

Then run.

dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

Finally restart your computer.

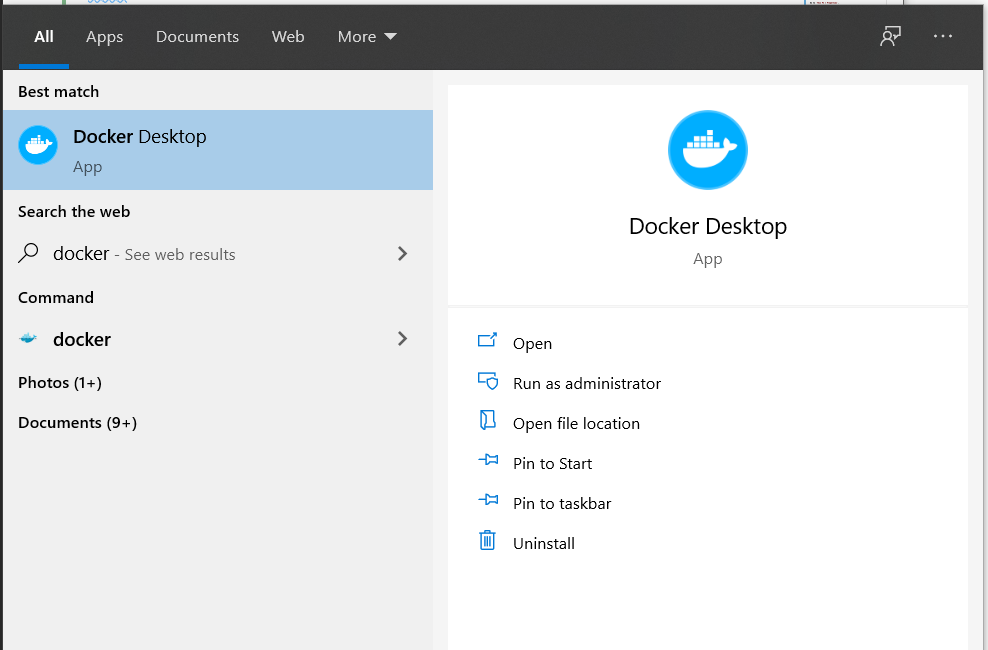
Finally you should be all set to download the installer for Docker from [here.](https://hub.docker.com/editions/community/docker-ce-desktop-windows/)

You will want the stable version.

Once the download completes, double click on the installer (the .exe file you just downloaded.) This will begin running the installer.

Select Enable WSL 2 Features when prompted.

Then follow the defaults and eventually select close to complete the installation process.

Now you must start up Docker Desktop. Go to Docker Desktop and select run.

Once the icon in the status bar is no longer changing, then you are ready to use docker. (It will look like the following image.)

docker status bar

The tutorial should automatically launch.

Additionally you can test docker out by opening up a command prompt and running the following command.

docker run hello-world

You should see something like

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

0e03bdcc26d7: Pull complete Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

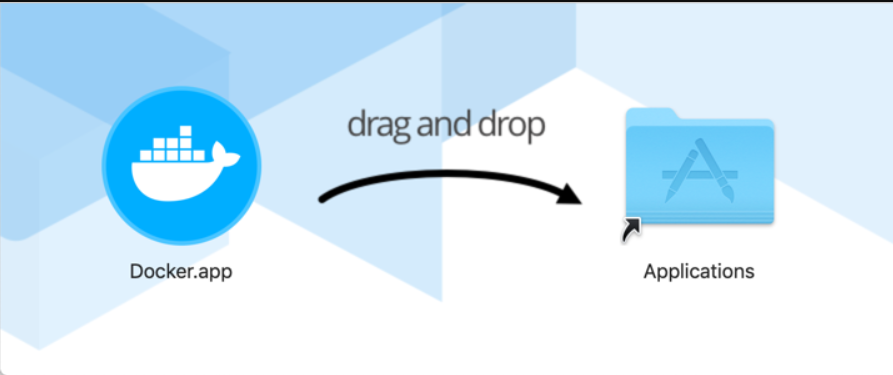
### [**Docker Installation for Mac**](https://docs.docker.com/docker-for-mac/install/)

**Note:** You must be working with a Mac newer than 2010 and your operating system must be one of the most recent three versions.

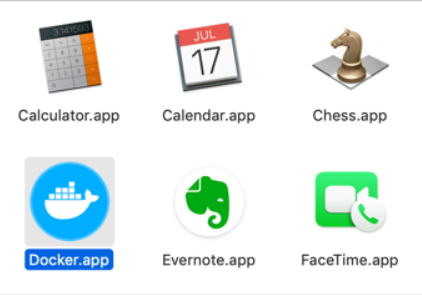
Go to [Docker Hub.](https://hub.docker.com/editions/community/docker-ce-desktop-mac/) And download the stable version.

Then double click on the Docker.dmg file you just downloaded to open the installer.

Drag and drop Docker into the applications folder.



Then, on the Docker.app in the Apps folder, double click.



You should see the little whale icon popup in the status bar and the on-boarding tutorial should begin.

status bar icon

Additionally, to test docker instead of or in addition to you can open up Terminal and run the following command.

docker run hello-world

You should see something like

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

0e03bdcc26d7: Pull complete Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

## Alternate instructions

### [**Docker Installation for WindowsPro**](https://docs.docker.com/docker-for-windows/install)

Paid upgraded version of Windows.

### [**Docker Installation for Linux**](https://docs.docker.com/engine/install/)

### [**Older Mac and Windows Machines**](https://docs.docker.com/toolbox/toolbox_install_windows/)

**Docker Architecture**

# Docker Architecture

## References

* [Docker Architecture from the Docker Getting Started Guide](https://docs.docker.com/get-started/overview/#docker-architecture)

## Overview

Docker adheres to a Client-Server Architecture. The client allows you to run commands and interact with the docker objects managed by the docker daemon, which is in effect the server.

This architecture, consequently, translates into these primary components that allow you to run and interact with Dockerized applications (applications that have been packaged up and run in Docker containers.)

* [Docker Daemon](https://app.revature.com/)
* [Rest API](https://app.revature.com/)
* [Docker CLI client](https://app.revature.com/)
* [Docker Registries](https://app.revature.com/)
* [Docker objects](https://app.revature.com/)

As you can see from the diagram the client includes the commands that manage docker containers and images. The docker host is the computer on which the docker daemon runs, and the docker daemon is responsible for managing the docker objects i.e. containers and images. The registry is an additional/optional component that provides a centralized place to share docker images. Finally, while the REST API is not labeled in the diagram it's effectively the lines connecting the client and docker daemon.

### **Docker CLI (Command Line Interface) client**

The command line interface is what Docker developers typically use for interactions with the Docker daemon.

It's prefaced by the docker command.

The CLI client may be on the same host machine as the Docker daemon with which it communicates, but it doesn't have to be.

All the commands outlined in the [docker workflow module](https://app.revature.com/) notes are effectively part of the client.

### [**Docker Daemon**](https://app.revature.com/)

The docker daemon is the long running process on the docker host that does all the heavy lifting of managing Docker objects- containers, images, etc. The docker daemon is at the core of running dockerized applications.

### **Rest API**

These are the underlying commands used by the CLI and other applications to interact with the Docker Daemon.

### **Docker registries**

Docker registries provide a centralized place to store images, allowing you to easily share images between docker hosts. The Docker Hub is a public registry managed by Docker that docker hosts can pull images from and push images to by default.

For more info see [these notes.](https://app.revature.com/)

### **Docker objects**

Docker objects are the building blocks that are managed by the docker daemon. The most fundamental docker objects are images and containers. Images are the templates that outline all dependencies for a particular container and it's primary process. Meanwhile the container is the runnable instance of a set of processes and their dependencies.

For more info on some of the key docker objects see the notes on [docker containers](https://app.revature.com/), [docker images](https://app.revature.com/), and [docker volumes](https://app.revature.com/).

#### **Typical Flow**

1. Using CLI commands (in your command line, Terminal, Command Prompt etc.) such as docker build ... or docker pull ..., acquire an image. This image lays out everything needed to create and run a container.
2. Behind the scenes the Docker daemon either pulls your image from a registry(repository of images) or it creates the image. (Depending on the command you gave.)
3. Then you use the CLI again with a command such as docker run .... (Note if you don't already have the image you were trying to run then the command will include that first step of pulling the image implicitly.) If you didn't already do this during the installation process, try the following command.

docker run hello-world

1. Thus, the CLI once again instructs the daemon. This time it tells the daemon to spin up a container from the image.
   * The application or set of processes in the container are tied to its life cycle and have started with its instantiation.
2. Finally, it's just a matter of using additional CLI commands to manage the now running container, unless the primary process of the container has already finished. In that case the container will exit on it's own.

**Docker Daemon**

# Docker Daemon

Long running process that is responsible for managing Docker objects.

dockerd command allows you to run and configure the docker daemon (working with flags and config files)

Listens for calls from the exposed REST API to manage containers, images, volumes etc.

Can communicate w/other Docker daemons -- in the case of Docker Services -- can be used to set up a swarm. (A swarm allows you to set up multiple docker daemons the behave like a cluster.)

#### **Daemon Configuration**

* Flags can be used when running the dockerd command
  + i.e. dockerd --debug
  + above specifies to run the daemon in debug mode
* JSON file
  + /etc/docker/daemon.json on Linux systems
  + C:\ProgramData\docker\config\daemon.json on Windows
  + MacOS go to the whale in the taskbar > Preferences > Daemon > Advanced

The file might look as simple as:

{ debug: true }

* Each configuration options should be configured using either a flag in conjunction with the dockerd command or in the config file.
* Persists all Docker data in a particular directory
  + By Default
    - /var/lib/docker on Linux
    - C:\ProgramData\docker on Windows

Note: In a Windows System it may not be clear to the OS where to find the dockerd.exe to run the Docker Deamon and it's command. The Docker Desktop app will help your OS to understand docker without any extra effort. To find the dockerd.exe (to run the dockerd command specifically) look under the resources folder in with in the Docker folder in Program Files.

### **References**

* [dockerd command reference docs](https://docs.docker.com/engine/reference/commandline/dockerd/)

**Docker images**

# Docker Images

Blueprint for a container

[This video from VMWare draws an analogy to docker images being like Java classes with containers being analogous to Java objects](https://www.youtube.com/watch?v=EnJ7qX9fkcU&t=4s)

Images form a kind of heirarchy. One image will be "From" another with added info, dependencies, commands, applications, etc. The added info and command each form a new layer on the image. With each of these layers being indicated in a the Dockerfile that defines what's needed for the image.

Images are named and tagged with the version. They also have an id which uniquely identifies them.

## Existing Docker Images

Pull images from some existing registry(repository of images). The default configuration is from the DockerHub.

* docker pull \*image name\*
* docker run \*image name\* (this will pull the image if it doesn't already exist in the local system)

## Building Our Own Images

* Dockerfile
* From existing container
  + docker commit

We can then push our images to a given registry including DockerHub

## Managing Images

We can use the docker CLI to manage the images on our local system. We can list out the existing images, get their details, remove and update them.

Additionally, we can use the CLI to aid in connecting to a registry to quickly and easily distribute changes. In this case updating the software is as simple as updating the image.

Users can easily pull new images and spin up containers/applications with the modifications made.

#### **References**

* Video linked to above from VMWare (This is also in the Containerization References)
* [References listed in Refernces in Containerization Refs.](https://app.revature.com/)

**Docker Containers**

# Docker Containers

Runnable isolated instance of a set of processes and their dependencies.

A Docker container is built from a Docker image. The image lays out everything the processes that run in the container will need.

Docker Containers are managed by the Docker Daemon as part of the Docker Engine. The Docker Engine allows Docker containers to be standardized and very portable.

