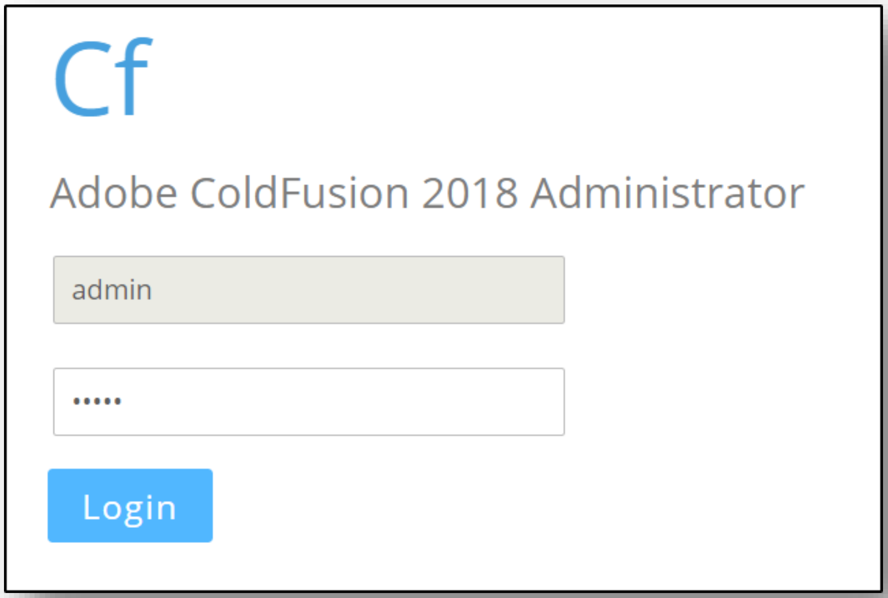
Adobe Systems

ColdFusion (2018 release) – Public Beta Help

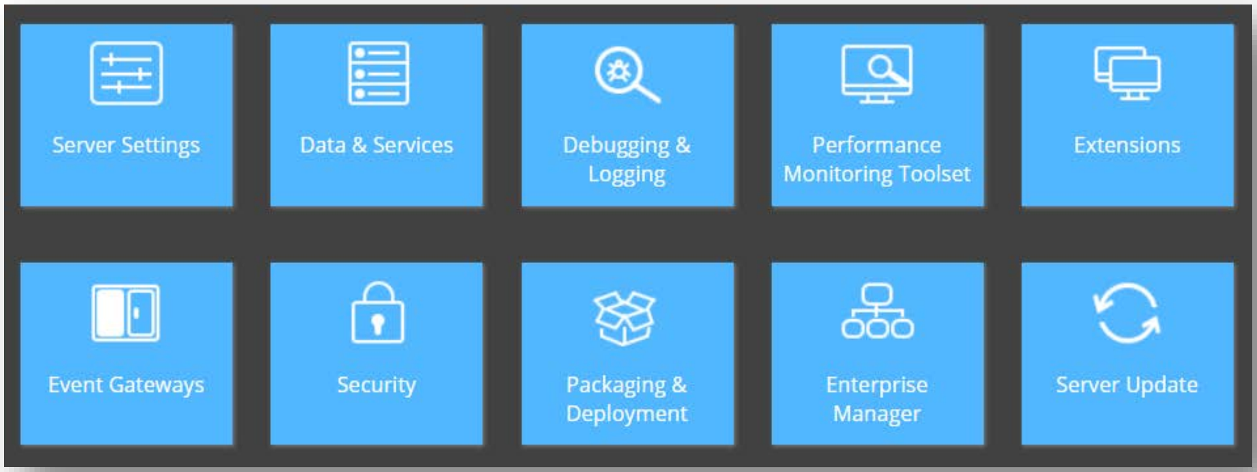
Documentation

ColdFusion Administrator has a new User Interface

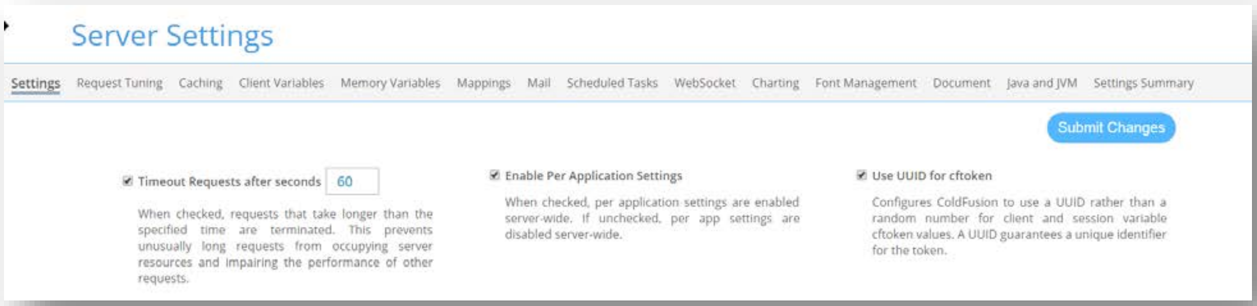
There is a new login page.



When you sign in successfully, you see a refurbished User Interface.



When you click any category, you can see all the options organized in the form of tabs.



Server Settings

Settings | Request Tuning | Caching | Client Variables | Memory Variables | Mappings | Mail | Scheduled Tasks | WebSocket | Charting | Font Management | Document | Java and JVM | Settings Summary

Submit Changes

☑ Timeout Requests after seconds 60

When checked, requests that take longer than the specified time are terminated. This prevents unusually long requests from occupying server resources and impairing the performance of other requests.

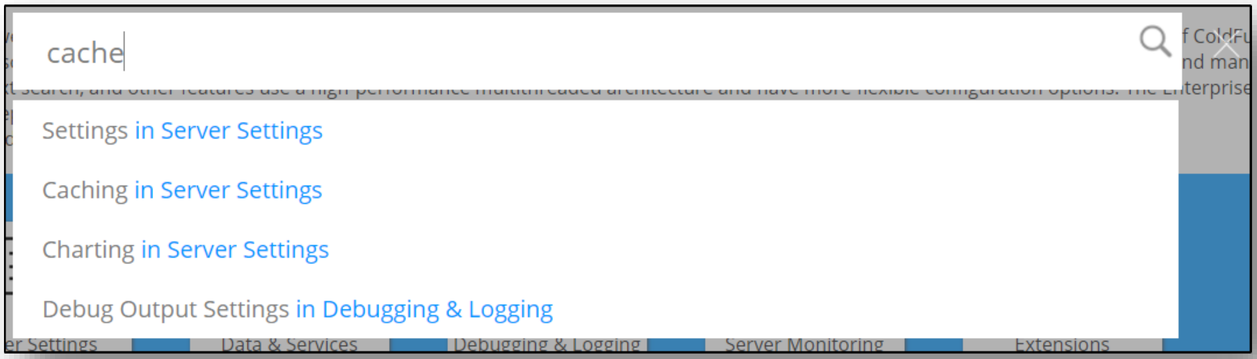
☑ Enable Per Application Settings

When checked, per application settings are enabled server-wide. If unchecked, per app settings are disabled server-wide.

☑ Use UUID for cftoken

Configures ColdFusion to use a UUID rather than a random number for client and session variable cftoken values. A UUID guarantees a unique identifier for the token.

In the UI, we have also enhanced the search experience. Click in the top-right corner of the screen. Enter your search string. You can see recommendations based on the search string. Choose the required option from the search list.



cache

Settings in Server Settings

Caching in Server Settings

Charting in Server Settings

Debug Output Settings in Debugging & Logging

Performance Monitoring Toolset

In the 2018 release of ColdFusion, we have introduced a comprehensive application monitoring solution called Performance Monitoring Toolset.

The 2018 release of Adobe ColdFusion Enterprise and Standard editions now offer you the Performance Monitoring Toolset, an all-new solution that ensure that your applications are as optimized as the high-performing ColdFusion engine.

Performance Monitoring Toolset is an application monitoring suite to enable developers gain insight on how applications perform at run time.

Performance Monitoring Toolset:

- Captures and persists data in real-time for business transactions and helps resolves issues effectively
- Provides Dynamic Server Index, which includes metrics for JVM, GC, CPU usage, and so on
- Analyzes thread dumps and helps users in identifying bottlenecks
- Supports clustered environments

PMT also:

- Continuously monitors applications, servers and databases to ensure increased stability.
- Identifies and sends out alerts before they impact critical processes
- Safeguards applications from common performance pitfalls.

For more information, see the documentation of Performance Monitoring Toolset.

Language enhancements

NULL support

CFML did not support NULL values. ColdFusion converts a null value to an empty string. In operations such as SerializeJSON and CFquery, Null values are automatically converted into empty strings, a non-null value.

While the code below runs as expected:

```
<cfscript>
      myvar=""; // empty string
      writeOutput(isNULL(myvar)); // returns NO
</cfscript>
```

The code below throws an exception:

```
<cfscript>
      myvar=NULL; // value is NULL
      writeOutput(isNULL(myvar)); // returns Variable NULL is undefined
</cfscript>
```

Although, you could always assign NULL to a variable through JavaCast function. For example,

```
<cfscript>
      myvar=JavaCast("null",0);
      writeOutput(isNULL(myvar)); // returns TRUE
</cfscript>
```

ColdFusion (2018 release) now supports Null values. Null support is turned off by default. To enable Null support, follow the following steps:

1. Enable NULL support in the ColdFusion Administrator for all global CF applications. In **Server Settings** > **Settings**, select the check-box **Enable Null Support**. After the change, the code below works as expected:

   ```
   <cfscript>
         myvar=NULL;
         writeOutput(isNull(myvar)); // returns TRUE
   </cfscript>
   ```

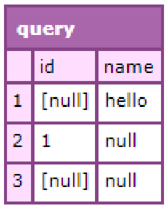
2. You can enable Null support at the application level. There is a new variable, ***enableNULLSupport***, which you can use to add NULL support in your project. For example, the code below adds NULL support in **Application.cfc**:

   ```
   component{
         this.enableNULLSupport=true;
   }
   ```

Null strings are different from null values. When you dump a null value using either writeDump or <cfdump>, the null value appears as **[null].** For example, For example, with the enableNULLSupport flag set to true, create a query as shown below, with null values and null strings.

```
<cfscript>
       myquery=querynew("id,name","integer,varchar",[[null,"hello"],[1,"null"],[null,
"null"]]);
       writedump(myquery);
       writedump(format="text",var=#myquery#);
</cfscript>
```

When you dump the values, you can see that null values are represented as [null].



query		
	id	name
1	[null]	hello
2	1	null
3	[null]	null

```
[Record # 1]
id: [null]
name: hello

[Record # 2]
id: 1
name: null

[Record # 3]
id: [null]
name: null
```

Note: With Null support enabled, isNull returns a YES / NO when variables are passed, and throws an exception if a non-existent variable name is passed.

Closures in ColdFusion tags

In previous versions of ColdFusion, you could use closures in script syntax, but not in tags. Supporting closures in Tags would mean enabling everything that is valid within a CFScript body to be embedded inside a CFTag.

