**The code**

To explain the differences I will use some example code that is hopefully easy to understand.

***If you have any question or remark, please don't hesitate to add a comment at the bottom of this article.***

Consider the following HTML markup:

<level-one>

<level-two>

<level-three>

Hello {{name}}

</level-three>

</level-two>

</level-one>

and the following JavaScript:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

compile: function(tElem, tAttrs){

console.log(name + ': compile');

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link');

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link');

}

}

}

}

}

}

app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

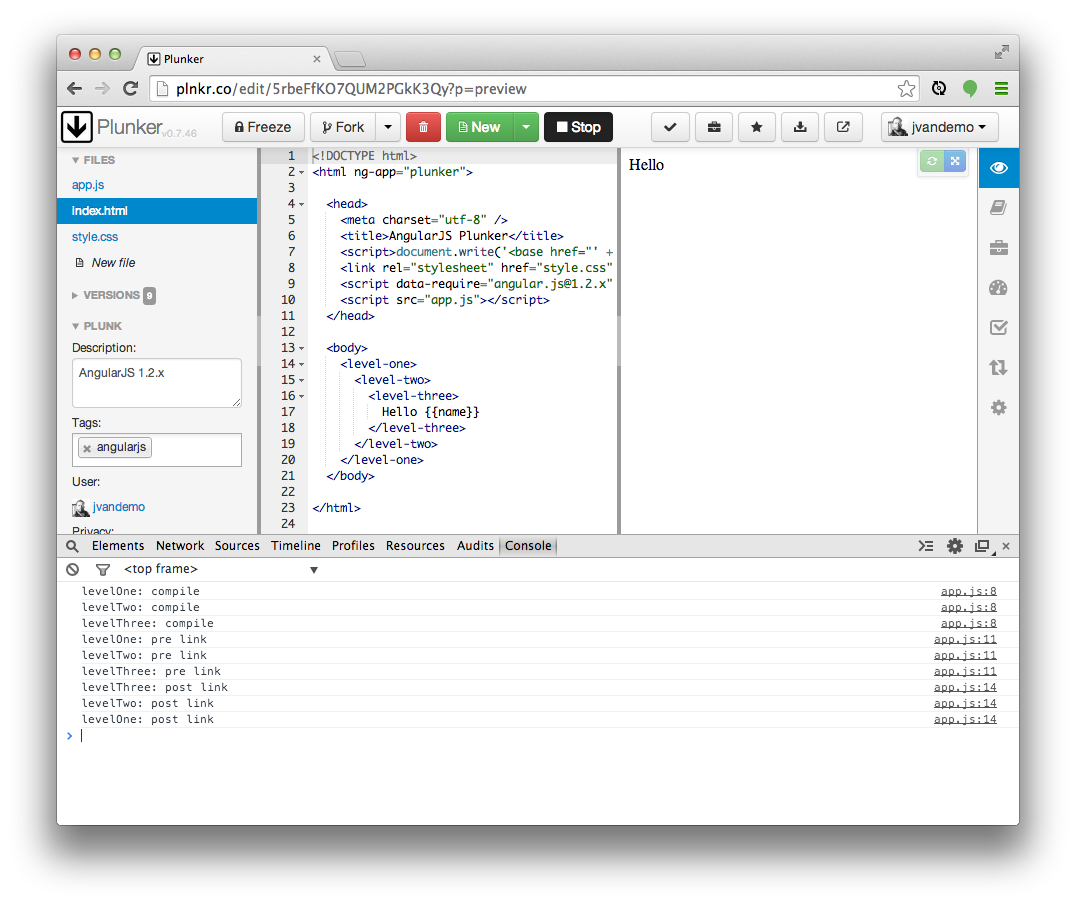
app.directive('levelThree', createDirective('levelThree'));

The goal is simple: let AngularJS process three nested directives where each directive has its own compile, pre-link and post-link function that logs a line to the console so we can identify them.

That should allow us to take a first glimpse at what is happening behind the scenes when AngularJS processes the directives.

**The output**

Here is a screenshot of the output in the console:



To try it out for yourself, just open [**this plnkr**](http://plnkr.co/edit/5rbeFfKO7QUM2PGkK3Qy?p=preview) and take a look at the console.

**Let's analyze**

The first thing to pay attention to is the order of the function calls:

// COMPILE PHASE

// levelOne: compile function is called

// levelTwo: compile function is called

// levelThree: compile function is called

// PRE-LINK PHASE

// levelOne: pre link function is called

// levelTwo: pre link function is called

// levelThree: pre link function is called

// POST-LINK PHASE (Notice the reverse order)

// levelThree: post link function is called

// levelTwo: post link function is called

// levelOne: post link function is called

This clearly demonstrates how AngularJS first compiles all directives before it links them to their scope, and that the link phase is split up in a pre-link and post-link phase.

Notice how the order of the compile and pre-link functions calls is identical but the order of the post-link function calls is reversed.