### **Under the hood**

The underlying nature of Docker containers is based on the capabilities provided by namespaces and cgroups. Docker containers also take advantage of a file system called UnionFS. Docker manages all this in tandem in a wrapper refered to as container format. The container format used by default is libcontainer.

Docker containers when run on a Linux system typically share the Host OS- just as one would expect of a containerized app. The goal is lightweight after all.

However, in the case of Windows, Docker containers may use an additional layer of virtualization enabling you to run Linux containers on a Windows OS. This is why it's necessary to have Hyper-V and virtualization enabled when trying to install Docker on a Windows OS. (Because truly it is akin to running a container in a VM.)

### **Benefits**

And they allow for all the benefits outlined in the containerization notes. i.e.(copied verbatim for convenience)

* Secure
  + Isolation and Virtualization keep your containerized apps more secure
* Standardized and thus Portable
  + Think write once run anywhere
* Lightweight
  + shares the host operating system's kernel
* Flexible and Loosely Coupled
* Scalable
  + Easy to spin up and because of this lightweight ease they can be scaled up quickly

### **States of a container:**

* created
* restarting
* running
* paused
* exited
* dead

### **References**

* <https://docs.docker.com/engine/api/v1.24/#31-containers>
* Docker docs <https://docs.docker.com/engine/reference/commandline/container/>

Day 4

**Dockerfile**

# Dockerfile

Defines everything needed for an image. It outlines the starting point, dependencies and commands that make up all the processes needed for an image and in turn a container.

Dockerfiles go step by step.

Dockerfiles always begin with the instruction FROM image name. (Technically a dockerfile might have parser directives, ARG commands, or comments first. )

* The command FROM indicates the image that you start from, either:
  + a parent image some pre-existing image the container is based on
  + a base image which is when the container is built from the command FROM scratch

Note, while technically a base image and parent image are different you might hear them used interchangeably.

After that essentailly each instruction forms anothe layer of the docker image. (These layers are cached to speed up the build.)

Dockerfile commands include things such as installing software in the container, copying and adding files to the container from local or remote systems, defining environment variables needed for the container, and executing commands such as running the intended app in the container. [More on dockerfile commands.](https://app.revature.com/)

### **References**

* [Dockerfile Best Practices](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
* [Dockerfile Reference](https://docs.docker.com/engine/reference/builder/)
* [Docker glossary](https://docs.docker.com/glossary/)

**Docker volumes**

Docker Volumes

References

* <https://docs.docker.com/storage/volumes/>
* <https://docs.docker.com/storage/>

Overview

**Volumes** are a way to persist data for a container.

Typically the goal is to have containers exist as mostly stateless. However, sometimes you need information for a container to remain even if the container stops.

Volumes are managed using the CLI and the Docker API.

(Note: While there are options for how to persist data, volumes are preferred.)

They facillitate

* sharing data between many different containers
* decoupling of host and container
* storing data remotely
* moving data between hosts or backing up data between hosts

Volumes are also helpful because they allow Docker to keep the containers slim by saving data in the volume rather than the writable layer that dissappears with the container.

**Docker best practices**

# Docker Best Practices

### **References**

* [Docker Guide- Dockerfile Best Practices](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)
* [Docker Guide- Docker Best Practices](https://docs.docker.com/develop/dev-best-practices/)

### **Goals**

* "ephemeral" containers-- containers should be as easy as possible to tear down and build up, requiring minimal configuration
* lightweight containers and images
  + "ephemeral" and lightweight go hand in hand, but additionally lightweight relies on a couple of other best practices

### **Build Context**

The working directory (including subdirectories) is sent over to the Docker daemon when the image is created i.e. the docker build command is run.

Thus, it's important to be mindful of what is in that directory. The less that's there the faster the process and lighter weight the image.

### **Leverage multi-stage builds and image cache**

Previous images are cached, since images are layered this can be leveraged to dramatically speed up building an image. It reduces the amount of times that you must pull from remote storage or rebuild image layers.

\*\* Be mindful of the cache though as some layers maybe cached and not properly updated. For instance, RUN apt-get update will be read as the same string every time and will not be run again if the cached image was built in exactly the same way with no other changes.

### **Additional Best Practices:**

* Least number of ultimate layers for an image possible
  + These will be defined by the different commands in the dockerfile
* Each Container should ideally serve one sole purpose and applications should be decoupled as much as possible
* Make commands in dockerfile readable by separating them on to different lines with \ (the escape character) and by keeping those lines organized in some fashion
* Use volumes for persistent data
* Use secrets for sensitive data and config files for configurations that are not sensitive

**Dockerfile keywords**

# Dockerfile Keywords

## References

* [Dockerfile Reference](https://docs.docker.com/engine/reference/builder/)

## Dockerfile Commands

FROM image name

* specifies the parent image from which the new image should be based.
* forms the first layer of this new image.
* might say FROM imagename AS temp-name if we want to have use multi-stage builds
  + This will essentially create our final image by building on the previous images that we set up in the same dockerfile.
* i.e. If we wanted to start with a ubuntu OS base (where we would take advantage of multi-stage builds) we might have:

FROM ubuntu AS ubuntu-with-java-example

RUN

* There are two forms of the command.
  + RUN <command>
    - Runs the command in a shell by default (the particular shell depends on whether the parent image defines a linux or windows container)
  + RUN ["executable", "param1", "param2" ]
    - Runs the command in the executable form without using a shell
* RUN will be used to set up your image- the state of the image after each run command is committed- forming a new layer
* i.e. If we wanted to install some programs as part of the image, you might have:

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

ADD <src> <destination>

* adds files from build context or url to the image
* copy is preferred over add
* can perform auto-extraction into the image from a local tar file (this differs from copy and is one of the few instances add is preferred) -i.e. The following command would copy HelloWorld.java into a file of the same name in the container's working directory.

ADD HelloWorld.java HelloWorld.java

COPY <src> <destination>

* adds files from the build context to the image -i.e. The following command would copy HelloWorld.java into a file of the same name in the container's working directory.

COPY HelloWorld.java HelloWorld.java

EXPOSE

* outlines the ports that the are being listened on by processes in the container-- i.e. it suggests what ports to bind to host ports when running the image -The following example would inform the host to bind to port 80 in the container

EXPOSE 80

VOLUME ["/nameofdir"]

* creates a mount point in the image and thus container with a particular name- it indicates that the files in this directory will be shared with the resources outside of the container
* it indicates what directory to connect a volume to when running the docker container -i.e. The following suggests that we should connect a volume to the data directory in the container when we spin it up.

VOLUME ["/data"]

WORKDIR <nameofdirectory>

* sets the working directory in the image and eventual container of commands that follow. i.e.

WORKDIR /example

RUN mkdir a

* Would make a directory a inside of the example folder. example/a

CMD

* used to run the app, processes etc. needed inside of your container
* only the last CMD will run when the built image is launched as a container
* CMD ["executable","param1","param2"] (exec form, this is the preferred form)
  + invokes the command with out a shell
* CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
  + the ENTRYPOINT instruction must be specified if you use the default format, it's another way you can specify the first commands to be run upon spinning up a
* CMD command param1 param2 (shell form)
  + invokes the command inside of a shell in /bin/sh -c
* i.e. The last line of a docker file might be a command running a java program:

CMD ["java", "HiWorld"]

## Dockerfile Examples

**NOTE**: The dockerfile will typically be saved as Dockerfile without any extension. (Save-as all files.)

Let's look at this dockerfile that runs a simple Java HelloWorld example:

# Define the parent image

FROM ubuntu

# Install needed programs

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

# Within the image and thus container, set the working directory to the new directory example

WORKDIR /example

# Create Hello World Java program and save it in the appropriate file

RUN echo 'public class HiWorld{ public static void main(String[] args){System.out.println("Hi world");}}'> HiWorld.java

# Compile the Java program, creating the file that the JVM can actually run

RUN javac HiWorld.java

# Run the HelloWorld program in the container

CMD ["java", "HiWorld"]

### **Description of Example:**

The first line of the Dockerfile indicates the parent image will be the official ubuntu image. It's common to see an image begin with a base OS and then install any needed programs on top of it. (Though typically, if java is the only needed program, one would might choose to use openjdk or some image that starts with java.)

The next lines that begin RUN ... install the needed jdk.

WORKDIR sets directory for any subsequent commands to be executed from the /example folder.

Thus, the lines RUN echo... and RUN javac ... create the program that we will actually run. With more complex programs its likely you would use something like COPY to takes the whatever files are needed from the local system and copies them into the destination.

Finally, the CMD line will execute the command java HiWorld from the example directory when the container starts up. It's important to remember that the lifecycle of the container will be tied to this initial process and the container will only exist long enough for the HiWorld program to run.

We can also expand on this to create this same example taking advantage of multi-stage builds.

# Define the parent image and give a name to the first stage of building our image

FROM ubuntu AS ubuntu-with-java-example

# Install needed programs

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

# This time we are building from the image that we just defined

FROM ubuntu-with-java-example

# Within the image and thus container, set the working directory to the new directory example

WORKDIR /example

RUN echo 'public class HiWorld{ public static void main(String[] args){System.out.println("Hi world");}}'> HiWorld.java

RUN javac HiWorld.java

# Run the HelloWorld program in the container

CMD ["java", "HiWorld"]

**Building an image**

# Building an Image

## References

* [Docker CLI reference](https://docs.docker.com/reference/)
* [docker build](https://docs.docker.com/engine/reference/commandline/build/)
* [docker commit](https://docs.docker.com/engine/reference/commandline/commit/)
* [docker file](https://docs.docker.com/engine/reference/builder/)

An image will typically be built from a dockerfile, though it can be built from a container that's already running and has been modified. The former is preferred, but the latter can be helpful in a development environment.

## Create image with build

> docker build anyflags PATH

* docker build indicates to the CLI that you are going to create a new image. You have the option of adding flags for additional information and configuration. A couple flags to note:
  + -f path indicates that the dockerfile to be used is in the location indicated from the subsequent path (it will be used when that path is different from the build context.)
  + -t name:version allows you to name the image being created and indicate the version. If you were to just have -t name it would default to having the version marked as latest.
* The path specified at the end of the docker build command will be understood as the build context. It will often just be . to indicate the current directory. Though it could be a git url or even a path to another part of your filesystem.

### **Example**

Create and save the following docker file in a new empty folder. Perhaps you create a folder docker-examples in your Documents folder.

# Define the parent image

FROM ubuntu

# Install needed programs

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

# Within the image and thus container, set the working directory to the new directory example

WORKDIR /example

# Create Hello World Java program and save it in the appropriate file

RUN echo 'public class HiWorld{ public static void main(String[] args){System.out.println("Hi world");}}'> HiWorld.java

# Compile the Java program, creating the file that the JVM can actually run

RUN javac HiWorld.java

# Run the HelloWorld program in the container

CMD ["java", "HiWorld"]

From the same directory in which the dockerfile has been saved run the subsequent command. (If you saved it in the directory as described above then you would need to run cd Documents\docker-examples)

> docker build -t java-hello-world .

Do not forget the . at the end.

There are a couple of things to take notice of here. First, the dockerfile need be the only thing in its directory from which we run the build command. That being said, typically you will also have other files in that directory that are needed for the image.

If we wanted to build the same docker image, perhaps from another folder entirely, then we would have the following command:

>docker build -f "C:\Users\UserName\Documents\Dockerfile" -t java-hello-world .

**Note**: Don't forget the . at the end- indicating that the current context is your build context.

## Create image with commit

Assume that you have a container that is already running and you have made some changes to it. Maybe you have installed needed software or changed some configuration.

Then you can commit these changes to a new image that can be used to spin up even more containers.

Note the new image won't include any data saved in the volumes of your container.