For example, the closure function in arraySort returns results as expected.

```
<cfscript>
       // Define an array of structs
       myArray = [
              {name="Thomas", age="22"},
              {name="Zaza", age="36"},
              {name="Novak", age="136"},
              {name="Marin", age="361"},
              {name="Rafa", age="03"},
              {name="$bl0091@", age="-23"}
       ];

       // Define a closure function that sorts the names in the array of structs
       callback=function (e1, e2){
              return compare(e1.name, e2.name);
       }

       // Use the closure function
       arraySort(myArray,callback);

       // Display the sorted array of structs
       WriteDump(myArray);
</cfscript>
```

On the other hand, the snippet below always returns an exception:

```
<cfset myClosure= function() {…}>
```

In this release of ColdFusion, you can use closure functions in tags. For example,

```
<cfset myarray=[
    {name="Thomas", age="22"},
    {name="Zaza", age="36"},
    {name="Novak", age="136"},
    {name="Marin", age="361"},
    {name="Rafa", age="3"},
    {name="$bl0091@", age="-23"}
]>
<!--- define closure function --->
<cfset closure=function (e1,e2){
       return compare(e1.name,e2.name);
}>
<cfset ar = arraySort(myarray,closure)>
<cfdump var="#myarray#">
```

Asynchronous programming

In this release of ColdFusion, we have added support for asynchronous programming via Future. A Future is an eventual result of an asynchronous operation.

Asynchronous programming is useful when you want to reduce the average response time of an application. You can use asynchronous programming to offload IO, database intensive tasks. Also use asynchronous programming to improve the responsiveness of a UI.

Some of the benefits of asynchronous programming are:

- Near real time processing
- Easy to distribute tasks
- Uses its own worker threads
- Dynamically configurable thread pool (Admin Console)
- On-demand thread pool creation

Example

When salary is credited to our bank accounts, we pay various bills like credit cards, mortgage, utilities, and so on. The payments are dependent on salary or in this context, chained to salary.

```
<cfscript>

      getAccountBalance = function(){
            var balance = 120000;
            return balance;
      }

      function payCreditCardBill(accountBalance){
            var ccBill = 1890;
            return accountBalance-ccBill;
      }

      payEMIs = function(accountBalance){
            var mortgageEMI = 1000;
            var carLeaseEMI = 750;
            var healthInsuranceEMI = 250;

            return accountBalance-(mortgageEMI+carLeaseEMI+healthInsuranceEMI);
      }

      miscellenousExpenses = function(accountBalance){
            var shopping = 1500;
            var clubExpense  =1000;
            var casinoExpense = 2000;
            return accountBalance-(shopping+clubExpense+casinoExpense);
      }

      checkBalance = function(accountBalance){
            while(accountBalance > 5000){
                        accountBalance = miscellenousExpenses(accountBalance);
                        writeOutput("checkBalance = " & accountBalance & "<br/>");
            }
            if(accountBalance < 5000)
                        throw (message="Account balance below threshold!!!",
type="info");
        }

        errorHandler = function(error){
            if(error.message contains "Account balance below threshold!"){
                        return "You have reached your spending limit!";
            }
        }

        future = runAsync(getAccountBalance).then(payCreditCardBill).then(payEMIs).
        then(miscellenousExpenses).then(checkBalance).error(errorHandler);

        writeOutput(future.get());

</cfscript>
```

You can also use closures with runAsync function.

For example,

```
<cfscript>

            future = runAsync(function(){return "I am invoked from
RunAsync directly!";});

</cfscript>
```

Methods available with runAsync are:

- cancel();
- error(callback, timeout);
- error(callback);
- get();
- get(timeout);
- isCancelled();
- isDone();
- then(callback);
- then(callback, timeout);

Note:

- UDF Method would be either a closure reference, closure, or a User Defined Method

For example,

```
<cfscript>
        function add(){
            return 10+20;
        }
        Future = runAsync(add);
        writeDump(Future.get());
</cfscript>
```

Here add() is a User Defined Method and has to be passed to runAsync as “add” and not “add()”.

Empty future

An empty future is an object, which can be explicitly marked as complete with a result value. It can be used in producer/consumer scenarios.

For example,

```
<cfscript>
        p = runAsync(); // empty future
        p.complete(10);
        writeOutput(p.get()); // displays 10
</cfscript>
```

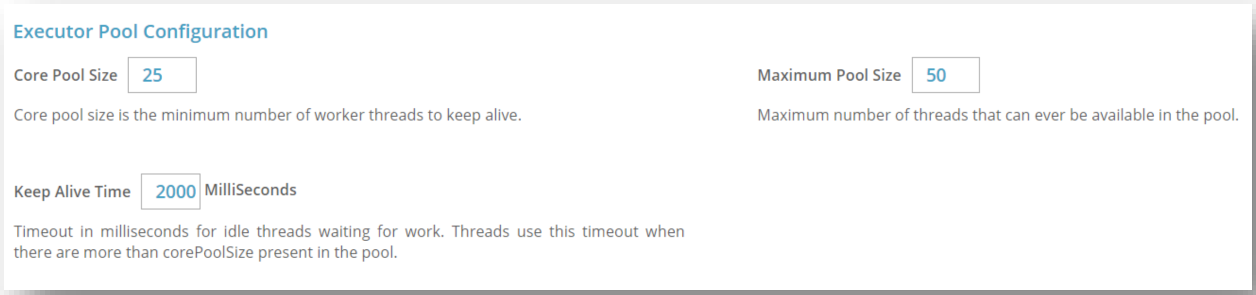
The methods available on an empty Future are:

- cancel()
- get()
- isCancelled()
- isDone()
- complete(value)

Executor Pool Configuration

In the **Server Settings** section in the Administrator, we have added **Executor Pool Configuration**, which enables you to specify values for:

- *Core pool size:* Core pool size is the minimum number of worker threads to keep alive. The value should be less than the value specified in **Maximum Pool Size**. The default value sis 25.
- *Maximum pool size:* Maximum number of threads that can ever be available in the pool. The default value is 50.
- *Keep alive time:* Timeout in milliseconds for idle threads waiting for work. Threads use this timeout when there are more than the corePoolSize present in the pool. The default value is 2000 ms.


Executor Pool Configuration

Core Pool Size 25

Core pool size is the minimum number of worker threads to keep alive.

Maximum Pool Size 50

Maximum number of threads that can ever be available in the pool.

Keep Alive Time 2000 MilliSeconds

Timeout in milliseconds for idle threads waiting for work. Threads use this timeout when there are more than corePoolSize present in the pool.

These settings enable you to finetune your async executor according to your requirements. Also, these property changes take effect without any server restart.

We have also added the following Admin APIs to support the properties mentioned above. These APIs are a part of runtime.cfc.

In this release, to support the pool configuration settings, we have also added three new properties to the API, getRuntimeProperty(required propertyName). They are:

- corePoolSize
- maxPoolSize
- keepAliveTime

For example,

```
<cfscript>
    // Login is always required.
    adminObj = createObject("component","cfide.adminapi.administrator");
    adminObj.login("admin");
    runtimeObj=createObject("component","cfide.adminapi.runtime");
    corePool=runtimeObj.getRuntimeProperty("corePoolSize");
    writeOutput("core pool size is: " & corePool & "<br/>");
    maxPool=runtimeObj.getRuntimeProperty("maxPoolSize");
    writeOutput("max pool size is: " & maxPool & "<br/>");
    keepAlive=runtimeObj.getruntimeProperty("keepAliveTime");
    writeOutput("keep alive time is: " & keepAlive & "<br/>");
</cfscript>
```

Data type preservation

ColdFusion is typeless and does not evaluate or preserve the type information at the time of code generation. At runtime, ColdFusion tries its best to infer the datatype, which may cause an unexpected behavior in some cases. For example, at the time of JSON serialization, ColdFusion attempts at converting a string to a number, if the number is given in double quotes. If the attempt is successful, then the given data is treated as number irrespective of whether you wanted it to be a number or not.

For example, in the previous version of ColdFusion, if the data type of column was not defined, CF used to serialize everything as a string. However, with the data type preservation functionality we infer the data type while serializing the data even though the data type was not defined. In the snippet below:

```
<cfscript>
      myQuery = queryNew("id,name,ctype","",
             [
             {id=1,name="1",ctype=true},
             {id=2,name="Two",ctype=false},
             {id=3,name="3",ctype="true"},
             {id="4",name="4",ctype="false"}
             ]);
      writeOutput ( SerializeJSON(myQuery) );
</cfscript>
```

New output:

```
{"COLUMNS":["ID","NAME","CTYPE"],"DATA":[[1,"1",true],[2,"Two",false],[3,"3","true"],[
"4","4","false"]]}
```

Old output:

```
{"COLUMNS":["ID","NAME","CTYPE"],"DATA":[["1","1","true"],["2","Two","false"],["3","
3","true"],["4","4","false"]]}
```

The 2018 release of ColdFusion infers the data type of a variable at compile type. This release has enabled this feature by default. However, these is a new flag introduced to disable this feature and fallback on the previous behavior. You can add this flag to the jvm.config file:

-Dcoldfusion.literal.preservetype="false/true"

For the flag to work, ensure the following:

- Restart ColdFusion after enabling/disabling this flag
- Remove all class files from the *<cf_install_root>/<instance_name>/wwwroot/WEB-INF/cfclasses* or re-compile the CFM template

There are a few known issues while interacting with the external systems like SharePoint, dotnet, web services, etc. For more information, see the Known Issues document.

Object Oriented Programming enhancements

Abstract components and methods

An abstract component can have abstract method without body and can also have methods with implementation.

Use the **abstract** keyword to create an abstract component and method. In ColdFusion, you cannot instantiate an abstract component. An abstract component is mostly used to provide base for sub-components. The first concrete component should have the implementation of all the abstract methods in its inheritance hierarchy.

Note: A concrete component cannot have abstract methods.

To declare an abstract component,

myAbstractClass.cfc

```
abstract component {

      // abstract methods

      // concrete methods

}
```

That is all there is to declaring an abstract component in ColdFusion. Now you cannot create instances of myAbstractClass. Thus, the following code is **NOT** valid:

```
myClassInstance=new myAbstractComponent();
```

Here is an example of using abstract components in ColdFusion.

Shape.cfc

```
abstract component Shape{
      abstract function draw();
}
```

Square.cfc

```
component Square extends="Shape" {
      function draw() {
     writeoutput("Inside Square::draw() method. <br/>");
   }
}
```

ShapeFactory.cfc

```
component ShapeFactory extends="AbstractFactory" {
      Shape function getShape(String shapeType){

       if(shapeType EQ "RECTANGLE"){
         return new Rectangle();

      }else if(shapeType EQ "SQUARE"){
         return new Square();

      }else
         return "not defined";
      }
}
```

Rectangle.cfc

```
component Rectangle extends="Shape" {
   function draw() {
      writeoutput("Inside Rectangle::draw() method. <br/>");
   }
}
```

AbstractFactory.cfc

```
abstract component AbstractFactory {
     abstract Shape function getShape(String shape) ;
}
```

FactoryProducer.cfc

```
component FactoryProducer {
 AbstractFactory function getFactory(String choice){
   if(choice eq "SHAPE"){
         return new ShapeFactory();

      }

   }
}
```

index.cfm

```
<cfscript>
   //get shape factory
   shapeFactory =  new FactoryProducer().getFactory("SHAPE");
   //get an object of Shape Rectangle
   shape2 = new shapeFactory().getShape("RECTANGLE");
   //call draw method of Shape Rectangle
   shape2.draw();
   //get an object of Shape Square
   shape3 = new shapeFactory().getShape("SQUARE");
   //call draw method of Shape Square
   shape3.draw();
</cfscript>
```

Final component, method, and variable

In ColdFusion, inheritance is a highly useful feature. But at times, a component must not be extended by other classes to prevent data security breaches. For such situations, we have the **final** keyword.