So at this point we can already clearly identify the different phases, but what is the difference between the compile and pre-link function? They run in the same order, so why are they split up?

**The DOM**

To dig a bit deeper, let's update our JavaScript so it also outputs the element's DOM during each function call:

var app = angular.module('plunker', []);

function createDirective(name){

return function(){

return {

restrict: 'E',

compile: function(tElem, tAttrs){

console.log(name + ': compile => ' + tElem.html());

return {

pre: function(scope, iElem, iAttrs){

console.log(name + ': pre link => ' + iElem.html());

},

post: function(scope, iElem, iAttrs){

console.log(name + ': post link => ' + iElem.html());

}

}

}

}

}

}

app.directive('levelOne', createDirective('levelOne'));

app.directive('levelTwo', createDirective('levelTwo'));

app.directive('levelThree', createDirective('levelThree'));

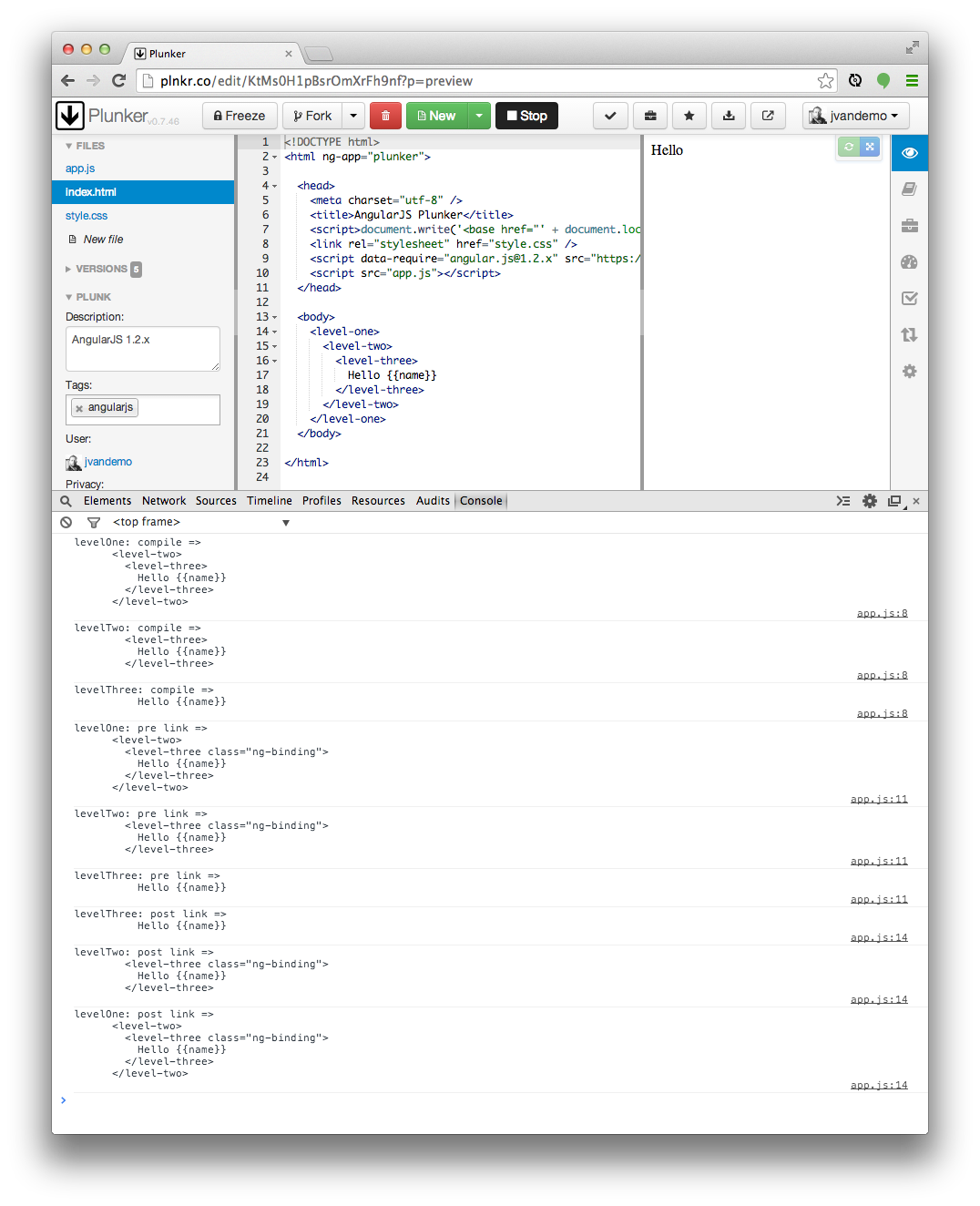
Notice the extra output in the console.log lines. Nothing else has changed and the original markup is still used.

This should provide us with more insights into the context of the functions.

Let's run the code again.

**The output**

Here is a screenshot of the output with the newly added code:



Again, if you want to try it out for yourself, just open [**this plnkr**](http://plnkr.co/edit/KtMs0H1pBsrOmXrFh9nf) and take a look at the console.