> docker commit flags CONTAINER imagename

* Essentially you are commiting the changes from the container specified to the image specified.
* It is typically preferrable to use a dockerfile, but there are some use cases like debugging where this is helpful.
* There are a number of flags you can use. They include:
  + -c will allow you to execute/apply Dockerfile instructions
  + -m message to include when committing changes
  + -p pauses the container whose state you are committing to the new image

For example imagine you installed some new software and you wanted to create a new image from your current running container:

>docker commit -c "WORKDIR /new-example" -m "added new technology" -p 1f7e0cf664ca my-fabulous-image

## Imgage management

To list all the images.

* docker images
* docker images -a
  + -a allows you to list even hidden images.

**Creating a container**

Creating Containers

References

* [docker create imagename](https://docs.docker.com/engine/reference/commandline/create/)
* [docker run imagename](https://docs.docker.com/engine/reference/commandline/create/)

Introduction

As we approach creating a container from our image, there are two paths we can take. In the instance where to set up the container requires a greater amount of configuration, it may be beneficial to create the container first and then move to the starting it up once prepared to do so. In most instances, however, we will build and run the container in one fell swoop with a single command. The former approach uses docker create and the latter uses docker run. Let's explore these ideas in more detail.

> docker create imagename

* can also be written as docker container create imagename
* creates a container that is in the "created" state - it configures and sets it up to be run including creating the *writable layer* on the image from which the container is created
* it's useful if there is some configuration of your container that you want to have ready, so you can easily just start up this new container
* it prints out the id of the newly created container
* flags:
  + -p hostport:containerport publish ports creates a mapping from the host port to the container ports
  + --env VARIABLE=value set environment variables of the container
  + -v hostdirectory:containerdirectory create a volume between the container and the host machine -- shared memory that persists past the given container's lifecycle
  + For example, let's pull an existing basic web app from Microsoft and create a container as described above.
* > docker docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
* > docker create -p 80:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

Then typically you would then start the container.

> docker start firstthreecharactersinthecontainerid

If you follow the example above- navigate to localhost:80 on your web browser and see that your app has been started - with the container's port 80 being mapped to port 80 on your local machine.

> docker run flags imagename

* Can also be written docker container run flags imagename
* First pulls the image from the registry(repo of images) if it doesn't already exist locally.
* Then it both creates and runs the container-- this brings a container directly into the state of *running*.
* Recall that when you run a container its processes and environment are in their own isolated sandbox.
* To interact with a container's processes and work inside of its environment, you must tie the container to your own current environment- usually by having your commandline tied to a commandline within the container.
* Many of the flags influence the ways in which the container can be tied to your host environment and/or the way the initial container environment is set up, highlights include:
  + -d
    - Indicates to run the container in the background
    - If this flag is *omitted* the shell from which you are running the CLI commands will run in the *foreground*
      * That is, STDOUT and STDERR will be bound to the shell by default. Thus, anything that's printed to STDOUT/STDERR from within the container will display in the current shell. This does not mean that you can interact with Try to test the difference between running:
  + > docker run hello-world

And

> docker run -d hello-world

* + - The container's id prints out instead of the container's output.
    - In the second case even though the "Hello World..." output doesn't print, we can view the logs to see that it did in fact run and do what it was supposed to.

(viewing the logs is explained in [managing containers](https://app.revature.com/), but for now run the command with containerID replaced with the output from earlier.)

> docker container logs containerID

* + -i
    - allows you to run the container in interactive mode
    - it will keep the STDIN stream open even if your shell isn't attached to it.
    - it's needed if you want to be able to type input to your container dynamically
  + -t
    - is related to TTY and essentially is used to create some kind of terminal or shell
  + -it
    - This is the combination of the previous two commands and will be very frequently used when you want to open up a container with its primary processes being an interactive shell.
    - This allows you to basically work within the container from your commandline. For example try running:
  + > docker run -it ubuntu

Then you can try typing commands in your containerized ubuntu shell that you'll have open. You are essentially working inside of the container.

* + -a standardstream
    - this allows you to attach your commandline to a particular standard stream: STDIN, STDOUT, or STDERR
    - if you want to attach multiple then you will need multiple -a each followed by the particular stream
    - Also note, just attaching STDIN doesn't automatically make your container interactive. For that you must use the -i flag.
    - For example try testing the differences between attaching to just the error or just the standard output stream
  + > docker run -a STDERR -p 88:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

You should see your shell kind of paused- as though it's looking for some output, but not receiving any. Then stop your container- see [managing containers.](https://app.revature.com/)

>docker stop containerid

Then create and run a new container from the same image attaching a different stream:

>docker run -p 88:80 -a STDOUT mcr.microsoft.com/dotnet/core/samples:aspnetapp

Notice how now the standard output from your container displays in the shell.

* + -v name:directoryincontainer
    - Used to create a volume
    - To create a volume you name the volume and then you specify its corresponding path that will be linked within the container (the path must always begin with a / ).
    - Using this flag you can specify an absolute path on the host machine instead of a name (beginning with /), but that will create a *bind mount*.
    - volumes will persist after the container no longer exists
  + > docker run -it -v first\_volume:/example ubuntu

If inside of the newly created ubuntu shell, run

$ cd example

$ echo 'Hello World' **>** text\_file.txt

And then stop that container(CTL+C or CTL+D to exit the shell and thus stop the container's primary process exiting the container completely). Now spin up another one with the same volume

> docker run -it -v first\_volume:/data ubuntu

And then run the command

$ cat /data/text\_file.txt

It will print

Hello World

Notice/experiment with how this differs from if you created another ubuntu container from the ubuntu image without specifying the volume. It would not by default contain the folders. Those were only created and available because of the way that the volumes were configured.

* + -p portonhost:portincontainer
    - This command publishes a port in the container to a host port. This creates a tie between the two of them.
  + docker run -p 90:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

Then go to localhost:90 in the web browser to see site being hosted in the container.

* + --env VARIABLE=value
    - set the environment variables for the container
  + >docker run --env VAR1=value1 --env VAR2=value2 ubuntu env

Notice that the a whole list of environment variables is displayed, including the ones you just set.

**Managing Containers**

# Managing containers

## References

* [Docker CLI reference](https://docs.docker.com/engine/reference/commandline/cli/)
  + [docker container ls](https://docs.docker.com/engine/reference/commandline/container_ls/)
  + [docker ps](https://docs.docker.com/engine/reference/commandline/ps/)
  + [docker container exec](https://docs.docker.com/engine/reference/commandline/container_exec/)
  + [docker container logs](https://docs.docker.com/engine/reference/commandline/container_logs/)
  + [docker container port](https://docs.docker.com/engine/reference/commandline/container_port/)
  + [docker container kill](https://docs.docker.com/engine/reference/commandline/container_kill/)
  + [docker container stop](https://docs.docker.com/engine/reference/commandline/container_stop/)
  + [docker pause](https://docs.docker.com/engine/reference/commandline/pause/)
  + [docker unpause](https://docs.docker.com/engine/reference/commandline/unpause/)
  + [docker container rm](https://docs.docker.com/engine/reference/commandline/container_rm/)
  + [docker container prune](https://docs.docker.com/engine/reference/commandline/container_prune/)
  + [docker volume rm](https://docs.docker.com/engine/reference/commandline/volume_rm/)
* [Introducing docker 1.13](https://www.docker.com/blog/whats-new-in-docker-1-13/)
* [SIGTERM and SIGKILL](https://linuxhandbook.com/sigterm-vs-sigkill/)
* [cgroup freezer](https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt)
* [Container names and ids](https://docs.docker.com/engine/reference/run/#name---name)

## Contents

* [List Containers](https://app.revature.com/)
* [Run Commands](https://app.revature.com/)
* [View Container Logs](https://app.revature.com/)
* [List Port Mappings](https://app.revature.com/)
* [Stop Containers](https://app.revature.com/)
* [Pause Containers](https://app.revature.com/)
* [Start Containers](https://app.revature.com/)
* [Remove Containers](https://app.revature.com/)
* [Remove Volumes](https://app.revature.com/)

## Introduction

When a docker container is first created, it's assigned a name and a container id. You can use these to manage the state of the container and the processes running inside of it. The name is a unique string- either generated or assigned by you using a flag. Meanwhile an id is a UUID identifier that will always come from the docker daemon.

As you know, in any given container there is one primary process to which the lifecycle of the container is tied and there may be any number of other adjacent processes that run in the container as well.

These basic container management commands allow us to check in on and manipulate both the containers and the processes running within them.

**Note**: In most cases the container id can be replaced with the first three characters of the container's id or the container's name.

## Container Management Commands

### **List containers:**

* running containers:
  + docker container ls
  + docker ps
* all containers
  + docker container ls -a (or --all instead of -a)
  + docker ps -a
* These commands will specify the container id's, when they were created, their state, from what image repository they came, a name, etc.
* This really comes in handy to identify the particular container id you need for subsequent commands.
* There is no difference between the output of ps and ls. Though docker ps came first and docker has expressed interest in at some point moving toward grouped commands. (The commands where you begin docker logical object command- i.e. docker container ...)
* ps stands for process status while ls stands for list

### **Run a command in a running container:**

* docker container exec flags container-id command
* The container must be running.
* You can identify the container with it's full id or the first three characters if they are unique.
* flags might include things like it if the command you wanted to run would run a commandline in a particular container
* flags might also include -a STDOUT where STDOUT could be STDOUT, STDIN, STDERR depending on which stream you wished to attach to the shell from which you run the exec command

docker container exec -it 563 bash

* This would run an interactive bash in container 563.

### **View a container's logs:**

* View a container's output
* docker container logs container-id
  + The logs of a container include anything written by the container's processes to STDOUT or STDERR
  + the container does not have to be running for you to view the logs

docker container logs 563

### **List port mappings between container's ports and hosts:**

* docker container port containerid
  + This is useful when you need to determine what ports a container is listening on

docker container port 563

### **Stop a container:**

* docker container kill containerid
  + the main process, to which the container is tied, receives a SIGKILL
    - a SIGKILL forces a process to terminate along with any of its threads
    - it cannot be ignored or handled gracefully
* docker container stop containerid
  + the main process, to which the container is tied, receives a SIGTERM
    - the polite way to ask a process to terminate- allowing a process to handle shutting down as it sees fit- including ignoring the signal
  + if this fails after some grace period will send a SIGKILL to the same main process
* [also see remove a container](https://app.revature.com/)

### **Pause and unpause a container:**

* docker container pause containerid
* docker pause containerid
  + to pause a container pauses all the processes running within the container
  + on a Linux machine, under the hood, this command uses the cgroup freezer --- this means the processes are unaware they are being paused
  + docker container pause ... is just the newer version of the the docker pause ... command
* docker container unpause containerid
* docker unpause containterid
  + to unpause a container---unpausing all running processes within the container
  + on Linux, also uses cgroup freezer to begin running the processes again without them knowing they had ever stopped

### **Start a container**

* docker container start containerid
  + start a created container that is not running
* docker container restart containerid (if the container may or may not already be running)
* starts main process to which container is tied

### **Remove a container:**

* docker container rm flags containerid
  + note optional flag -f to force the removal of a running container
  + note optional flag -v to remove anonymous volumes with a container-- this will not remove named volumes that you have created
* docker container prune
  + Remove exited containers

### **Remove a volume:**

* docker volume rm volumename
  + **Note:** You cannot remove volumes that are being used by a container
  + to remove a volume removes the container's access to that named persistent and shared memory
  + [See the notes on volumes.](https://app.revature.com/)

**Docker Command Cheatsheet**

Docker Command Cheatsheet

References

* [Docker CLI reference](https://docs.docker.com/reference/)
  + [docker build](https://docs.docker.com/engine/reference/commandline/build/)
  + [docker commit](https://docs.docker.com/engine/reference/commandline/commit/)
  + [docker ps](https://docs.docker.com/engine/reference/commandline/ps/)
  + [docker container exec](https://docs.docker.com/engine/reference/commandline/container_exec/)
  + [docker container logs](https://docs.docker.com/engine/reference/commandline/container_logs/)
  + [docker container port](https://docs.docker.com/engine/reference/commandline/container_port/)
  + [docker container kill](https://docs.docker.com/engine/reference/commandline/container_kill/)
  + [docker container stop](https://docs.docker.com/engine/reference/commandline/container_stop/)
  + [docker push](https://docs.docker.com/engine/reference/commandline/push/)
  + [docker pull](https://docs.docker.com/engine/reference/commandline/pull/)

List of major commands and relevant flags. All of these commands can be found in other places in the lecture notes, so their detailed explanations are omitted.