In the 2018 release of ColdFusion, we have introduced the following:

- Final component
- Final method
- Final variable

Final variable

Final variable are constants. Use the **final** keyword when declaring a variable. You cannot change the value of a final variable once you initialize it. For example,

```
component{
	final max=100;

	function mymethod(){
		//max=100;
		writeoutput("final variable is: " & max);
	}
}
```

Create an index.cfm, instantiate the component, and call the method defined in the component. If you change the value of max in the method, you see an exception. The following snippet also produces an exception.

```
component{
	final max;
	// constructor

	function mymethod(){
		max=50;
		writeoutput("final variable is: " & max);
	}
}
```

Final component

You cannot extend a final component and override a final method. A final component, however, can extend other components.

To define a final component, see the example below:

```
final component{

	// define methods

}
```

To define a final method, use the specifier before the final keyword and the return type. For example,

```
public final void function myMethod() {
        // method body
}
```

A final component helps secure your data. Hackers often create sub-components of a component and replace their components for the original component. The sub-component looks like the component, and may perform potentially malicious operation, which would compromise the entire application. To prevent such subversion, declare your component as final and prevent the creation of any sub-component. For example,

A.cfc

```
final component {
}
```

The code below produces an exception:

B.cfc

```
component extends=”A” {
}
```

Final method

A method with a final keyword cannot be overridden. This allows you to create functionality that cannot be changed by sub-components. For examples,

Test.cfc

```
component {
        // final method and cannot be overridden
        public final void function display(){
                writeOutput("From super class, final display() method");
        }
        // concrete method
        public void function show(){
                writeOutput("From super class, non-final show() method");
        }
}
```

Demo.cfc

```
component extends="Test"{
        //public void function display(){}
        public void function show()
        {
        writeOutput("From subclass show() method");
        }
}
```

In Demo.cfc, uncommenting the above line produces an exception, since you cannot override a Final method. You can however override a non-final method, as shown in the code above.

main.cfm

```
<cfscript>
	d1=new Demo();
	d1.display();
	d1.show();
</cfscript>
```

Produces the output,

From super class, final display() method

From subclass show() method

Default functions in interfaces

This feature enables you to declare a function in an interface, and not just the signature. While extending the interface, developers can choose to either override or reuse the default implementation.

Default Functions enables extending the interfaces without breaking old applications.

For example, create an interface, **I.cfc**, that defines a function **returnsany**, which returns an object of any type. Use the keyword "**default**", while defining such functions.

```
interface {
	public default any function returnsany(any obj)
	{
		return obj;
	}
}
```

Create a cfc, **comp.cfc**, that implements the interface, I.cfc.

```
component implements="I"
{

}
```

Now, create on object that returns a string.

```
<cfscript>
	myobj=new Comp();
	writeOutput(myobj.returnsany("hello world"));
</cfscript>
```

Similarly, create a cfm that returns an array object.

```
<cfscript>
	myobj=new Comp();
	writeDump(myobj.returnsany(arrayNew<String>(1)));
</cfscript>
```

Covariance

Covariance is the relationship between types and subtypes. In ColdFusion (2018 release), Covariance is supported for return types in overriding a method.

In ColdFusion (2018 release), a return type is covariant in the component implementing the interface. The example below shows how you can leverage covariance.

The argument name is not compared during interface implementation, i.e, if the interface function takes argument name as x , the implementing function in the component can have any argument name.

1. Define an interface, **Animal.cfc**.

```
interface {
      IPerson function returnsAny();
      void function acceptAny(required string x, required numeric y, required
numeric z);
}
```

2. Define an interface, **IPerson.cfc**.

```
interface {
}
```

3. Define an interface, **CPerson.cfc**, that implements the interface defined in **I.cfc**.

```
interface extends="IPerson"{
}
```

4. Define an interface, **Person.cfc**, that extends **CPerson**, and defines the properties of the object defined in **ReturnsString.cfc**.

```
component displayName="Person" implements="CPerson"{
      this.firstName="John";
      this.lastName="Doe";
}
```

5. Define the functions, returnsAny and acceptAny, in **ReturnsString.cfc**.

```
  component implements="I"{

         Person function returnsAny(){
           obj1 = createObject("component", "Person");
           return obj1;
}


      void function acceptAny(required string a,required numeric b,required numeric
c)
{
          WriteOutput(a);
          WriteOutput(b);
          WriteOutput(c);
}
}
```

6. Finally, create an **index.cfm**, that calls the component, **ReturnsString**, and further invokes the function, **returnsAny**.

```
<cfscript>
	writeDump(invoke("ReturnsString", "returnsAny"));
</cfscript>
```

Optional semicolon in cfscript

In this release, we have added support for making semicolons optional in **cfscript**. For example, the snippet below does not use any semicolon at the end of a statement.

```
<cfscript>
	animals = ['cat','dog','fish','bison']
	lastAnimal=animals.last()
	writeOutput("The last element of the array is: " & lastAnimal)
</cfscript>
```

A semicolon is optional in **for** and **do-while** statements. For example,

```
<cfscript>
	for(i=1;i<9;i++){
		for(j=1;j<=i;j++)
			writeoutput("*")
		writeoutput("<br/>")
	}
</cfscript>
```

A semicolon is optional in generic syntax for ColdFusion tags. For example,

```
<cfscript>
	cfhttp(url="http://localhost:8500", timeout="60")
</cfscript>
```

A semicolon is also optional in script-based components. For example,

```
component{
	function myfunc(){
		writeoutput("Optional Semi Colon Example")
		}
}
```

In closures in tags,

A semi-colon is also optional while defining the body of closure functions in tags. For example,

```
<cfset closureFunc1=function(){
		return true
	    }>
```

Named parameters

In the 2018 release of ColdFusion, we have introduced named parameters for the functions, that are listed here.

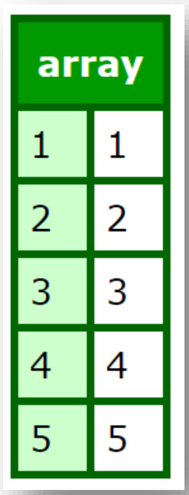
Array slicing

In ColdFusion (2018 release), to slice an array means extracting elements from an array depending on a start and stop. Typically, you specify a first index, a last index, and an optional step.

For example, in the following script,

```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[1:6])
</cfscript>
```

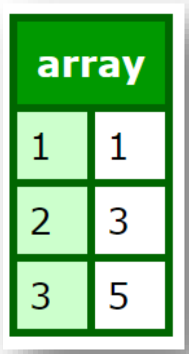
You see the following output:



array	
1	1
2	2
3	3
4	4
5	5

Similarly,

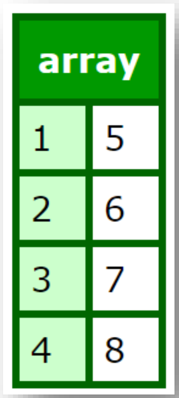
```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[1:6:2]) // In steps of 2
</cfscript>
```



array	
1	1
2	3
3	5

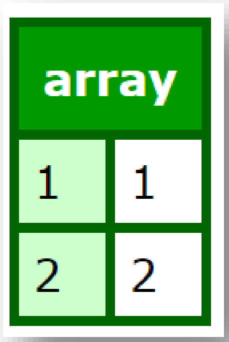
Also,

```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[5:]) // from index 5 till the end of the array
</cfscript>
```



array	
1	5
2	6
3	7
4	8

```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[:3]) // All elements till index 3, exclusive of index 3
</cfscript>
```

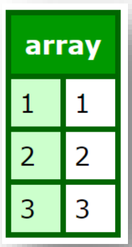


1	1
2	2

```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[:]) // returns the entire array
</cfscript>
```

Using negative indices,

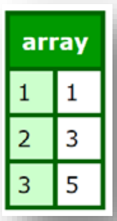
```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[:-5]) // Returns all elements from the array, except the last 5 
elements
</cfscript>
```



```
array
1  1
2  2
3  3
```

Similarly,

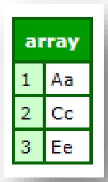
```
<cfscript>
	a=[1,2,3,4,5,6,7,8]
	writedump(a[:-2:2])
</cfscript>
```



```
array
1  1
2  3
3  5
```

In case of strings,

```
<cfscript>
	values = ["Aa","Bb","Cc","Dd","Ee"]
	odds = values[::2]
	writedump(odds)
</cfscript>
```



```
array
1  Aa
2  Cc
3  Ee
```

Member functions

String literals

In this release of ColdFusion, we have added support for string literals. For example,

The following snippet also produces same output **olleh**.

```
<cfscript>
        str="hello";
        writeoutput(str.reverse());
</cfscript>
```

Chaining more than one method to a string literals is also possible. For example,

```
<cfscript>
        writeOutput("ABCDEFGHIJ".substring(1,7).substring(2,5).lCase().len());
</cfscript>
```

Numeric member functions

In this release, we have introduced equivalent member functions for the following numeric functions:

• someVar.abs()
• someVar.aCos()
• someVar.aSin()
• someVar.atn()
• someVar.bitAnd(number2)
• someVar.bitMaskClear(start, length)
• someVar.bitMaskRead(start, length)
• someVar.bitMaskSet(mask, start, length)
• someVar.bitNot()
• someVar.bitOr(number2)
• someVar.bitSHLN(count)
• someVar.bitXor(number2)
• someVar.ceiling()
• someVar.cos()
• someVar.decrementValue()
• someVar.exp()
• someVar.fix()
• someVar.floor()
• someVar.formatBaseN(radix)
• someVar.incrementValue()
• * someVar.inputbasen() – does not work as expected in this release
• someVar.log()
• someVar.log10()
• someVar.max(number2)
• someVar.min(number2)
• someVar.round()
• someVar.sgn()
• someVar.sin()

• someVar.sqr()
• someVar.tan()
• someVar.bitSHRN(count)
• someVar.precisionEvaluate()
• someVar.randomize([algorithm])
• someVar.randRange(number2[, algorithm])

For example,

```
<cfscript>
	x=10;
	whichIsBigger=x.Max(10.01);
	writeOutput(whichIsBigger); // displays 10.01
</cfscript>
```

Also,

```
<cfscript>
	a=0.3;
	writeoutput(a.acos()); // displays 1.26610367278
</cfscript>
```

Typed collections

This release of ColdFusion (2018 release) supports declaring arrays and structs with a set of related types.

In previous versions of ColdFusion, you could create an array with the following code:

```
<cfscript>
      arr=arrayNew() // or arr=[];
      arr=["hello","world"];
      writeDump(arr);
</cfscript>
```

In this release, using type, you can rewrite the above as:

```
<cfscript>
      arr=arrayNew['String'](1);
      arr.add("hello");
      arr.add("world");
      writeDump(arr);
</cfscript>
```

The output is an array of strings.

Note: The type declaration allows you to insert data of only the declared type.

Typed arrays include the following:

- Support for inheritance while inserting CFCs
- Support for typed arrays in method argument (`numeric function getMax(numeric[] numbers)`)
- Support for typed arrays in method return type (`Student[] function getStudents(numeric[] studentIds )`)

In typed arrays, the following are the datatypes supported:

- String
- Numeric
- Boolean
- Date / Datetime
- Array
- Struct
- Query
- Component
- CFC (By Name) – Allows extended components
- Binary
- Function

New operator support using name-spaces for java, com and cfc

In this release, we have added support for new operator/syntax for com, component, CORBA, Java,. Net, webservice. For examples,

- obj1 = new java("java.lang.String");
- m = new component("employee ");

For example, in the following cfcs (using **Super** keyword):

Employee.cfc

```
component {
	public numeric function getPaid() {
		var salary=40*20;
		return salary;
	}
}
```

Manager.cfc

```
component extends="employee"{
	public numeric function getPaid(){
		var salary=1.5*Super.getPaid();
		return salary;
	}
}
```

President.cfc

```
component extends="manager"{
	public numeric function getpaid(){
		var salary=1.5*Super.getPaid();
		return salary;
	}
}
```

Payday.cfm

```
<cfscript>
	empObj=new component:employee();
	manObj=new component:manager();
	presObj=new component:president();
	writeOutput("Employee earns: " & empObj.getPaid() & "<br/>");
	writeOutput("Manager earns: " & manObj.getPaid() & "<br/>");
	writeOutput("President earns: " & presObj.getPaid() & "<br/>");
</cfscript>
```

Chaining of member functions

In this release, you can chain member functions together to produce a desired output.