Also many of these commands have additional flags. Only the most important/relevant are listed here.

Create Images

These commands can be used to create images either from running containers or from a dockerfile. [See notes on building an image.](https://app.revature.com/)

* **docker build**
  + used to create an image from a dockerfile.
  + Example:
* > docker build -t java-hello-world .
  + Flags:
    - -t imagename:version
    - -f dockerfilepath
* **docker commit**
  + used to create an image from an existing container.
  + Example
* > docker commit 586 java-hello-world
  + Flags
    - -c
    - -m
    - -p

Manage Docker Images

The following commands can be used to manage images on your local machine as well as on the DockerHub. [See notes on DockerHub.](https://app.revature.com/) [See notes on building an image.](https://app.revature.com/)

* **docker images**
  + list all the images on your local machine
  + Example
* >docker images -a
  + Flags:
    - -a
* **docker pull**
  + retrieve an image from the DockerHub
  + Example
* >docker pull ubuntu
* **docker push**
  + share your image to DockerHub
  + Example
* >docker push revature/java-hello-world:1.0

Manage Docker Containers

These commands can be used to start and manage docker containers. See [notes on creating containers.](https://app.revature.com/) See [notes on managing containers.](https://app.revature.com/)

* **docker run**
  + same as docker container run
  + create and run a container from an image
  + if the image is not already on your local machine, then pull it from the DockerHub
  + Example
* >docker run -it ubuntu
  + Flags
    - -d
    - -it
    - -p
    - -v
    - env
* **docker ps**
  + list all containers
  + Example
* >docker ps -a
  + Flags
    - -a
* **docker container logs**
  + view the standard error and output of a container
  + Example
* >docker container logs 586
* **docker container exec**
  + execute a command in a running container
  + Example
* docker container exec -it bin/bash 586
  + Flags
    - -it
* **docker container port**
  + list port mappings between a container and its host
  + Example
* >docker container port 563
* **docker container stop**
  + stop a particular container
  + Example
* >docker container stop 563
* **docker container kill**
  + forcibly stop a container
  + Example
* >docker container kill 563
* **docker container rm**
  + remove a particular container
  + Example
* >docker container rm -f -v 563
  + Flags:
    - -f
    - -v
* **docker volume rm**
  + remove a volume
  + Example
* >docker volume rm my\_volume
* **docker container prune**
  + remove exited containers
  + Example
* >docker container prune

**Docker Compose**

# Docker compose

## References

* [Docker compose](https://docs.docker.com/compose/)
* [Docker references](https://docs.docker.com/reference/)
* [Differences between versions](https://docs.docker.com/compose/compose-file/compose-versioning/#upgrading)
* [Docker compose networking](https://docs.docker.com/compose/networking/)
* [Docker compose volumes](https://docs.docker.com/compose/compose-file/#volumes)

## Overview

Docker compose is the tool that makes creating and managing multi-container applications easier.

It's fundamental use is based around the docker-compose file.

The docker-compose file allows you to run multiple containers in a way that eases otherwise cumbersome configuration. Moreover it makes it easy to set up those containers to talk to one another. It's a YAML file(.yaml or .yml) that includes key information including environment variables, ports, volumes etc.

The file is typically named docker-compose.yml or docker-compose.yaml.

The file can be used simply with the docker-compose up command. This searches the current directory for the appropriate file and spins up the containers outlined.

## Docker-compose file components

Docker compose files have an overarching format that has evolved from version to version. There are three overarching versions of the docker-compose file, 1.x, 2.x and 3.x, with smaller upgrades constituting the second part of the version number. This document discusses version 3, but both version 2 and 3 follow the same general format of services, networks, and volumes. There are just some differences in the options/parameters available between the two versions.

Each service definition is analogous to the docker run command. Meanwhile the networks and volumes definitions are analogous to docker network create and docker volume create.

**Compose file options**:

* [version](https://app.revature.com/)
* [services](https://app.revature.com/)
  + [Options for a service](https://app.revature.com/)
    - [image](https://app.revature.com/)
    - [build](https://app.revature.com/)
    - [ports](https://app.revature.com/)
    - [environment](https://app.revature.com/)
    - [env\_file](https://app.revature.com/)
* [volumes](https://app.revature.com/)
* [networks](https://app.revature.com/)

### **Version declaration:**

The docker-compose file begins with the version declaration. It's the first line of the file and helps the docker compose tool understand/process the file correctly.

For example if we were to use version 3.8. Then we would have:

version: '3.8'

### **services**

The various docker containers that you spin up via a docker compose file each constitute a particular service.

At the top level of the docker compose file, you specify a services option and then all the information for each service definition is specified under that service's name (under the services option). For instance:

version: '3.8'

services:

db:

#all the information for the database set up

webapp:

#all the information for the webapp setup

...

#### **Options for a particular service**

The following constitute a foundational list of the options for configuring a particular service.

##### **image**

The image from which the service ought to be built. For example, it might look like this

version: '3.8'

services:

db:

image: postgres:latest

##### **build**

The build option allows you to outline how to build a service from a dockerfile. It allows you to specify the configuration needed at build time. This includes information like context, arguments and dockerfile.  
For example, in the simplest case, where the dockerfile and context are the same as the directory of the compose file, you might have:

version: '3.8'

services:

db:

build: .

This indicates to build the database from the context of the current directory and a dockerfile within it.

##### **ports**

Allows you to expose ports in the container for the particular service and to map them to ports on the host. The short version of the option syntax is ports: host:container. So for example, the declaration might look like this, if postgres was listening in the container on port 8080:

version: '3.8'

services:

db:

image: 'postgres:11'

ports:

- "80:8080"

...

##### **environment**

Allows you to specify environment variables for a particular service. For example, you might have

version: '3.8'

services:

db:

image: 'postgres:11'

ports:

- "80:8080"

environment:

- POSTGRESQL\_HOST=postgresql

- POSTGRESQL\_ROOT\_USER=postgres

##### **env\_file**

Allows you to specify environment variables for a particular service as the contents of a file. Each line of the file will be an environment variable. (Variables defined under the environment option override environment variables from the env\_file) For example, you might have a file called db.env that contains:

POSTGRESQL\_HOST=postgresql

Then your compose file might look like this:

version: '3.8'

services:

db:

image: postgres:latest

ports:

- "80:8080"

env\_file:

- ./db.env

##### **restart**

The restart option indicates the conditions upon which to restart a container.

The possibilities are:

* no
  + the default and indicates not to restart the container under any circumstances
* always
  + always tries to restart the container
* on-failure
  + when there is an on-failure error
* unless-stopped
  + restart unless the container has been stopped (regardless of whether it's manual or not)

For example you might want you web application to always try to restart:

version: '3.8'

services:

web:

build: .

ports:

- "80:8080"

env\_file:

- ./web.env

restart: always

### **volumes**

Volumes is both a top level option and part of the service definition.

It allows you to specify how the container might interact with some shared and persistent portion of memory.

For a particular service definition the volumes option outlines that service's named volumes or mount host paths.

Here is a set of examples from the docker docs that outline the different short syntax for the various versions of volumes/mounts.

myservice:

...

volumes:

# Just specify a path and let the Engine create a volume

- /var/lib/mysql

# Specify an absolute path mapping

- /opt/data:/var/lib/mysql

# Path on the host, relative to the Compose file

- ./cache:/tmp/cache

# User-relative path

- ~/configs:/etc/configs/:ro

# Named volume

- datavolume:/var/lib/mysql

See [this for reference](https://docs.docker.com/compose/compose-file/#volumes).

If a named volume is meant to be shared among multiple services then you will need the top level volumes element as well.

In the top level volumes you will list the named volumes and for each you can specify the driver or you can leave it empty. If its empty then it will use the default driver configured for your docker engine.

The driver of a volume provides a layer of abstraction allowing you to have your volume locally or on some remote host.

For example, you might have something similar to this, to use the default volume driver which is typically local for docker engines that have not been reconfigured:

version: '3.8'

services:

db:

...

volumes:

- mydata:/var/postgres

...

backup-db:

...

volumes:

- mydata:/example

volumes:

mydata:

The ... signify omitted portions of the file.

### **networks**

Networks were introduced with version 2 of docker compose and they enable containers to easily communicate with one another in a controlled way. (In docker compose version 1, links facilitated the communication between services and they are no longer recommended.)

By default all services specified in a docker compose file are considered part of the same network. The network is by default named after the directory in which the docker-compose file can be found. Containers on the same network are by default reachable and discoverable by one another using the name of the service and the container.

Let's take for example this file out lined in the [docker documentation.](https://docs.docker.com/compose/networking/)

version: "3"

services:

web:

build: .

ports:

- "8000:8000"

db:

image: postgres

ports:

- "8001:5432"

The web service can connect to the db service through the url- postgres://db:5432. Note the url begins with the protocol in this case postgres and then specifies the host and the port. In this case the host is db and the port is 5432 (which is the container port). For inter container communication the container's use the container ports.

If we wanted to connect to the db via the host. Our url would be postgres://{DOCKER\_IP}:8001. Notice that the port would be the host port.

The docker ip is assigned when the docker container is spun up. By default it's assigned one for each Docker network it's a part of. We can find this information through the command docker inspect --format='{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' containerID.

So that's all great, but what if you want the containers to be part of a smaller network. Maybe you only want two out of the three containers you define in your docker compose file to be able to talk to one another. Well that's where the user defined networks come into play.

You can name one of the networks or multiple networks in a particular service definition, but then you also need to name these networks using the top level option.

For example, here there is a db that connects to a backend network. An app that connects to both networks and a service called web that connects to the frontend network.

version: "3"

services:

app:

build: ./app

networks:

- backend

- frontend

web:

build: ./web

ports:

- "8000:8000"

networks:

- frontend

db:

image: postgres

ports:

- "8001:5432"

networks:

- backend

networks:

frontend:

backend:

In the top level definition, where we define the custom networks, we can specify to use specific network drivers and some other configuration options. In this case we use all the defaults.

As with volume drivers, network drivers provide core networking functionality in a way that abstracts it from the hardware itself. The default type is bridge. More details on the specifics of drivers and networking can be found [in the article on networking standalone containers.](https://docs.docker.com/network/network-tutorial-standalone/)

Now that the key components of the docker-compose file have been outlined. Let's put them all together.

## Docker-compose format

The following is the format all put together with comments detailing the use of the options.

# This is a comment and has nothing to do with the file,

# but it is helpful for me to describe the different components.

version: "version number"

# ^ The version of docker compose. For the differences between the different versions see the reference.

services:

nameofservice:

# First you name the service then you define how to build it

# Either you specify a build including context and dockerfile

# path and arguments

build:

context:

dockerfile: Dockerfilepath

# any build arguments

args:

nameofarg: value

# Then you specify additional information like the ports, network, volumes

ports:

- portnum:portnum

# networks allows you to name a network so that services can only talk to other services on the same network

networks:

- nameofnetwork

# Then you specify a name of another service to be created

volumes:

- nameofvolumeorpath:/pathincontainer

#environment variables for the particular container

env\_file:

- nameoffile.env

environment:

VARIABLE\_NAME: value

# The other format has - Variablename=value instead of VARIABLE\_NAME: value

restart: restartoption

# indicates the conditions upon which to restart a container

# either always, on-failure, unless-stopped, and "no"

nameofanotherservice:

#if you are using a service with a predefined image you can

#omit using the build instruction

image: nameofimage

# any additional information needed just like the format above

# If there is a named network earlier in the file then it will need to be specified as a network with it's driver indicated

networks:

nameofnetwork:

# If there is a named volume earlier in the file then it will need to be specified as a volume

volumes:

nameofvolume:

## Examples

The following is a simple example of a web application. While we wouldn't typically use a docker compose file to spin up such a simple application, it illustrates a basic version of using docker compose.

Save the following file as docker-compose.yaml.

version: '3'

services:

webapp:

image: 'mcr.microsoft.com/dotnet/core/samples:aspnetapp'

ports:

- '80:80'

Then from the commandline in the directory in which the file is saved.

Run the following command.

docker-compose up -d

(The -d flag indicates to do it in detached mode, so that all the standard output and error information isn't printing to the shell from which you executed the command.)

You should see the application if you go to localhost:80 in your web browser.

Then to stop the containers you can run the following command in the command line from the same directory.

docker-compose down

For an even simpler example that uses a dockerfile. Save the following docker compose file (docker-compose.yml) to the same folder as the docker file after it.

version: '3'

services:

javahelloworld:

build: .

# Define the parent image

FROM ubuntu

# Install needed programs

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

# Within the image and thus container, set the working directory to the new directory example

WORKDIR /example

# Create Hello World Java program and save it in the appropriate file

RUN echo 'public class HiWorld{ public static void main(String[] args){System.out.println("Hi world");}}'> HiWorld.java

# Compile the Java program, creating the file that the JVM can actually run

RUN javac HiWorld.java

# Run the HelloWorld program in the container

CMD ["java", "HiWorld"]

From the directory with both files run the command docker-compose up and you'll see "Hello World" output.

Finally, let take a look at a more complex example from [docker docs](https://docs.docker.com/compose/wordpress/).

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

WORDPRESS\_DB\_NAME: wordpress

volumes:

db\_data: {}

The file outlines using a volume to persist data for the database and application that connects to it. The environment variables allow you to spin up the container in such a way that you have the default credentials set up. You'll also notice that the wordpress web application depends on the database. This ensures the order that the services spin up.

Save the file into a new directory and from that directory run the command docker-compose up -d.

Then after a giving it a few seconds check out your site at <http://localhost:8000/>.

Great work!

### **Helpful Examples**

* [Docker compose example](https://docs.docker.com/compose/gettingstarted)

**DevOps Overview**

DevOps Concepts

This page covers the core of DevOps Concepts and defines the terminology and principles defined for this module.

References

* [DevOps Culture](https://martinfowler.com/bliki/DevOpsCulture.html)
* [Continuous Delivery - Atlassian](https://www.atlassian.com/continuous-delivery/pipeline)
* [CI CD CD - Atlassian](https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment)
* [Continuous Deployment vs Continuous Delivery - Atlassian](https://www.atlassian.com/continuous-delivery/continuous-deployment/how-to-get-to-continuous-deployment)
* [DevOps Image Source](https://www.logigear.com/blog/continuous-delivery-devops/continuous-delivery-everything-you-need-to-know/)

DevOps

Software Development (dev) Operations (ops) are a set of practices and methodologies designed to combine the development (production/writing of code), deployment and maintenance of code into a streamlined process. The primary goal of DevOps is to expedite the lifecycle of application development, particularly through the automation of tasks.

Steps in Dev Ops and Production of Code

The steps or phases for Dev Ops refers to the creation, testing, and deployment of an application.

These steps include:

1. Source code Control: Producing (writing) code and pushing to a repository
2. Building and Testing Automation: Test basic functionality of code (Generally unit testing) and create a new, working build
3. Deploying to Staging: Deployment of working build to a temporary environment
4. Acceptance Testing: Undergo other more complex tests (systems, integration) within temporary environment
5. Deployment of Build: Migrate working build to Production environment accessible by end users

[DevOps and Agile](https://app.revature.com/)

Agile is a mentality or philosophy utilized when approaching the creation of information systems, and is a flexible approach of addressing the steps of the Software Development Life Cycle. Development teams who practice an Agile methodology place a focus on producing code through iteration and collabortion rather than following a rigid plan.

Though DevOps, which involves the creation of a systematic approach to producing code, and Agile, which is a mentality that focuses on creating products by adapting to change quickly, seem contradictory, the goal of both methodologies is to produce working and valuable product more efficiently. DevOps pertains to the entire system working together to produce, test, deploy and maintain the code base, while Agile practices allow for each step of that process to change wherever and whenever needed.

* Agile Practices with DevOps:
  + Continuous Integration
  + Continuous Delivery
  + Continuous Deployment

Adoption of the Agile philosophies can provide a stepping stone for the establishment of a working DevOps pipeline, as Agile practices intrinsically produce more continuous feedback loops. Continuous Integration, Continuous Delivery and Continuous Deployment seek to automate the phases of DevOps as much as possible.

[Continuous Integration](https://app.revature.com/)

The process of regularly and consistently merging code into a central repository and reviewing new code to ensure that it integrates well within the previously established code base.

Tools used for this task:

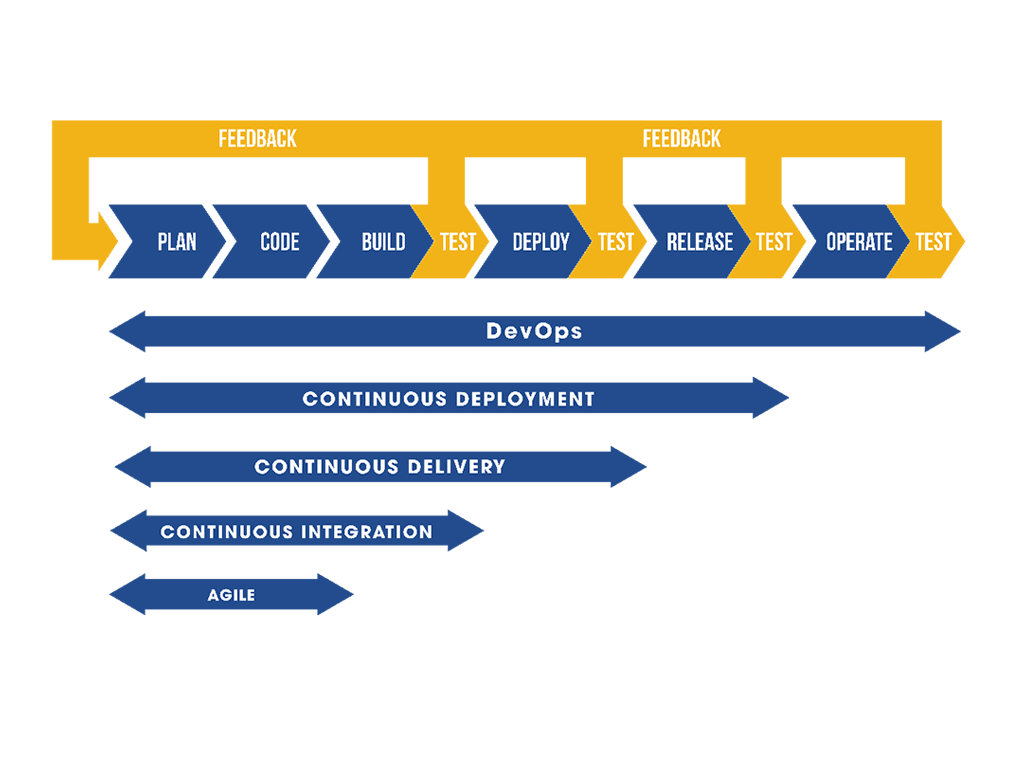
* [Github](https://github.com/features)
* [Gitlab](https://about.gitlab.com/features/)
* [Perforce](https://www.perforce.com/support/self-service-resources/documentation)
* [Bazaar](http://doc.bazaar.canonical.com/en/)
* [DevOps Tools](https://app.revature.com/)

[Continuous Delivery](https://app.revature.com/)

Development principle which focuses on the automation of the Dev Ops pipeline to the extent that human intervention is not required. Generally, this means that steps 1 - 3 (source code control, building and testing, and deployment to staging) are automated, while steps 4 (acceptance testing) may be handled by a Human, if necessary and step 5 (deployment to production environment) requires manual approval.

[Continuous Deployment](https://app.revature.com/)

Development principle which automates all phases of the Dev Ops Pipeline.



**DevOps and Agile**

# DevOps and Agile (vs Waterfall)

Agile is a mentality or philosophy utilized when approaching the creation of information systems, and is a flexible approach of addressing the steps of the Software Development Life Cycle.

### **References**

References are resources (either external or internal) that trainers and associates can use to lookup information about the technology - typically documentation, notes, videos, or tutorials

* [Agile and DevOps - Atlassian](https://www.atlassian.com/agile/devops)
* [How to Combine DevOps and Agile](https://devops.com/how-to-combine-devops-and-agile/)
* [DevOps Culture - Martin Fowler](https://martinfowler.com/bliki/DevOpsCulture.html)
* [Waterfall vs Agile - Atlassian](https://www.atlassian.com/agile/project-management/program)
* [Waterfall vs Agile - Guru99](https://www.guru99.com/waterfall-vs-agile.html)

## SDLC

The Software Development Life Cycle [SDLC] outlines the process to plan, create, test and deploy information systems and applications.

SDLC follows these General Steps:

* Requirements Phase
  + The existing system (if any) is evaluated, and determinations are made to address existing flaws or systems necessary for the new/improved functionality desired.
  + Performed by: Business Analysts.
* Analysis Phase
  + The system requirements are defined for the new system. Particularly, deficienceies and proposals to improve the system are addressed.
  + Performed by: Business Analysts with Collaboration from Senior Members of staff.
* Design Phase
  + The proposed system is designed and product features are mapped. Does not involve the production of actual code.
  + Performed by: System Architects and Senior Developement staff.
* Development Phase
  + Production of actual code to build systems.
  + Performed by: Development team
* Testing Phase
  + Software is tested against system requirements to ensure intended functionality.
  + Performed by: Development team & Testing team
* Deployment and Maintenance Phase
  + Product is deployed to customer or end users. System is maintained if/when issues arise.
  + Performed by: Operations team with possible input from Development team as needed

Practicing SDLC refers to the methodology utilized when performing these outlined steps.