The following member functions will return array or struct objects instead of Boolean values after this release:

- arrayObj.Clear()
- arrayObj.DeleteAt()
- arrayObj.Delete()
- arrayObj.DeleteNoCase()
- arrayObj.InsertAt()
- arrayObj.Prepend()
- arrayObj.Resize()
- arrayObj.Set()
- arrayObj.Sort()
- arrayObj.Swap()
- structObj.Insert()
- structObj.Update()

Example 1,

```
<cfscript>
       myarray=ArrayNew(1);
       new_arr= myarray.clear().Resize(10).set(1,10,"blah");
       writedump(new_arr);
</cfscript>
```

Example 2,

```
<cfscript>
     ordered_struct=["key1":"val1","key2":"val2"]
     unordered_struct={"key3":"val3","key4":"val4"}
        new_struct=unordered_struct.append(ordered_struct).update("key4","updated
val4").insert("key5","inserted val5");
     writedump(new_struct);
     writedump(unordered_struct);
</cfscript>
```

Example 3,

```
<cfscript>
       myResult=QueryExecute(("SELECT * FROM ORDERS WHERE ORDERID BETWEEN :low AND
:high"),{low=1,high=23},
{datasource="cfartgallery"});

       // chaining methods
       a=myResult.deleteColumn("PHONE").filter(function(item){
              return item.STATE=="CO" || item.STATE=="CA";
       }).valueArray("TOTAL");
       WriteDump(a);

       // use reduce function to calculate sum of all prices
```

```
	red=a.reduce(function(prev,next){
		return prev+next; // sum of all prices
	},0);
	writeoutput(red);
</cfscript>
```

Simplified date format functions

To make date member functions consistent and convenient, we have introduced a new member function **dateObj.format()**, which is equivalent to the member function **dateObj.dateTimeFormat()**.

Syntax

dateObj.format(date,[mask,timezone])

For example,

```
<cfscript>
	todayDateTime = Now()
	writeOutput("Today's date and time are" & #todayDateTime# & "<br/>")
	writeoutput("Using .Format(), we can display that date and time in different 
ways: ")
	writeoutput(#todayDateTime.Format()#)
	writeoutput(#todayDateTime.Format( "yyyy.MM.dd G 'at' HH:nn:ss z")#)
	writeoutput(#todayDateTime.Format("EEE, MMM d,yy")# )
	writeoutput(#todayDateTime.Format( "h:nn a")# )
	writeoutput(#todayDateTime.Format( "hh 'o''clock' a, zzzz")# )
	writeoutput(#todayDateTime.Format( "K:nn a, z")# )
	writeoutput(#todayDateTime.Format( "yyyyy.MMMMM.dd GGG hh:nn aaa")# )
	writeoutput(#todayDateTime.Format( "EEE, d MMM yyyy HH:nn:ss Z")# )
	writeoutput(#todayDateTime.Format( "yyMMddHHnnssZ", "GMT")# )
</cfscript>
```

ArrayFirst and ArrayLast functions

There are two new array functions in this release. They are:

- ArrayFirst
- ArrayLast

Member functions for the above also exist as array.First() and array.Last() respectively. For example, animals.Last() returns the last element in the animals array.

ArrayFirst

Description
Gets the first element from an array.

Returns
The first array element.

Syntax
ArrayFirst(Obj array)

History
New in ColdFusion Aether

Parameters

Parameter	Req/Opt	Description
Array	Required	The input array from which the first element is to be displayed.

Example
```
<cfscript>
	animals = ['cat','dog','fish','bison'];
	firstAnimal=ArrayFirst(animals);
	writeOutput("The first element of the array is: " & firstAnimal);
</cfscript>
```

ArrayLast

Description
Gets the last element from an array.

Returns
The last array element.

Syntax
ArrayLast(Obj array)

History
New in ColdFusion Aether

Parameters

Parameter	Req/Opt	Description
Array	Required	The input array from which the last element is to be displayed.

Example
```
<cfscript>
	animals = ['cat','dog','fish','bison'];
	lastAnimal=ArrayLast(animals);
	writeOutput("The last element of the array is: " & lastAnimal);
</cfscript>
```

Negative Indices support for Arrays

In this release of ColdFusion, we have provided support for negative indices in arrays.

Support for indices from the end in an array. For example,

```
        <cfscript>
                animals = ['cat', 'dog', 'fish', 'bison'];
                writeOutput(animals[-1]); //gets the last element
                writeOutput(animals[-2]); //gets the last but one element
</cfscript>
```

QueryDeleteColumn

Description

Removes a column from a query object.

Returns

Modified query object after the deletion of the specified column.

Syntax

QueryDeleteColumn(Query queryObject, String columnName)

History

New in ColdFusion Aether

Parameters

Parameter	Required/Optional	Description
queryObject	Required	The query object in which a column is to be deleted.
columnName	Required	The name of the column to be deleted.

Example

```
<cfscript>
        myResult=QueryExecute(("SELECT * FROM EMPLOYEES WHERE EMP_ID BETWEEN :low AND
:high"),{low=4,high=14},
{datasource="cfdocexamples"});
        writeOutput("Original query object" & "<br/>");
        WriteDump(myResult);
        // delete column
        result=myResult.deleteColumn("IM_ID");
        writeOutput("Deleting column IM_ID" & "<br/>");
        WriteDump(result);
</cfscript>
```

QueryDeleteRow

Description

Deletes a row from the query object.

Returns

Boolean, which indicates whether the row is successfully deleted.

Syntax

QueryDeleteRow(Object queryObject, int rowNum)

History

New in ColdFusion (2018 release)

Parameters

Parameter	Required/Optional	Description
queryObject	Required	The query object in which a column is to be deleted.
rowNum	Required	The row number corresponding to the row to be deleted.

Example

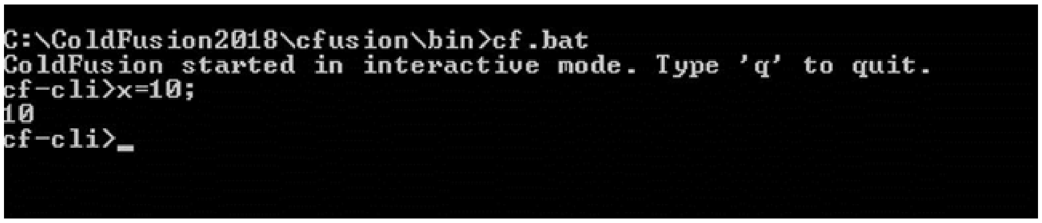
```
<cfscript>
       myResult=QueryExecute(("SELECT * FROM EMPLOYEES WHERE EMP_ID BETWEEN :low AND
:high"),{low=4,high=14},
       {datasource="cfdocexamples"});
     writeOutput("Array before deleting any row" & "<br/>");
     writeDump(myResult);
     writeOutput("Deleting row 2" & "<br/>");
     myDeletedRow=QueryDeleteRow(myResult,2);
     // Display output after row deletion
     writeOutput("<br/>" & myDeletedRow);
     writeOutput("<br/>" & "Array after deleting the row" & "<br/>");
     WriteDump(myResult);
</cfscript>
```

CLI- REPL

In the 2016 release of ColdFusion, we had introduced support for Command Line Interface. In this release, we have introduced support for Read-Eval-Print-Loop (REPL). REPL, a shell, is an interactive programming environment that takes single user inputs, evaluates them, and returns the result to the user. An REPL is like command line shells.

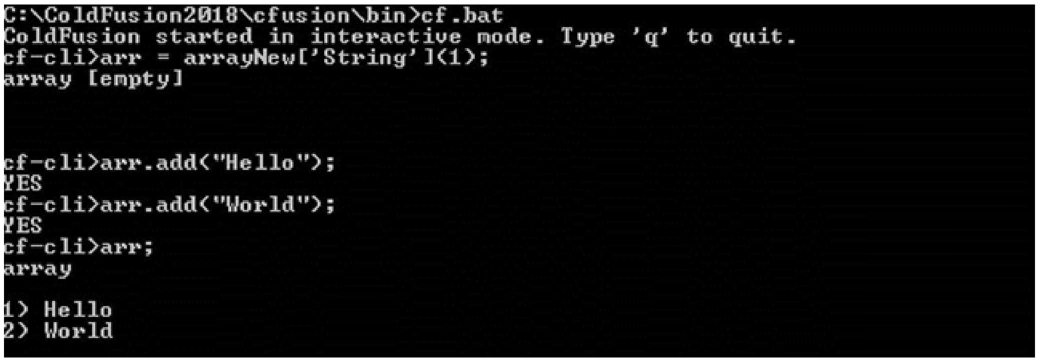
Getting started with REPL

In the command-prompt mode, navigate to <CF_HOME>/cfusion/bin. Enter cf.bat and the REPL mode displays.



```
C:\ColdFusion2018\cfusion\bin>cf.bat
ColdFusion started in interactive mode. Type 'q' to quit.
cf-cli>x=10;
10
cf-cli>_
```

To test if the REPL works as expected, assign a value to a variable, and display the variable, as shown below:



```
C:\ColdFusion2018\cfusion\bin>cf.bat
ColdFusion started in interactive mode. Type 'q' to quit.
cf-cli>arr = arrayNew['String'](1);
array [empty]


cf-cli>arr.add("Hello");
YES
cf-cli>arr.add("World");
YES
cf-cli>arr;
array

1) Hello
2) World
```

To exit the REPL mode, enter q.

Note:

- REPL works even if the ColdFusion server is NOT up and running
- REPL only supports **cfscript** syntax, not tags

Support for Admin APIs

You can execute Admin APIs through CLI as well as in REPL mode. For example,

```
<cfscript>
	createObject("component","cfide.adminapi.administrator").login("admin");
	//instantiate caching object
	myObj = createObject("component","cfide.adminapi.debugging");
	//get all IP List
	returnValue = myObj.getIPList();
	//set IP
	myObj.setIP("10.10.10.10");
```

```
            myObj.setIP("22.22.22.22");
            //get all IP List
            returnValue = myObj.getIPList();
            //delete IP
            myObj.deleteIP("10.10.10.10");
            myObj.deleteIP("22.22.22.22");
            returnValue = myObj.getIPList();
    </cfscript>
```

A CLI instance and the server share the same configuration files. Any admin settings changed from the CLI are reflected in CLI immediately. The changes are also applied to the server. However, to reflect the changes in the ColdFusion Administrator Pages, restart the server. Any changes to settings done from ColdFusion administrator applies to CLI as well. Restart CLI, to reflect the changes.

REPL is more enhanced

To open the console in the REPL mode, run cf.bat/cf.sh without any arguments. If you open the console in REPL mode, it does the job of CLI as well now. You can specify the path of a cfm in REPL mode and execute it. Like CLI, this also handles both positional and named arguments to the cfm.

For example,

```
cf-cli>C:\ColdFusion\cfusion\wwwroot\run.cfm arg1 arg2
```

Executing cfms in REPL mode is much faster than running from CLI mode, there is no requirement for re-initialization of services. There is null support and semicolon is optional at the end of the line. When a single line is typed, it is evaluated and printed.

Support for multi-line statements

You can type in multiple lines in the REPL mode. A multi-line can be introduced in two ways:

1. *Auto-detection of logical blocks*

If you type `function add(){` in the REPL mode and press Enter, it will automatically go to the next line and waits for the next line of input in the console.