## Waterfall

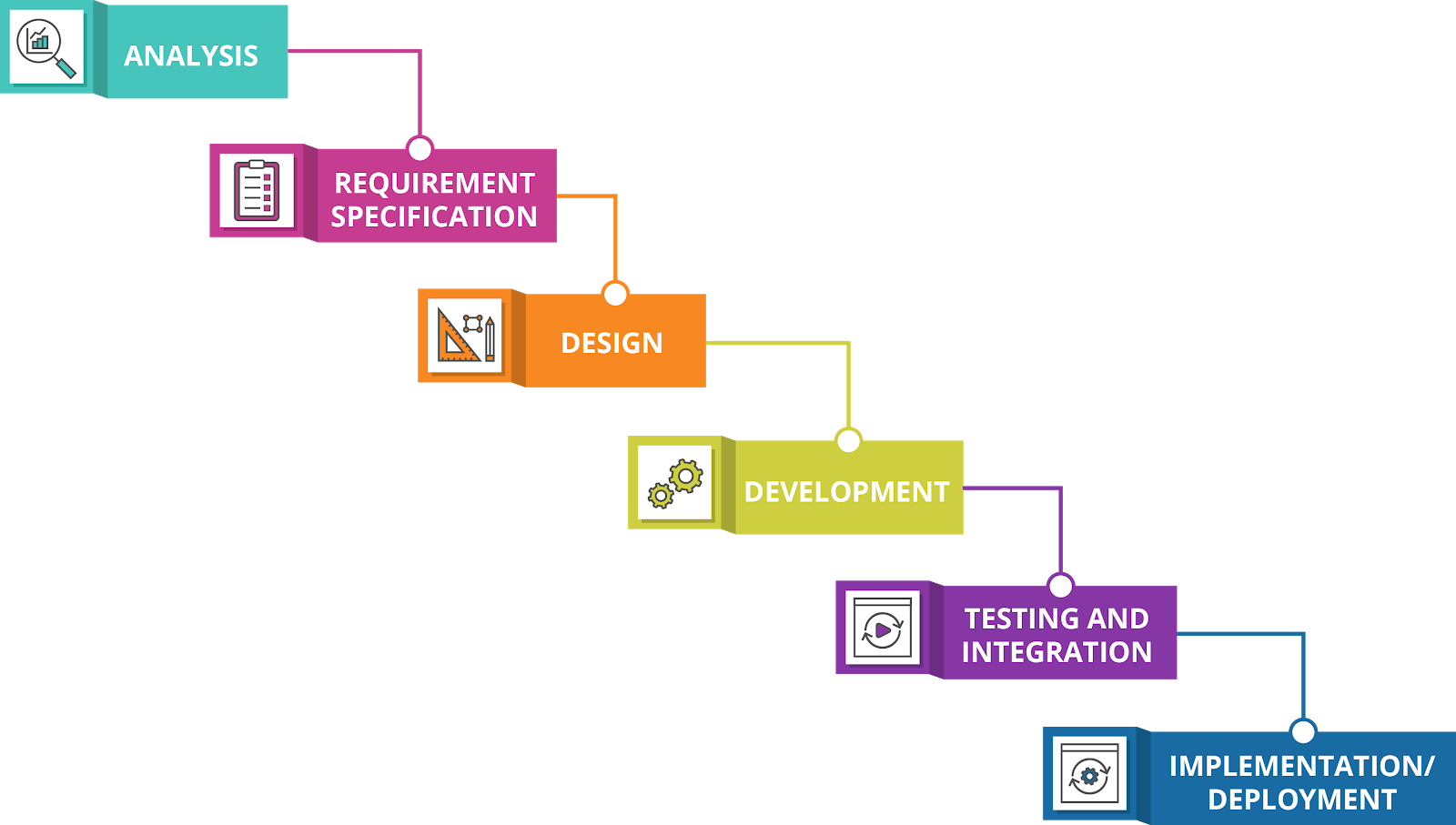
A linear approach to following the SDLC. Traditionally, a waterfall method only allows for forward movement through each phase of the SDLC. A waterfall process sees a single planning stage at the start of development, followed by the delivery and review of each SDLC phase in sequential order. Once a phase is considered completed (and approved after a review), development moves to the next stage. If requirements change or issues arise, then the progress must be halted and the product must be re-evaluated at the Requirements level.

### **Benefits to Waterfall**

* Incredibly easy to manage the workflow because there are specific deliverables and review process for each phase.
* Well suited for small teams or short-term projects that will not require any changes to the original specifications.
* Generally results in a faster delivery of product.
* Process and results can be easily documented.
* Easy to adapt to shifting teams since the steps of each phase is clearly outlined.

### **Risks/Considerations**

* Method can be quite inflexible and inefficient
* Not ideal for large teams or large projects
* Testing does not begin until after development has completed, meaning that it is more prone to bugs.



## Agile

An approach to SDLC that is based on iterative development; wherein, requirements, solutions and systems evolve throughout the production of software, and collaboration between cross-functional teams. Due to its flexibility Agile is considered the standard for development.

* More widely accepted and utilized method of following the SDLC.
* Follows Four Core Values from the [Agile Manifesto](https://agilemanifesto.org/iso/en/manifesto.html):
  + **Individuals and Interactions** over processes and tools
  + **Working Software** over comprehensive documentation
  + **Customer Collaboration** over contract negotiation
  + **Responding to change** over following a plan

Agile methods (often referred to as frameworks) are comprehensive approaches to the phases of the SDLC.

* "Scrum" is the most common Agile framework
  + Fun Fact: 'Scrum' gets its name from rugby. Short for scrummage, which is a method to restart play during a match of rugby football, involving players to pack closely together in an attempt to gain possession of the ball. Scrums require teamwork and cooperation to find the greatest success. \*Agile Scrum Methodology
  + In a "Scrum" the entire project is divided into "sprints".
  + A sprint is a smaller, isolated task or system, completed within a specified timeframe, which is required for a software development product.

### **Benefits of Agile**

* Client Collaboration is generally regarded positively
* Agile team cultures tend to stay more self-organized and motivated
* Overall quality of product is usually higher due to iterative nature
* Less risk in development process due to incremental nature of development

### **Risks/Considerations**

* Not as useful for smaller development projects
* Generally there are higher costs associated with an Agile workflow
* Development time can bloat if managed improperly or requirements are not clear during each step of the development
* Requires more experienced members during the planning and management of projects

## DevOps with Agile

As the development and deployment of applications became increasingly important, DevOps and Agile have increased in popularity as methods to optimize the process of software production. DevOps originally sought to reduce the number of manual steps necessary to produce software, resulting in more streamlined and faster releases. Agile practices encourage collaboration and re-evaluation of systems in favor of improvements. The separation of traditional DevOps and Agile approaches can lead to mismanagement of infrastructure as it can lead to a mentality of "it's someone else's problem".

Many tasks in operations can be planned for, including system upgrades, moving to new datacenters or releasing large system changes. However, many unplanned changes can arise in operations, such as system outages, performance spikes, or compromised security. Since these problems require immediate responses, and as such Agile is a useful approach to solving these operational problems, due to their unpredictable nature. DevOps seeks to eliminate this unpredictability by having a set of guidelines and a systematic approach to the integration, development, and eventual deployment of code. DevOps also seeks to eliminate the barrier between Development and Operations by integrating development systems with testing and deployment.

Though DevOps (automating structured tasks to accomplish specific goals and produce completed products), and Agile (adapting to changing workflow or new requirements rather than following formulaic plans) seem to be contrary, both methodologies uderlyingly serve the same purpose: To create working and valuable code as quickly and efficiently as possible. DevOps actually inherently adopts many Agile methodologies, as it encourages the collaboration of development and operations when creating software products.

* Agile Practices with DevOps:
  + Continuous Integration
  + Continuous Delivery
  + Continuous Deployment

Generally, there are three ways of approaching DevOps culture.

1. System Thinking: Placing emphasis on the performance of an entire system, as opposed to a specific silo of work or department.
2. Amplify Feedback Loops: Through the use of automation, the speed and efficiency of feedback loops can drastically improve.
3. Culture of Continual Experimentation and Learning: Fostering a work culture which values taking risks, learning from failure, and understanding that practice and repetition are the prerequisites to mastery, leads to increased collaboration and self-reflection. This promotes further improvement for an entire workplace community, and is even reflected through Agile retrospective meetings.

These ways are not necessarily mutually exclusive, and the integration of these approaches, with Agile philosophies can lead to:

* More streamlined processes
* Better collaboration
* Fewer bugs and faster fixes
* Higher overall quality of product

Day – 5

**Continuous integration**

# DevOps - Continuous Integration

This page covers the core of Continuous Integration, including its importance. Continuous Integration is the process of regularly and consistently merging code into a central repository and reviewing new code to ensure that it integrates well within the previously established code base.

### **References**

* [Continuous Integration - Martin Fowler](https://martinfowler.com/articles/continuousIntegration.html)
* [CI Circle Image](https://www.qfs.de/en/blog/article/2019/07/11/using-qf-test-in-continuous-integration-systems-1.html)

Below are examples version control software:

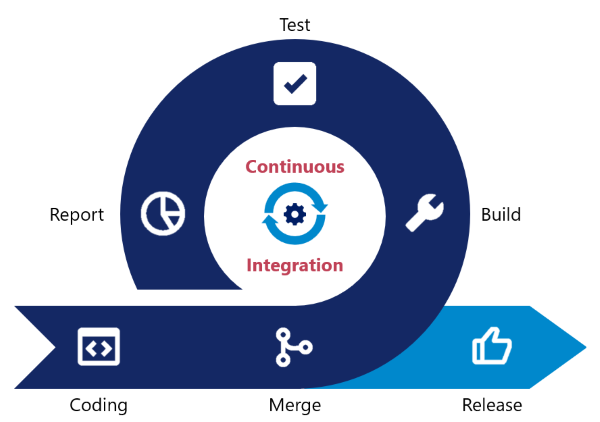
* [Github](https://github.com/features)
* [Gitlab](https://about.gitlab.com/features/)
* [Perforce](https://www.perforce.com/support/self-service-resources/documentation)
* [Bazaar](http://doc.bazaar.canonical.com/en/)
* [DevOps Tools](https://app.revature.com/)

## Continuous Integration

Continuous Integration (CI) is the first, and most fundamental step in creating an autonomous development pipeline.

Similarly to [Continuous Delivery](https://app.revature.com/) and [Continuous Deployment](https://app.revature.com/), Continuous Integration is a development team mentality, and is achieved when all members of the development team practice consistent merging of code into a central repository. For CI to take place, these Central repositories should be in the form of version control software.

Version control software is a tool which utilizes some directory structure to store files. These tools can track changes to code, and allow for changes to be merged (allowing you to select which changes to keep or reject if/when conflicts arise) or files to be rolled back to a previous version. The integration of code into these repositories should happen as often as possible with at least one commit each day. Generally, the more frequently code is merged, the less conflicts and/or integration issues will arise.



Continuous Integration establishes the foundation for an automated DevOps pipeline, because it provides the following benefits:

* Ensures the entire team works on the most up to date code
  + Frequently pushing code allows developers to account for changes performed by other team members quickly.
* Detects broken builds quickly
  + If problems arise, version control software can help detect the root cause or rollback changes when necessary.
* Code can be tested easily by creating separate, test or development branches based on the mainline code.
* Reduces risk in development when a large codebase has already been established.
* Reduces the overall amount of bugs in a project

The best way to ensure your code integrates well is to marry the integration of your code with testing the code. Running test suites on the code base after new commits helps to minimise potential disruptions if conflicts do arise, particularly when utilizing certain [DevOps tools](https://app.revature.com/) to automatically run unit and integration tests. Despite this, it is also best to practice Test Driven Development.

### [**Test Driven Development**](https://gitlab.com/revature_training/automation-testing-team)

The practice of developing code based on written test cases, as opposed to writing test cases to based on developed code, in other words, Development cycle in which test cases are first written, and then code is developed in order to allow those tests to pass.

**Continuous deployment**

# DevOps - Continuous Deployment

This page details the use and importance of Continuous Delivery for a DevOps pipeline.

### **References**

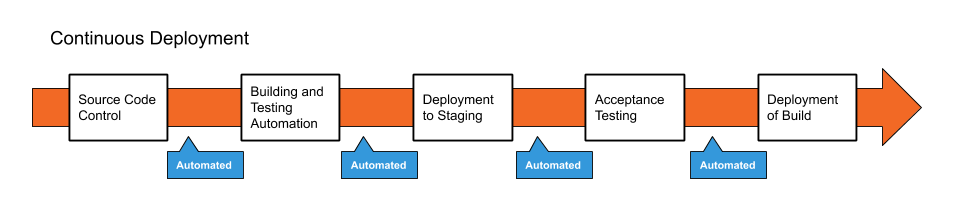
* [Continuous Deployment - Atlassian](https://www.atlassian.com/continuous-delivery/continuous-deployment)
* [How to Achieve Continuous Deployment - Atlassian](https://www.atlassian.com/continuous-delivery/continuous-deployment/how-to-get-to-continuous-deployment)
* [CI CD CD - Atlassian](https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment)

## Continuous Deployment

Continuous Deployment is a process of releasing software in which changes are tested for stability and correctness automatically. This results in immediate, autonomous deployment of code to production environments.