2. *Adding ^ (caret)*

Add ^ at the end of line, and if you hit Enter, it will go to the next line and waits for the next line of input. You can use this when you have a line to type in which doesn't have any logical block opening(**{ / (**)

When the input line has any logical block opening, you need not type in ^. Use it when you don't have it.

For example, in a line you just want to type in '`function add()`' and '`{`' in the next line, you have to type as the line as `function add()^`

When you want to exit from the multiline midway, you can type in '`multilineexit`'

For example,

```
cf-cli>x=1

1


cf-cli>if (x==1)^

...writeOutput("Hello ColdFusion")

Hello ColdFusion
```

Set Working directory

The working directory from where cf.bat is started to be treated as reference point for searching cfcs when they are referred in REPL.

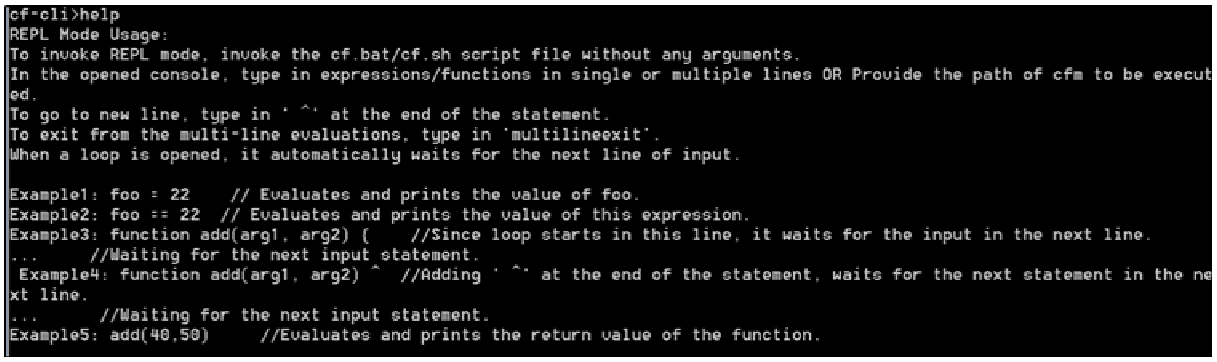
For example,

```
C:\apps\Adobe\ColdFusion\2018\express\cfusion\wwwroot\REPL\example>..\
..\..\bin\cf.bat
```

ColdFusion started in interactive mode. Type 'q' to quit.

cf-cli>

For more information, type **help** in the command-prompt for all REPL usage options.



```
cf-cli>help
REPL Mode Usage:
To invoke REPL mode, invoke the cf.bat/cf.sh script file without any arguments.
In the opened console, type in expressions/functions in single or multiple lines OR Provide the path of cfm to be execut
ed.
To go to new line, type in ' ^' at the end of the statement.
To exit from the multi-line evaluations, type in 'multilineexit'.
When a loop is opened, it automatically waits for the next line of input.

Example1: foo = 22    // Evaluates and prints the value of foo.
Example2: foo == 22  // Evaluates and prints the value of this expression.
Example3: function add(arg1, arg2) {    //Since loop starts in this line, it waits for the input in the next line.
...     //Waiting for the next input statement.
 Example4: function add(arg1, arg2) ^  //Adding ' ^' at the end of the statement, waits for the next statement in the ne
xt line.
...      //Waiting for the next input statement.
Example5: add(40,50)     //Evaluates and prints the return value of the function.
```

REST enhancements

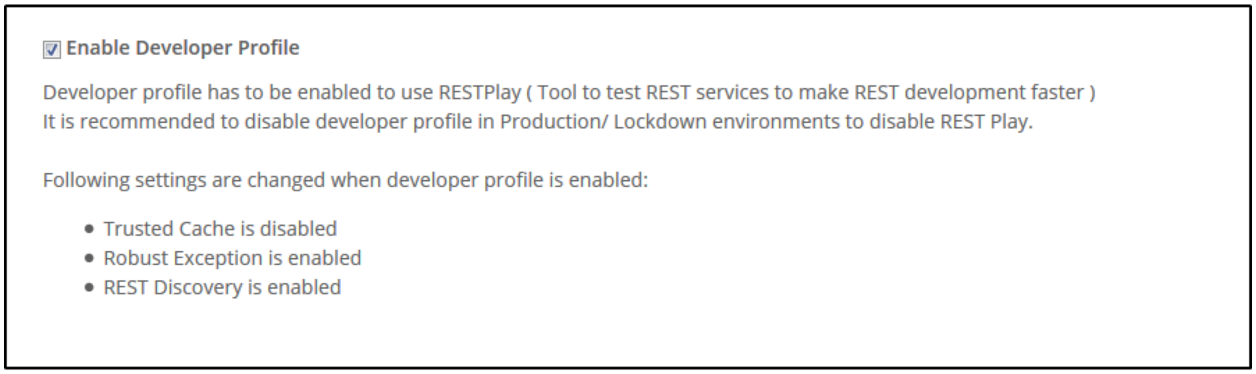
In the 2016 and earlier versions of ColdFusion, after developing a REST service, you had to log in to the ColdFusion Administrator to register the REST application. Also, with every change in the app, you had to re-register/refresh the REST app in the Administrator. In addition, there were issues in reporting and logging of errors related to the CFCs.

The highlights of this release are as follows:

- Profile toggling "**Enable Developer Profile**" in CF Administrator
- REST Playground application to try out registered APIs
- Auto refresh of REST services (if trusted cache is off)
- Language changes (component as argument, lifecycle methods)
- Easy error tracking and debugging
- PATCH support
- Custom REST path
- Under development/qa- JSON field projection
- Auto register the REST APIs (by accessing URL)

Enabling Developer Profile option

To use the REST Playground, you must enable the option **Developer Profile**, while installing ColdFusion. Also, after installation, verify that the option **Enable Developer Profile** is selected in **Debugging & Logging > Developer Profile**.



☑ **Enable Developer Profile**

Developer profile has to be enabled to use RESTPlay (Tool to test REST services to make REST development faster)
It is recommended to disable developer profile in Production/ Lockdown environments to disable REST Play.

Following settings are changed when developer profile is enabled:

- Trusted Cache is disabled
- Robust Exception is enabled
- REST Discovery is enabled

NOTE: It is recommended to enable Developer profile only during REST Services development.

Introducing REST Playground application

ColdFusion provides an application, **REST Playground**, located in the webroot, which you can launch in the browser, and then traverse to the location of the application. It's a client to test your ColdFusion REST services.

To use the REST Playground app, the Developer Profile must be enabled.

In the browser bar, enter `<host name>:<port>/REST Playground`. In the REST playground, you can add and manage your REST services, without involving the ColdFusion Administrator.

Note:

Since REST Playground application is not meant for production, the application is accessible only through internal server (not accessible through Web server).

REST Playground pulls the resources from the swagger docs and lists the resources. These are refreshed when there is a change in the CFC method and the service is invoked.

A status code of 500 is sent to the client when there are any server runtime errors.

If there are any compile time errors, they are shown when trying to access the service, or when refreshing the application (or even clicking on the application).

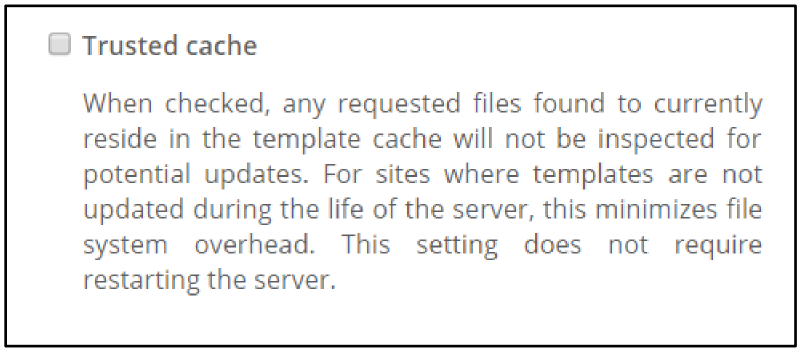
Sending the Response Status Code as 401 when the developer profile is disabled on the server when REST Playground tries to access the services.

Auto refresh of REST CFCs

Any change to the REST CFCs are reflected immediately on REST request, similar to the behavior of any cfm/cfc.

To reload the CFC or to load from cache is dependent on the server setting Trusted Cache (**Server Settings > Caching**). The same applies to REST CFCs as well.

Note: If you enable the Developer Profile option, the Trusted Cache options gets disabled.

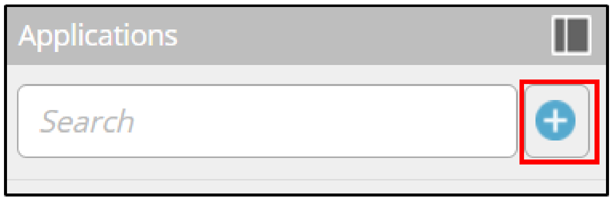


☐ Trusted cache

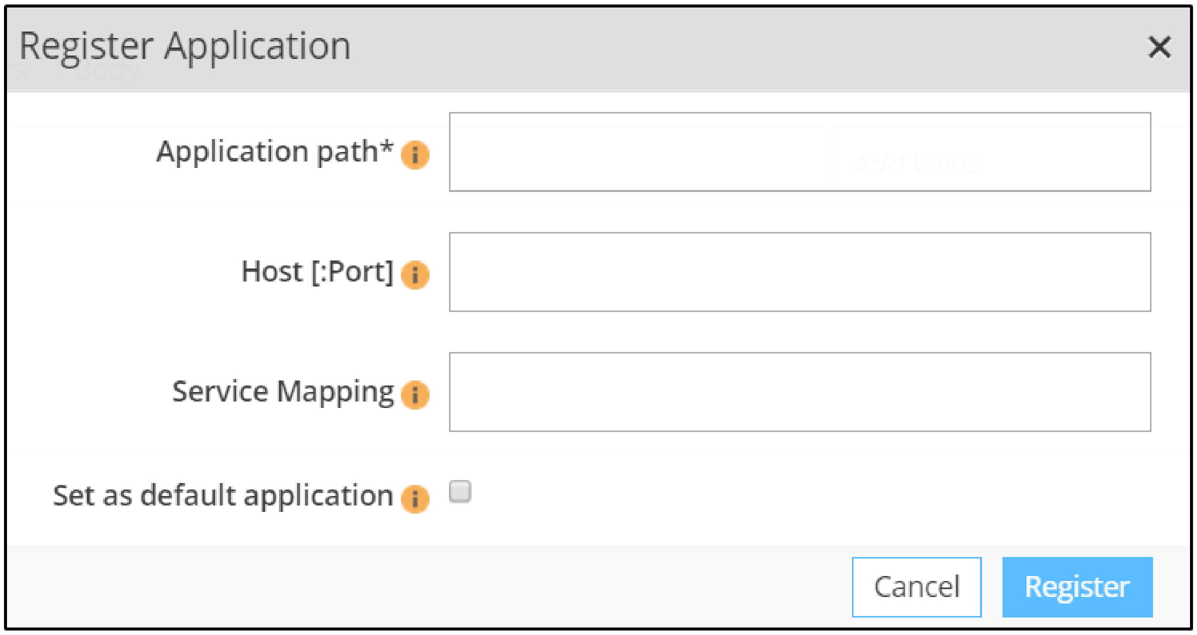
When checked, any requested files found to currently reside in the template cache will not be inspected for potential updates. For sites where templates are not updated during the life of the server, this minimizes file system overhead. This setting does not require restarting the server.

Adding an application in REST Playground

After you create a REST application, add the application through the REST Playground application. Launch REST Playground (***localhost:8500/restplay***), and click +, as shown below:

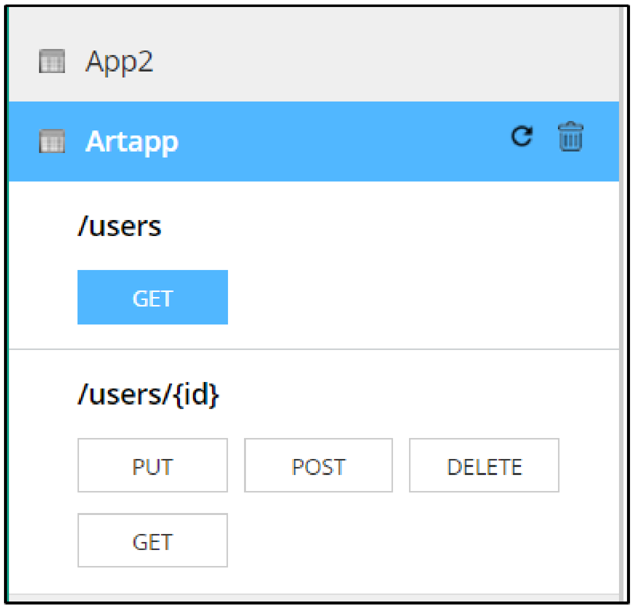


After clicking the button, enter the details in the following dialog:



Application path (Required)	The location of the REST-enabled CFC.
Host (Optional)	Host name for the REST service. For example, localhost:8500.
Service Mapping (Optional)	Identifier for the REST application when calling the service.
Set as default application (Optional)	Set an application as default to exclude the name of the application in the URL, when invoking the service. For a host, you can make only one application as default.

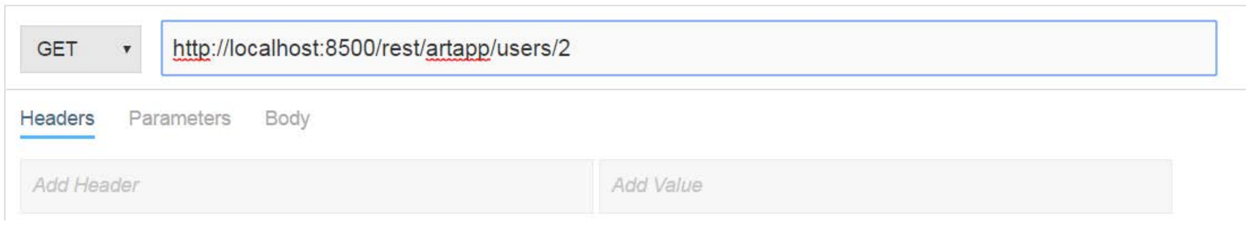
When you click **Register**, your REST app features in the portal, as shown below:



Depending on your REST service, the REST actions are displayed in the application pod. When you add any action, click the refresh button at the top, and the action displays.

Example: GET

To retrieve the details of a user, pass the user id in the URL, as shown below:


GET

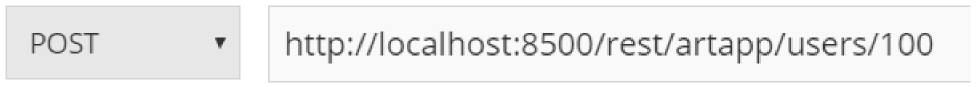
http://localhost:8500/rest/artapp/users/2

After clicking **Send**, you get the following response:

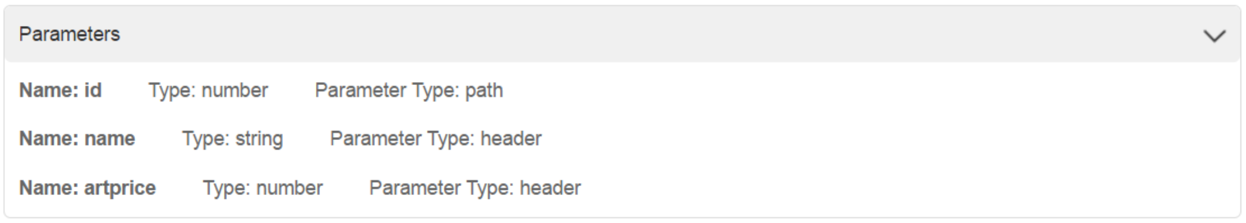
```
{
    "price": 13900,
    "id": 2,
    "name": "Michael"
}
```

Example: POST

Depending on the parameters defined, you can issue a POST request, as shown below:

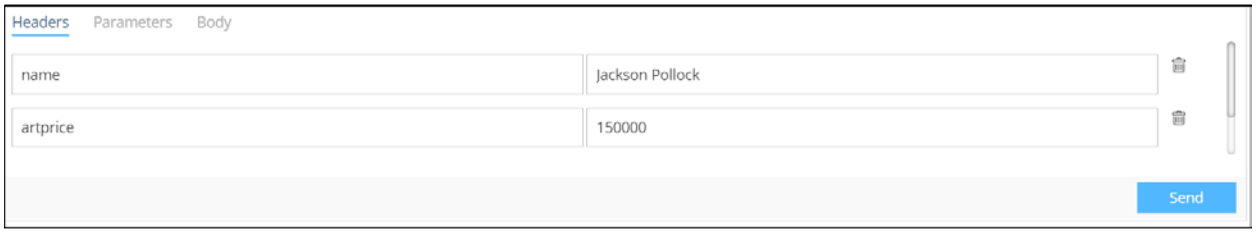

POST http://localhost:8500/rest/artapp/users/100

You can see the parameters in the section **Documentation**.


Parameters

Name: id	Type: number	Parameter Type: path
Name: name	Type: string	Parameter Type: header
Name: artprice	Type: number	Parameter Type: header

If the parameters are defined to be passed as headers, then insert the parameters in the header section, as shown below:



name	Jackson Pollock
artprice	150000

Click **Send** to insert the record in the database.

Language changes

Support for REST functions with component as argument

When the argument is component, the REST request body should be sending the corresponding JSON object.

student.cfc

```
component hint="This is a student cfc"
{

     property name="name" type="string"
displayname="name"   hint="name of student";
     property name="age"  type="numeric" displayname="age" hint="age
of student";

}
```

restCompTest.cfc

```
component restpath="student"{
remote any function setuserPost(student st) httpmethod="POST"
restpath="setStudent"
  {
        studentcomp.name = st.name;
        studentcomp.age = st.age;
        return studentcomp;
   }
}
```

REST functions

You need not add `access="remote"` in REST functions. It is assumed inherently for all the REST enabled functions.

Support for onRESTRequest

Like `OnCFCRequest`, which is called on invoking CFC, `onRESTRequest` is introduced to be called on invoking REST URL. From the URL corresponding CFC, Method & Arguments are derived and passed to the function.

```
<cffunction name="onRESTRequest" returntype="any" output="true">

<cfargument type="string" name="cfcname" required=true>
      <cfargument type="string" name="method" required=true>
      <cfargument type="struct" name="args" required=true>
      <cflog text="onRESTRequest() Before">

      <cfinvoke component = "#cfcname#" method = "#method#"
argumentCollection  = "#args#" returnVariable = "resultval">

      <cflog text="onRESTRequest() After">
      <cfreturn "#resultval#">
</cffunction>
```

Behavior of restSetResponse changed

In previous ColdFusion versions, the response set in `restSetResponse(response)` used to be ignored if the return type was non-void.

In this release, the response set in `restSetResponse(response)` is always considered.

When the same function is used as normal function(non-REST), it sends the response set in the return statement.

In case of a REST call, the response returned using the return statement is ignored.

Easy debugging and logging

Like all cfm/cfc error messages, you can see the full stack trace when there are errors. You can also see the line number where there is an error.

Compile time errors are shown as html response with full stack trace of the error.

Support for PATCH verb

For PATCH support, there should be a GET resource with same resource path as PATCH resource. This is required to fetch the corresponding object on which PATCH must be applied. The GET resource should return the Object on which Patch must be applied, while PATCH resource Patches the value.

Both the operations are merged and the result is passed on the PATCH method as argument, where you can control what you want to write in the function with the patched object.

Example of PATCH

```
component restpath="users" rest="true"
{
    import student;
    student = createobject("component","student");
    student.name = "Joe";
    student.age = 22;

   remote any function patchuser(struct x) httpmethod="PATCH"
restpath="patchuser"
    {
    //x   is patched object
       return x;
    }
    remote any function patchuserget() httpmethod="GET"
restpath="patchuser"
    {
       return student;
    }
}
```

Patch to be posted in the format as follows:

```
     [
       {
         "op" : "replace",
         "path" : "/age",
         "value" : "40"
       }
     ]
```

JSON Field Projection

Using field projection, you have the option to receive a thinned response from the server. This is ideal when you wouldn't want to receive the entire JSON object in the response, but instead, would only want filtered values. For example,

http://example.com/addresses/51234?select=region,streetAddress results in:

```
{
   "region" : "CA",
   "streetAddress" : "1234 Fake St.“,
}
```

NOTE: This feature is under active development, and may produce unexpected results.

To see a working example, create the following files:

EmployeeService.cfc

```
<cfcomponent restpath="employeeapp" >

    <cffunction name="getEmployee" returntype="Employee" restpath="{name}-{age}"
httpmethod="GET" description="retrieve employee" produces="application/json">
        <cfargument name="name" type="string" required="yes" restargsource="Path"/>
        <cfargument name="age" type="string" required="yes" restargsource="Path"/>

        <cfset myobj = CreateObject("component", "Employee")>

        <cfset myobj.name = name>
        <cfset myobj.age = age>
        <cfreturn myobj>
    </cffunction>

</cfcomponent>
```

Employee .cfc

```
component
{
    property string name;
    property numeric age;
}
```

Application.cfc

```
component{
      this.name="empapp";
      this.restsettings.skipCFCWithError=false;
      this.restsettings.generateRestDoc=true;
}
```

In the REST Playground app, add the service, and try out the API, using the example, below:

http://localhost:8501/rest/jsonfieldproj/employeeapp/John-25 produces

```
{

    "NAME": "John",

    "AGE": 25

}
```

Using JSON field filtering, you can use the filter, as shown below:

http://localhost:8501/rest/jsonfieldproj/employeeapp/John-25?select=name produces

```
{
    "name": "John"
}
```

Similarly,

http://localhost:8501/rest/jsonfieldproj/employeeapp/John-25?select=age produces

```
{
    "age": 25
}
```

Auto registering REST applications

To auto register a REST application, access the application by specifying the webroot path from the browser.

You require a REST-enabled cfc , Application.cfc, and any cfm page (say index.cfm) in a folder. When you access the cfm page (from browser) , all the REST services in the folder and the sub folders gets registered.

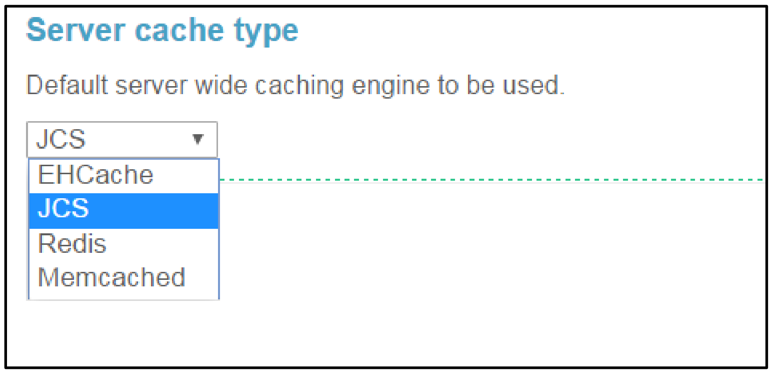
The Application.cfc needs to have `this.name` and at least one of the `this.restSettings.*` (for example, `this.restSettings.restEnabled=true`) variables .