Continuous Deployment is often confused with Continuous Delivery due to nomenclature as both are referred to as 'CD'; however, Continuous Delivery is simply a precursor to Continuous Deployment. In Continuous Delivery there is a final, manual approval process needed before code is deployed to production environments. Continuous Deployment forgoes human intervention at every step of the deployment process, and pushes new code into the working production environment immediately so long as it meets the test requirements. When Continuous Deployment is achieved, every commited change to the code base creates and deploys a new build to the production environment.

Continuous Deployment is the ultimate goal for establishing a true DevOps pipeline, as it ensures that all steps for the creation of product, including code creation, testing, building, and deployment are automated and work seamlessly together.



As the major difference for Continuous Deployment and Delivery resides in the manual approval of deploying code to production, many benefits ([feedback speed, code quality and efficiency](https://app.revature.com/)) are retianed with the use of Continuous Deployment. However, there are some additional considerations for Continuous Deployment:

### **Costs/Risks of Continuous Deployment**

* Establishing a Continuous Deployment pipeline requires a more substantial investment in the engineering, and the testing culture.
* Documentation of processes is required to communicate to development, production and testing teams.
* Ongoing maintenance of deployment pipeline is required to ensure work continues running smoothly, increasing production costs.
* Feature flags (communication of completed features and progress) are required for coordination between departments.

### **Benefits of Continuous Deployment**

* Even faster development process, without the need to pause for deployment.
* New releases are less risky, as small changes can be easily recognized and fixed, allowing for better and quicker feedback.
* Increased communication and regular streams of improvements are generally regarded highly by customers.

**Continuous delivery**

# DevOps - Continuous Delivery

This page details the use and importance of Continuous Delivery for a DevOps pipeline.

### **References**

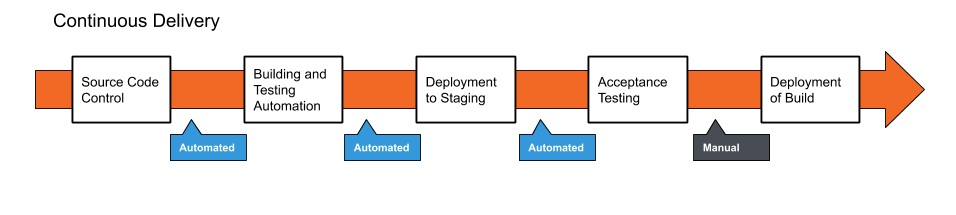
* [Continuous Delivery - Martin Fowler](https://martinfowler.com/bliki/ContinuousDelivery.html)
* [Continuous Delivery - Atlassian](https://www.atlassian.com/continuous-delivery/pipeline)
* [CI CD CD - Atlassian](https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment)

## Continuous Delivery

Continuous Delivery is a paradigm in which the building, management and testing of produced software is automated such that deployments can be performed at the push of a button.

Continuous delivery is often confused with Continuous Deployment, which automates the entire production pipeline, including deployment. Continuous Delivery; however is the process of automating all steps of a Development pipeline except for the final deployment step. Inherently, Continuous Delivery is dependent on the implementation of Continuous Integration, and also serves as a stepping stone to creating a fully automated Development Pipeline (Continuous Deployment). Though Continuous Integration can technically be achieved without automation, Continuous Delivery is only achieved when code integration, testing and product building has been automated. In this way, you are able to perform frequent deployments "at the press of a button", but may choose not to do so, usually for business purposes or possibly due to a preference for a regular scheduled deployment process.

The deployment to production may also be kept manual so that final user acceptance tests can be performed manually as a final safety check on the code to ensure that it meets business needs. This is due to the difficulty and cost of creating tests to evaluate the user experience and not simply the functionality.



Benefits of Continuous Delivery:

* Reduced Risk in Deployment
  + Since new builds are not deployed automatically, faulty code is less likely to by deployed to production environments.
  + Less pressure is placed on the development team by allowing for small, more frequent changes to be made. This expedites the iteration of code.
* Predictable Progress
  + Since pipelines are a programmable infrastructure by the development team, desired behavior during the production of code is easier to configure.
  + With a pipeline the deployment process is more predictable, allowing development team to focus on the production of code rather than operational steps required to deploy the new codebase.
* Frequent Feedback
  + With the increased efficiency of producing code, smaller, more incremental changes can be applied to a system more frequently. If human error does cause problems, it is easy to roll back changes to a working build.
  + More releases accelerates the communication and feedback loop with client/product owners as well.

### **Costs/Considerations of Continuous Delivery**

* Requires a strong foundtation with Continuous Integration culture, and test suite coverage
* The Final deployment must still be automated which is an additional cost, Though the trigger to begin the process is manual this can still cause slowdown for the development.
* Communication of incomplete features and backlog must be maintained rigorously to communicate expectations to client and development team

**Jenkins**

# DevOps Tools: Jenkins

Jenkins is a self-contained, open source automation server, which can be used to automate the building, testing and deployment of software. Jenkins can be installed standalone, through native system packages, or using Docker.

## Projects/Jobs and Builds

In Jenkins, you work with projects or jobs. Each job is a repeatable set of steps that automate a task, such as building, testing, and deploying your software. In Jenkins, it is possible for a job to be triggered manually, externally (via a REST endpoint or a push to a repository), or even by another job. When a job is triggered, Jenkins creates a build of the project.

Jobs have a status called Health, represented by a weather condition, and builds have a more traditional status. Each build has a colored circle that represents the success or failure of a build. The health of a job is a status that is dependent on the status of the most recent builds.

### **Build Statuses**

Blue - Success Yellow - Unstable (usually indicates failed tests) Red - Failure Grey - No builds or aborted build

## Weather

Sunny - More than 80% of runs passing Partially Sunny - 61% to 80% passing Cloudy - 41% to 60% passing Raining - 21% to 40% passing Stormy - less than 21% passing

### **References**

* [Jenkins Documentation](https://www.jenkins.io/doc/)
* [Installation of Jenkins](https://www.jenkins.io/doc/book/installing/)
* [Jenkins Pipeline Tutorial](https://www.youtube.com/watch?v=s73nhwYBtzE)
* [Using Jenkins](https://www.jenkins.io/doc/book/using/)
* [Managing Jenkins](https://www.jenkins.io/doc/book/managing/)
* [Defined Pipeline-compatible steps](https://www.jenkins.io/doc/pipeline/steps/)

## Jenkins Installation

To utilize Jenkins, it is recommended to [install Docker](https://docs.docker.com/get-docker/) on your operating system. Moreover, you can also utilize an [AWS Linux Ubuntu EC2](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html) server.

* [Jenkins Install - MacOS](https://www.jenkins.io/download/lts/macos/)
* [Jenkins Install - Linux](https://www.jenkins.io/doc/book/installing/#linux)
* [Jenkins Install - Windows](http://mirrors.jenkins.io/windows/)

## Using Jenkins

After you [download Jenkins](https://www.jenkins.io/doc/pipeline/tour/getting-started/), open a terminal a Run:

java -jar jenkins.war --httpPort=8080.

Next, navigate to http://localhost:8080 and follow the instructions to complete the installation process.

## Pipeline Tutorial

A Jenkins Pipeline is a suite of plugins which support the implementation of a Continuous Delivery Pipeline in Jenkins. The follow details steps for utilizing a Jenkins pipeline with Java.

You can follow these steps to establish your pipeline [here](https://gitlab.com/revature_training/devops-team/-/tree/master/trainer-examples/matthew-oberlies/pipeline-demo/scripts). Alternatively you can follow these [steps to build a Java app with Maven](https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/).

Creating a custom pipeline can offer you a great deal of flexibility. To create a custom pipeline, you will need to create a jenkinsfile:

1. Configure your jenkinsfile (see below)
2. Navigate to your Jenkins server and select 'New Item' from the Jenkins menu.
3. Provide a name for your new item (i.e. "Pipeline-Example")
4. Click "Add Source" to choose the type and details for your repository
5. Click "Save". You can now build your project using this new pipeline

A jenkinsfile consists of agents, stages, and steps.

An [agent](https://www.jenkins.io/doc/book/pipeline/syntax/#agent) informs Jenkins where, and how to execute the Pipeline. For example:

* From a docker image

pipeline {

agent {

docker { image 'maven:3.3.3' }

}

// additional information to follow //

}

* From anywhere

pipeline {

agent any

// additional information to follow //

}

An agent is required at the top level of a pipeline block but can also optionally be used at the stage level to define stage-level usage.

[Stages](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) are a sequence of one or more individual custom [stage](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) directives. Each stage will consist of multiple steps, that are performed by a Jenkins pipeline.

Stages Block

* The stages block is required to list each individual stage directive
* Each pipeline should only contain a single stages block.
* This stages block must contain as least one stage directive
* May contain multiple stage directives.

Stage Directive

* Each stage directive requires a string parameter for the name of the stage.
* Each stage directive requires at least one step
* May contain multiple steps

For example:

pipeline {

agent any

stages {

stage('Deploy') {

// List of steps go here //

}

}

}

[Steps](https://www.jenkins.io/doc/book/pipeline/syntax/#steps) are the specific actions taken by a Jenkins Pipeline.

Steps Block

* Do not require parameters
* May contain a script block to define behavior

For example:

pipeline {

agent any

stages {

stage('Deploy') {

steps {

echo 'Hello World'

script {

def browsers = ['chrome', 'firefox']

for (int i = 0; i < browsers.size(); i++) {

echo "Testing the ${browsers[i]} browser"

}

}

}

}

}

}

### **Disadvantages of Jenkins**

* Requires custom jenkinsfile syntax for pipeline configuration

### **Advantages of Jenkins**

* Completely Open Source
* Free
* Supports all Operating systems
* Supports all source code repositories

**Jenkins job/project**

# DevOps Tools: Jenkins

Jenkins is a self-contained, open source automation server, which can be used to automate the building, testing and deployment of software. Jenkins can be installed standalone, through native system packages, or using Docker.

## Projects/Jobs and Builds

In Jenkins, you work with projects or jobs. Each job is a repeatable set of steps that automate a task, such as building, testing, and deploying your software. In Jenkins, it is possible for a job to be triggered manually, externally (via a REST endpoint or a push to a repository), or even by another job. When a job is triggered, Jenkins creates a build of the project.

Jobs have a status called Health, represented by a weather condition, and builds have a more traditional status. Each build has a colored circle that represents the success or failure of a build. The health of a job is a status that is dependent on the status of the most recent builds.

### **Build Statuses**

Blue - Success Yellow - Unstable (usually indicates failed tests) Red - Failure Grey - No builds or aborted build

## Weather

Sunny - More than 80% of runs passing Partially Sunny - 61% to 80% passing Cloudy - 41% to 60% passing Raining - 21% to 40% passing Stormy - less than 21% passing

### **References**

* [Jenkins Documentation](https://www.jenkins.io/doc/)
* [Installation of Jenkins](https://www.jenkins.io/doc/book/installing/)
* [Jenkins Pipeline Tutorial](https://www.youtube.com/watch?v=s73nhwYBtzE)
* [Using Jenkins](https://www.jenkins.io/doc/book/using/)
* [Managing Jenkins](https://www.jenkins.io/doc/book/managing/)
* [Defined Pipeline-compatible steps](https://www.jenkins.io/doc/pipeline/steps/)

## Jenkins Installation

To utilize Jenkins, it is recommended to [install Docker](https://docs.docker.com/get-docker/) on your operating system. Moreover, you can also utilize an [AWS Linux Ubuntu EC2](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html) server.

* [Jenkins Install - MacOS](https://www.jenkins.io/download/lts/macos/)
* [Jenkins Install - Linux](https://www.jenkins.io/doc/book/installing/#linux)
* [Jenkins Install - Windows](http://mirrors.jenkins.io/windows/)

## Using Jenkins

After you [download Jenkins](https://www.jenkins.io/doc/pipeline/tour/getting-started/), open a terminal a Run:

java -jar jenkins.war --httpPort=8080.