Caching enhancements

In this release, ColdFusion adds caching support for engines, such as, Java Caching System (JCS), and Memcached, apart from the default caching engine, Ehcache.

Changes in the Administrator settings

In the ColdFusion administrator, you can choose the engine from **Server Settings** > **Caching**.



For each option, there are some configuration changes you need to make.

Memcached

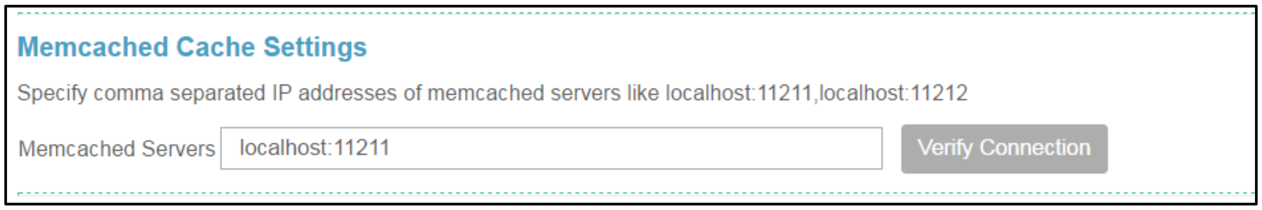
Memcached is a distributed caching solution for Java enterprise applications. A Memcached server can be shared across multiple ColdFusion instances, or multiple Memcached servers can be configured with a single ColdFusion instance.

To use Memcached, download Memcached, and run the server from the command line using the command:

```
Memcached -p <port_number>
```

The Memcached server, picks up the default port 11211.

In the ColdFusion administrator, specify the Memcached server details in **Server Settings** > **Caching**.



Memcached Cache Settings

Specify comma separated IP addresses of memcached servers like localhost:11211,localhost:11212

Memcached Servers: localhost:11211

You can also configure Memcached at the application level. There are two new application variables that you need to use to declare the caching engine.

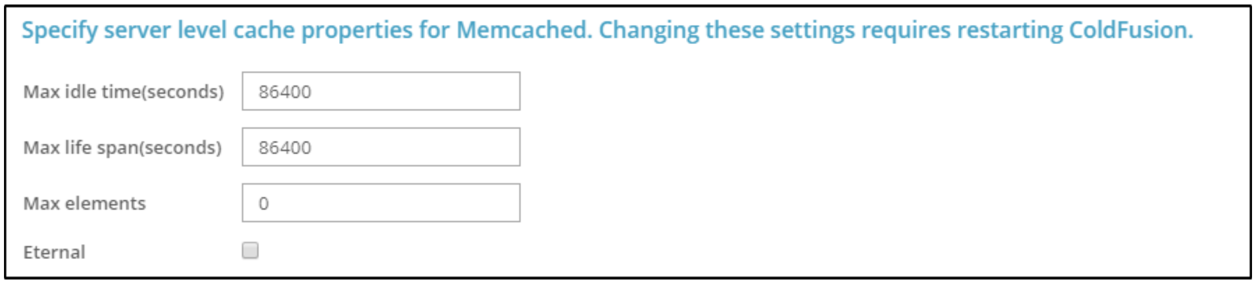
***this.cache.engine*:** Specify the caching engine to be used. Choose from jcs or memcached or ehcache.

***this.cache.configFile*:** Specify the configuration file for memcached. For example, the file can contain key-value pairs of caching properties, as shown below:

```
maxElementsInMemory=5
```

```
eternal=false
timeToIdleSeconds=30
timeToLiveSeconds=5
```

You can also modify the settings via the Administrator (**Server Settings** > **Caching**), as shown below. The interface is similar for other cache engines:



Specify server level cache properties for Memcached. Changing these settings requires restarting ColdFusion.

Max idle time(seconds)	86400
Max life span(seconds)	86400
Max elements	0
Eternal	☐

Application.cfc

```
component{
	this.name='appUsingMemcached';
	this.cache.engine='memcached';
	this.cache.configFile='memcachedconfig.properties';
	this.cache.applicationTimeout=createtimespan(0,0,0,5);
}
```

memcachedconfig.properties

```
maxElementsInMemory=5
eternal=false
timeToIdleSeconds=30
timeToLiveSeconds=10
```

cache_1.cfm

```
<cfscript>
	writeoutput(cacheGetEngineName()); // Returns the name of the cache engine
	writedump(cacheGetProperties()); // Returns the cache engine properties
</cfscript>
```

JCS

Java Caching System (JCS) is an open source caching engine released through the Apache Jakarta subproject. JCS provides in-memory caching and algorithms for selectively removing objects from the cache. It also offers more advanced features, such as, indexed disk caching and support for distributed caches.

In this release of ColdFusion, we have provided in-built support for JCS.

To use JCS, choose the option in the ColdFusion administrator.

Like Memcached, you can also configure JCS at the application level as well. Use the new application variables to set the cache engine and the caching properties file.

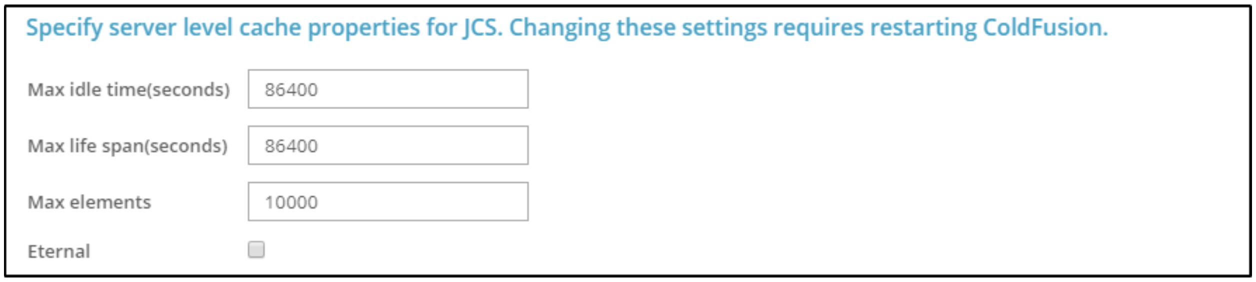
Application.cfc

```
component{
		this.name = "appSpecificCacheTest";
		this.cache.configfile = "jcsconfig.properties";
		this.cache.engine = 'jcs';
		this.applicationTimeout = createtimespan(0,0,0,5);
}
```

jcsconfig.properties

```
maxElementsInMemory=5
eternal=false
timeToIdleSeconds=30
timeToLiveSeconds=5
```

You can also modify the settings via the Administrator (**Server Settings > Caching**), as shown below:


Specify server level cache properties for JCS. Changing these settings requires restarting ColdFusion.

Max idle time(seconds)	86400
Max life span(seconds)	86400
Max elements	10000
Eternal	☐

cache_2.cfm

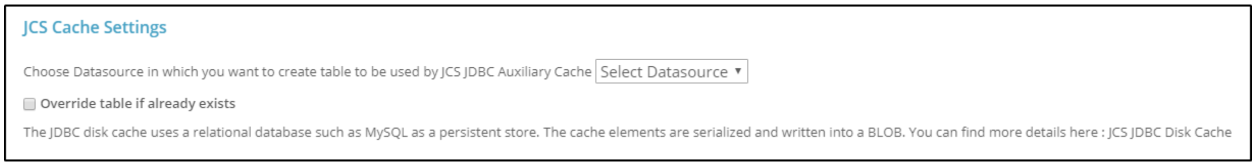
```
<cfscript>
	writeoutput(cachegetengineproperties().name); // Returns the name of the cache engine

</cfscript>
```

Auxiliary cache support in JCS

You can use JCS to persist cache into a database, which can be accessed via multiple nodes. From the ColdFusion administrator, add a data source for clustering JCS.

From the ColdFusion Administrator, click **Server Settings** > **Caching**. Choose the data source from the drop-down.



JCS Cache Settings

Choose Datasource in which you want to create table to be used by JCS JDBC Auxiliary Cache [Select Datasource ▾]

☐ Override table if already exists

The JDBC disk cache uses a relational database such as MySQL as a persistent store. The cache elements are serialized and written into a BLOB. You can find more details here : JCS JDBC Disk Cache

When you save the changes, a table, **JCS_STORE** gets created in the selected datasource.

This example uses My SQL as auxiliary cache.

To support auxiliary cache in JCS, edit the file **cache.ccf** located in *<coldfusion_install_dir>/cfusion/lib*. Add the following lines:

```
# MYSQL disk cache used for flight options
jcs.auxiliary.MYSQL=org.apache.commons.jcs.auxiliary.disk.jdbc.JDBCDiskCacheFactory
jcs.auxiliary.MYSQL.attributes=org.apache.commons.jcs.auxiliary.disk.jdbc.JDBCDiskCacheAttributes
jcs.auxiliary.MYSQL.attributes.userName=<user name>
jcs.auxiliary.MYSQL.attributes.password=<password>
# Make sure the datasource is the one in which JCS_STORE is created
# via the Administrator
jcs.auxiliary.MYSQL.attributes.url=jdbc:mysql://localhost:3306/<datasource>
jcs.auxiliary.MYSQL.attributes.driverClassName=com.mysql.jdbc.Driver
jcs.auxiliary.MYSQL.attributes.tableName=JCS_STORE
jcs.auxiliary.MYSQL.attributes.UseDiskShrinker=false
```

Also set `jcs.default = MYYSQL` in cache.ccf.

Restart ColdFusion.

The JDBC disk cache uses a relational database, such as MySQL, as a persistent data store. The cache elements are serialized and written into a BLOB.

Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database and cache. It supports data structures such as strings, hashes, lists, sets, and so on.

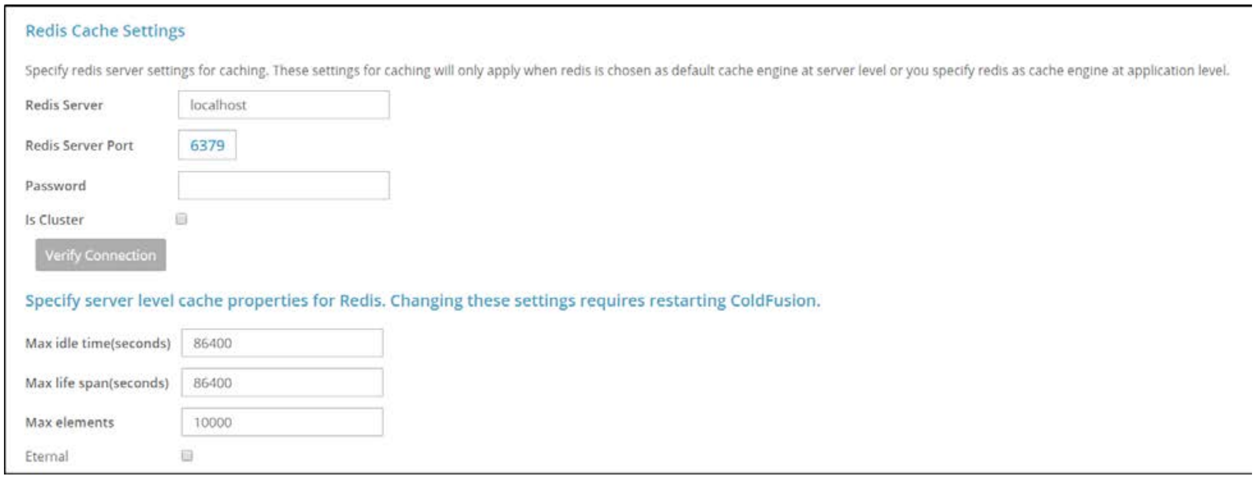
To use Redis, download Redis, and run the server from the command line using the command:

```
redis-server
```

The Redis server picks up the default port 6379.

Redis is distributed. In a clustered environment, all nodes can communicate with the same Redis node.

In the ColdFusion administrator, specify the Redis server details in **Server Settings** > **Caching**.



Redis Cache Settings

Specify redis server settings for caching. These settings for caching will only apply when redis is chosen as default cache engine at server level or you specify redis as cache engine at application level.

Redis Server	localhost
Redis Server Port	6379
Password	
Is Cluster	☐

Specify server level cache properties for Redis. Changing these settings requires restarting ColdFusion.

Max idle time(seconds)	86400
Max life span(seconds)	86400
Max elements	10000
Eternal	☐

You can also configure Redis at the application level. There are two new application variables that you need to use to declare the caching engine.

this.cache.engine: Specify the caching engine to be used. Choose from jcs, redis, memcached or ehcache.

this.cache.configFile: Specify the configuration file for redis. For example, the file can contain key-value pairs of caching properties, as shown below:

```
    maxElementsInMemory=5
    eternal=false
    timeToIdleSeconds=30
    timeToLiveSeconds=5
```

Application.cfc

```
<cfcomponent>
        <cfscript>
                this.name = "appSpecificCacheTest";
                this.cache.configfile = "redisconfig.properties";
                this.cache.engine = "redis";
                this.applicationTimeout = createtimespan(0,0,0,5);
        </cfscript>
</cfcomponent>
```

redisconfig.properties

```
    maxElementsInMemory=5
    eternal=false
    timeToIdleSeconds=30
    timeToLiveSeconds=5
```

cache.cfm

```
<cfscript>
        writeoutput(cachegetengineproperties().name); // Returns the name of the cache
engine
</cfscript>
```

Custom cache plugin

Apart from using Memcached, Redis, or JCS, you can implement your custom cache plugin. We have provided an interface (***C:\ColdFusion2018\cfusion\wwwroot\CFIDE\cache\ ICustomCache.cfc***), through which you can implement a custom caching engine.

Place your implementations in the same folder as ICustomCache.cfc.

ColdFusion takes advantage of Infinispan libraries. Download the files from the stable release,

- infinispan-embedded-query-9.1.3.Final.jar
- infinispan-embedded-9.1.3.Final.jar

in the location **C:\ColdFusion2018\cfusion\lib** and restart ColdFusion.

To add a custom plugin:

1. Create a folder in C:\ColdFusion2018\cfusion\wwwroot\CFIDE\cache. The name of the folder is the name of the custom plugin engine.
2. Create **<any CFC name>.cfc** that implements ICustomCache.cfc and write the implementations. For example,

```
< <cfcomponent implements="CFIDE.cache.ICustomCache">
          <cffunction name="put" >
            <cfargument  name="obj"  type="struct">
                    <cfoutput>"inside put"</cfoutput>
                     <cfset defaultCache=Application.defaultCacheManager.getcache()>
              <cfset defaultCache.put(obj.id,#obj.value#)>
          </cffunction>
</cfcomponent>
```

3. Create **config.xml** that contains references to Infinispan libraries.

```
<infinispan>
  <cache-container default-cache="local">
        <local-cache name="local"/>
    </cache-container>
</infinispan>
```

4. Create **Application.cfc**

```
<cfcomponent>
<cfset this.name= "xyz">
          <cfscript>

                    function onApplicationStart()
                    {
```

```
                writelog("In onApplicationStart()");
                Application.defaultCacheManager=
        CreateObject("java","org.infinispan.manager.DefaultCacheManager").init('C:\ColdFusion2018\cfusion\
        wwwroot\custom_cache\config.xml');
                                writelog("In onApplicationStart()");
                        }

                        function onApplicationEnd(){
                                writelog("In onApplicationEnd()");
                        }
          </cfscript>
        </cfcomponent>
```

Create an app with the following:

Application.cfc

```
<cfcomponent>
<cfscript>
    this.name = "mycache_app";
      this.cache.engine = "<folder name>";
</cfscript>
</cfcomponent>
```

Cache.cfm

```
<cfscript>
      writeoutput(cachegetengineproperties().name); // Returns the name of the cache
engine

</cfscript>
```

New function- getCacheServerSettings

We have introduced a new ColdFusion function for distributed caching that returns the server level properties for the redis and memcached caching engines. For example,

```
<cfscript>
      logInAPIObj = createObject("component","cfide.adminapi.administrator");
      logInAPIObj.login("password");
      rtAdminObj = createObject("component", "CFIDE.adminapi.runtime");
      memcachedSettings = rtAdminObj.getCacheServerSettings('memcached');
      redisSettings = rtAdminObj.getCacheServerSettings('redis');
      writeDump(redisSettings);
      writeDump(memcachedSettings);
</cfscript>
```

Hibernate upgrade

This version of ColdFusion uses Hibernate 5.2. The following are the highlights:

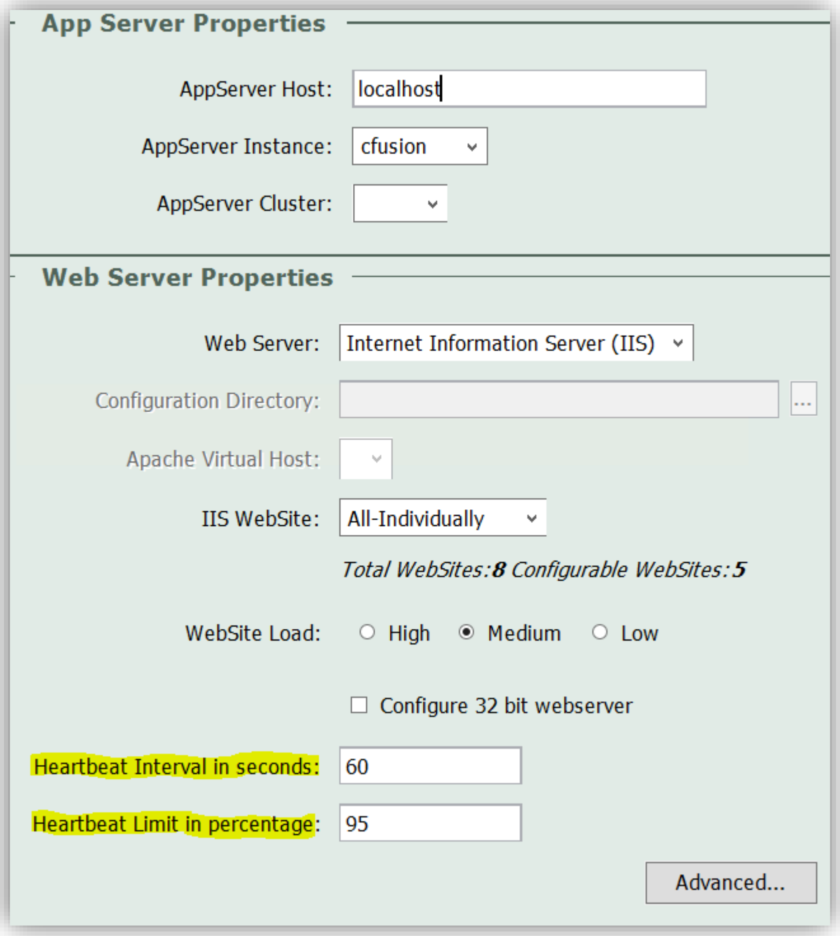
- Hibernate 5 XML Parser now uses JAXP (Java API for XML Processing). As a result, the way of declaring the class name in hbmxml ORM mapping file has changed, for example, `cfc:cfsuite.orm.settings.dialect.invalid_dialect.invalid_dialect` to `cfc.cfsuite.orm.settings.dialect.invalid_dialect.invalid_dialect`.
- Hibernate 5 no longer supports seqhilo generator.
- Hibernate 5 no longer supports offset queryoptions as it was inefficient and most of the underlying databases have the support for offset. However, this functionality can be enabled by setting the flag **hibernate.legacy_limit_handler** to true in the hibernate configuration file.
- Hibernate 5 Schema Update/Export validates the case sensitivity in table names. If the table name defined in CFC and database are in different cases, it does not work. The previous version of Hibernate allowed tables names defined in CFC and database to be case-insensitive.

Changes to wsconfig tool

In the wsconfig tool, we have added new configuration settings related to tuning of connectors.

The options are:

1. **Heartbeat Interval in seconds:** The time interval in seconds after which the connector load data is sent to ColdFusion.
2. **Heartbeat Limit in percentage:** The upper limit of load percentage at which the connector load data is immediately sent to ColdFusion.



App Server Properties

AppServer Host: localhost

AppServer Instance: cfusion

AppServer Cluster:

Web Server Properties

Web Server: Internet Information Server (IIS)

Configuration Directory:

Apache Virtual Host:

IIS WebSite: All-Individually

*Total WebSites:**8** Configurable WebSites:**5***

WebSite Load: ○ High [● Medium] ○ Low

☐ Configure 32 bit webserver

Heartbeat Interval in seconds: 60

Heartbeat Limit in percentage: 95

New Admin APIs

In this release, we have also added the following ColdFusion Admin APIs related to caching. The APIs are a part of runtime.cfc.

1. ***verifyRedisCacheStorageConnection***

Description: Verifies connection to the Redis cache storage.

Syntax: `void verifyRedisCacheStorageConnection (sessionStorageHost, numeric sessionStoragePort, sessionStoragePassword)`

Parameter	Req/Opt	Default	Description
sessionStorageHost	Optional	Any	The hostname for Redis cache storage.
sessionStoragePort	Optional	Numeric	The port number for Redis cache storage.
sessionStoragePassword	Optional	Any	The password for the Redis cache storage.

2. ***setServerCachingEngine***

Description: Changes the caching engine at the server level.

Syntax: `void setServerCachingEngine (required engine)`

Parameter	Req/Opt	Default	Description
engine	Required	Any	1. Ehcache 2. JCS 3. Memcached 4. Redis

3. ***setJCSClusterDsnName***

Description: Set the data source for JCS cluster.

Syntax: `void setJCSClusterDsnName (required dsn, required boolean createTables)`

Parameter	Req/Opt	Default	Description
dsn	Required	Any	Name of the data source.
createTables	Required	Any	Whether to create a table.

4. ***setCachingRedisServer***

Description: Set the caching engine for Redis.

Syntax: `void setCachingRedisServer (required host, required port, required password, required boolean cluster)`

Parameter	Req/Opt	Default	Description
host	Required	any	Host address of the server.
port	Required	any	Port number of the server.
password	Required	any	Password of the server.
cluster	Required	Boolean	Whether a cluster is enabled in Redis.

5. ***getMemcachedServer***

Description: Gets the details of the Memcached caching engine.

Syntax: `any getMemcachedServer ()`

ColdFusion Builder updates

- ColdFusion Builder now uses significantly less memory footprint when compared to earlier versions. It now consumes at least 100-150 MB less memory.
- ColdFusion Builder now contains support for language changes, such as, abstract and final keyword, and optional semicolon.

In this release, we have upgraded Eclipse from Mars to Oxygen. If you want to use ColdFusion Builder as a plugin to Eclipse, use Eclipse version 4.7.2 or higher.

We have also added support for the following language constructs:

- Query functions – QueryDeleteRow, QueryDeleteColumn
- Array functions - ArrayFirst , ArrayLast
- Abstract component and abstract functions
- Final methods, components, and variables
- Typed collections