Next, navigate to http://localhost:8080 and follow the instructions to complete the installation process.

## Pipeline Tutorial

A Jenkins Pipeline is a suite of plugins which support the implementation of a Continuous Delivery Pipeline in Jenkins. The follow details steps for utilizing a Jenkins pipeline with Java.

You can follow these steps to establish your pipeline [here](https://gitlab.com/revature_training/devops-team/-/tree/master/trainer-examples/matthew-oberlies/pipeline-demo/scripts). Alternatively you can follow these [steps to build a Java app with Maven](https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/).

Creating a custom pipeline can offer you a great deal of flexibility. To create a custom pipeline, you will need to create a jenkinsfile:

1. Configure your jenkinsfile (see below)
2. Navigate to your Jenkins server and select 'New Item' from the Jenkins menu.
3. Provide a name for your new item (i.e. "Pipeline-Example")
4. Click "Add Source" to choose the type and details for your repository
5. Click "Save". You can now build your project using this new pipeline

A jenkinsfile consists of agents, stages, and steps.

An [agent](https://www.jenkins.io/doc/book/pipeline/syntax/#agent) informs Jenkins where, and how to execute the Pipeline. For example:

* From a docker image

pipeline {

agent {

docker { image 'maven:3.3.3' }

}

// additional information to follow //

}

* From anywhere

pipeline {

agent any

// additional information to follow //

}

An agent is required at the top level of a pipeline block but can also optionally be used at the stage level to define stage-level usage.

[Stages](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) are a sequence of one or more individual custom [stage](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) directives. Each stage will consist of multiple steps, that are performed by a Jenkins pipeline.

Stages Block

* The stages block is required to list each individual stage directive
* Each pipeline should only contain a single stages block.
* This stages block must contain as least one stage directive
* May contain multiple stage directives.

Stage Directive

* Each stage directive requires a string parameter for the name of the stage.
* Each stage directive requires at least one step
* May contain multiple steps

For example:

pipeline {

agent any

stages {

stage('Deploy') {

// List of steps go here //

}

}

}

[Steps](https://www.jenkins.io/doc/book/pipeline/syntax/#steps) are the specific actions taken by a Jenkins Pipeline.

Steps Block

* Do not require parameters
* May contain a script block to define behavior

For example:

pipeline {

agent any

stages {

stage('Deploy') {

steps {

echo 'Hello World'

script {

def browsers = ['chrome', 'firefox']

for (int i = 0; i < browsers.size(); i++) {

echo "Testing the ${browsers[i]} browser"

}

}

}

}

}

}

### **Disadvantages of Jenkins**

* Requires custom jenkinsfile syntax for pipeline configuration

### **Advantages of Jenkins**

* Completely Open Source
* Free
* Supports all Operating systems
* Supports all source code repositories

**Jenkins build**

# DevOps Tools: Jenkins

Jenkins is a self-contained, open source automation server, which can be used to automate the building, testing and deployment of software. Jenkins can be installed standalone, through native system packages, or using Docker.

## Projects/Jobs and Builds

In Jenkins, you work with projects or jobs. Each job is a repeatable set of steps that automate a task, such as building, testing, and deploying your software. In Jenkins, it is possible for a job to be triggered manually, externally (via a REST endpoint or a push to a repository), or even by another job. When a job is triggered, Jenkins creates a build of the project.

Jobs have a status called Health, represented by a weather condition, and builds have a more traditional status. Each build has a colored circle that represents the success or failure of a build. The health of a job is a status that is dependent on the status of the most recent builds.

### **Build Statuses**

Blue - Success Yellow - Unstable (usually indicates failed tests) Red - Failure Grey - No builds or aborted build

## Weather

Sunny - More than 80% of runs passing Partially Sunny - 61% to 80% passing Cloudy - 41% to 60% passing Raining - 21% to 40% passing Stormy - less than 21% passing

### **References**

* [Jenkins Documentation](https://www.jenkins.io/doc/)
* [Installation of Jenkins](https://www.jenkins.io/doc/book/installing/)
* [Jenkins Pipeline Tutorial](https://www.youtube.com/watch?v=s73nhwYBtzE)
* [Using Jenkins](https://www.jenkins.io/doc/book/using/)
* [Managing Jenkins](https://www.jenkins.io/doc/book/managing/)
* [Defined Pipeline-compatible steps](https://www.jenkins.io/doc/pipeline/steps/)

## Jenkins Installation

To utilize Jenkins, it is recommended to [install Docker](https://docs.docker.com/get-docker/) on your operating system. Moreover, you can also utilize an [AWS Linux Ubuntu EC2](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html) server.

* [Jenkins Install - MacOS](https://www.jenkins.io/download/lts/macos/)
* [Jenkins Install - Linux](https://www.jenkins.io/doc/book/installing/#linux)
* [Jenkins Install - Windows](http://mirrors.jenkins.io/windows/)

## Using Jenkins

After you [download Jenkins](https://www.jenkins.io/doc/pipeline/tour/getting-started/), open a terminal a Run:

java -jar jenkins.war --httpPort=8080.

Next, navigate to http://localhost:8080 and follow the instructions to complete the installation process.

## Pipeline Tutorial

A Jenkins Pipeline is a suite of plugins which support the implementation of a Continuous Delivery Pipeline in Jenkins. The follow details steps for utilizing a Jenkins pipeline with Java.

You can follow these steps to establish your pipeline [here](https://gitlab.com/revature_training/devops-team/-/tree/master/trainer-examples/matthew-oberlies/pipeline-demo/scripts). Alternatively you can follow these [steps to build a Java app with Maven](https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/).

Creating a custom pipeline can offer you a great deal of flexibility. To create a custom pipeline, you will need to create a jenkinsfile:

1. Configure your jenkinsfile (see below)
2. Navigate to your Jenkins server and select 'New Item' from the Jenkins menu.
3. Provide a name for your new item (i.e. "Pipeline-Example")
4. Click "Add Source" to choose the type and details for your repository
5. Click "Save". You can now build your project using this new pipeline

A jenkinsfile consists of agents, stages, and steps.

An [agent](https://www.jenkins.io/doc/book/pipeline/syntax/#agent) informs Jenkins where, and how to execute the Pipeline. For example:

* From a docker image

pipeline {

agent {

docker { image 'maven:3.3.3' }

}

// additional information to follow //

}

* From anywhere

pipeline {

agent any

// additional information to follow //

}

An agent is required at the top level of a pipeline block but can also optionally be used at the stage level to define stage-level usage.

[Stages](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) are a sequence of one or more individual custom [stage](https://www.jenkins.io/doc/book/pipeline/syntax/#stages) directives. Each stage will consist of multiple steps, that are performed by a Jenkins pipeline.

Stages Block

* The stages block is required to list each individual stage directive
* Each pipeline should only contain a single stages block.
* This stages block must contain as least one stage directive
* May contain multiple stage directives.

Stage Directive

* Each stage directive requires a string parameter for the name of the stage.
* Each stage directive requires at least one step
* May contain multiple steps

For example:

pipeline {

agent any

stages {

stage('Deploy') {

// List of steps go here //

}

}

}

[Steps](https://www.jenkins.io/doc/book/pipeline/syntax/#steps) are the specific actions taken by a Jenkins Pipeline.

Steps Block

* Do not require parameters
* May contain a script block to define behavior

For example:

pipeline {

agent any

stages {

stage('Deploy') {

steps {

echo 'Hello World'

script {

def browsers = ['chrome', 'firefox']

for (int i = 0; i < browsers.size(); i++) {

echo "Testing the ${browsers[i]} browser"

}

}

}

}

}

}

### **Disadvantages of Jenkins**

* Requires custom jenkinsfile syntax for pipeline configuration

### **Advantages of Jenkins**

* Completely Open Source
* Free
* Supports all Operating systems
* Supports all source code repositories

**Maven Review**

[Maven](https://maven.apache.org/) Review

Maven is a dependency manager and build automation tool for java projects.

Project Object Model (POM):

* Handles project configuration and dependencies which are defined in the **pom.xml** file. Here is an example pom.xml file.

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.revature.app</groupId>

<artifactId>my-app</artifactId>

<version>1</version>

<dependencies>

<dependency>

<groupId>org.apache.maven</groupId>

<artifactId>maven-artifact</artifactId>

<version>${mavenVersion}</version>

</dependency>

<dependency>

<groupId>org.apache.maven</groupId>

<artifactId>maven-core</artifactId>

<version>${mavenVersion}</version>

</dependency>

</dependencies>

</project>

Maven Lifecycle:

* When Maven builds your project, it goes through several steps called phases. The default maven lifecycle is:

1. Validate => project is correct and all necessary information is available
2. Compile => compiles project source code
3. Test => tests all compiled code
4. Package => packages all compiled code to WAR/JAR file
5. Integration => performs all integration tests on WAR/JAR
6. Verify => runs checks on the results of integration tests
7. Install => installs WAR/JAR to local repository
8. Deploy => copies final WAR/JAR to the remote repositor

**SonarCloud & SonarLint**

# DevOps Tools - Sonar

Sonar cloud, Sonar Qube and Sonar lint are code quality analysis tools which increase the readability, security and matainability of code. Utilizing code quality analysis tools can help development teams produce higher quality code, and provide standardization between developers, which can play a vital role in the integration of code.

### **References**

* [Sonar Cloud - Documentation](https://sonarcloud.io/documentation)
* [Sonar Qube - Documentation](https://docs.sonarqube.org/latest/)
* [Sonar Lint - Github](https://github.com/SonarSource/sonarlint-eclipse)

## Code Quality Analysis and Code Smells

Code Quality Analysis tools are programs specifically designed to expose errors as well as 'code smells'.

Code Smells

* Vulnerabilities
  + Data security issues
* Bugs
  + Issues with functionality of code
* Maintainability issues
  + Confusing or hard to maintain code
  + Repeated instances of code
  + Unused imports
  + Empty code blocks
  + Unaddressed automated code comments

## Sonar Cloud

A cloud based code review solution which can be configured to review code within a cloud repositoy, such as Github.

### **Sonar Cloud with Github Installation**

* Navigate to the [Sonar Cloud Login Page](https://sonarcloud.io/sessions/new), and select "Log in with Github"
* Click on "Analyze your code" and follow the steps for project setup
* Select the repository to analyze.

You may Automatically analyze your code with Sonar Cloud, or by configuring another Continuous Integration Tool.

* [Sonar Cloud with Travis CI](https://sonarcloud.io/documentation/integrations/ci/travis-ci/)
* [Sonar Cloud with Circle CI](https://sonarcloud.io/documentation/integrations/ci/circleci/)

## Sonar Qube

Code review tool built to work on a centralized server or integrated into a development pipeline. Due to the increased flexibility and configuration options of Sonar Qube, it can be seen as a more powerful tool.

### **Sona Qube Installation**

* [Quickstart Guide](https://docs.sonarqube.org/latest/setup/get-started-2-minutes/)
* [Production Environment Guide](https://docs.sonarqube.org/latest/setup/install-server/)

## Sonar Lint

Sonar Lint is a free, open-source linting tool. A linting tool/linter is a software tool which, when integrated with an IDE, can provide increased feedback to the developer. Traditional IDE tools utilize built-in linters to identify code syntax errors, and exceptions. Sonar Lint provides further suggestions by recognizing code smells.

### [**Sonar Lint - Ecplise**](https://marketplace.eclipse.org/content/sonarlint)

* Open Eclipse
* Select "Ecplise Marketplace..." from the Help Menu
* Search for "Sonar Lint" in the searchbox
* Select Install and accept the User Agreement
* Restart Ecplise

### [**Sonar Lint - Visual Studio Code**](https://marketplace.visualstudio.com/items?itemName=SonarSource.sonarlint-vscode)

* Open Visual Studio Code
* Select the "Extensions" Tab from the project explorer
* Search for "Sonar Lint" and Select Install