**Observation**

Printing the DOM reveals something very interesting: the DOM is different during thecompile and pre-link function.

**So what is happening here?**

Compile

We already learned that AngularJS processes the DOM when it detects that the DOM is ready.

So when AngularJS starts traversing the DOM, it bumps into the <level-one> element and knows from its directive definition that some action needs to be performed.

Because a compile function is defined in the levelOne directive definition object, it is called and the element's DOM is passed as an argument to the function.

If you look closely you can see that, at this point, the DOM of the element is still the DOM that is initially created by the browser using the original HTML markup.

***In AngularJS the original DOM is often referred to as the the template element, hence also the reason I personally use tElem as the parameter name in the compilefunction, which stands for template element.***

Once the compile function of the levelOne directive has run, AngularJS recursively traverses deeper into the DOM and repeats the same compilation step for the <level-two>and <level-three> elements.

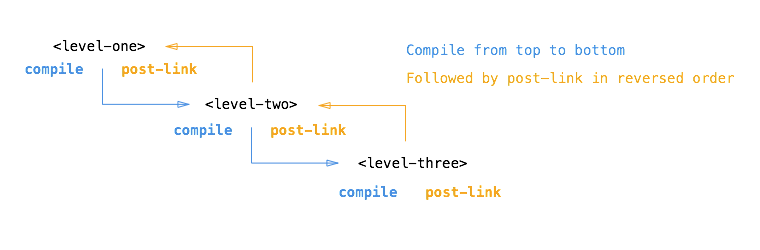
Post-link

Before digging into the pre-link functions, let's first have a look at the post-linkfunctions.

***If you create a directive that only has a link function, AngularJS treats the function as a post-link function. Hence the reason to discuss it here first.***

After AngularJS travels down the DOM and has run all the compile functions, it traverses back up again and runs all associated post-link functions.

The DOM is now traversed in the opposite direction and thus the post-link functions are called in reverse order. So while the reversed order looked strange a few minutes ago, it is now starting to make perfect sense.



This reverse order guarantees that the post-link functions of all child elements have run by the time the post-link function of the parent element is run.

So when the post-link function of <level-one> is executed, we are guaranteed that thepost-link function of <level-two> and the post-link function of <level-three>have already run.

**This is the exact reason why it is considered the *safest* and *default* place to add your directive logic.**

But what about the element's DOM? Why is it different here?

Once AngularJS has called the compile function of a directive, it creates an ***instance*** element of the template element (often referred to as ***stamping out instances***) and provides a scope for the instance. The scope can be a new scope or an existing one, a child scope or an isolate scope, depending on the scope property of the corresponding directive definition object.

So by the time the linking occurs, the instance element and scope are already available and they are passed by AngularJS as arguments to the post-link function.

***I personally always use iElem as parameter name in a link function to refer to the element instance.***

So the post-link function (and pre-link function) receive the instance element as argument instead of the template element.

Hence the difference in the log output.

Pre-link

When writing a post-link function, you are guaranteed that the post-link functions of all child elements have already been executed.

In most cases that makes perfect sense and therefore it is the most often used place to write directive code.

However, AngularJS provides an additional ***hook***, the pre-link function, where you can run code **before** any of the child element's post-link functions have run.

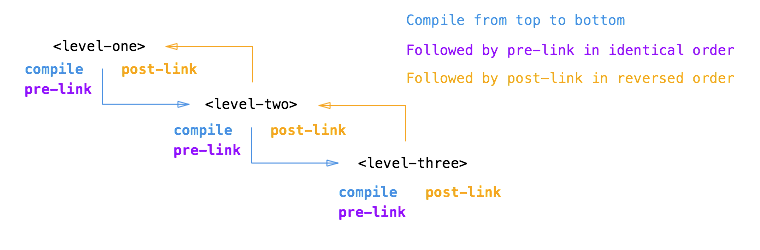
That is worth repeating:

**The pre-link function is guaranteed to run on an element instance before any post-link function of its child elements has run.**

So while it made perfect sense for post-link functions to be called in reverse order, it now makes perfect sense to call all pre-link functions in the original order again.

This also implies that a pre-link function of an element is run before any of its child elements pre-link functions as well, so for the sake of completeness:

**A pre-link function of an element is guaranteed to run before any pre-link orpost-link function of any of its child elements.**



Looking back

If we now look back at the original output, we can clearly recognize what is happening:

// HERE THE ELEMENTS ARE STILL THE ORIGINAL TEMPLATE ELEMENTS

// COMPILE PHASE

// levelOne: compile function is called on original DOM

// levelTwo: compile function is called on original DOM

// levelThree: compile function is called on original DOM

// AS OF HERE, THE ELEMENTS HAVE BEEN INSTANTIATED AND

// ARE BOUND TO A SCOPE

// (E.G. NG-REPEAT WOULD HAVE MULTIPLE INSTANCES)

// PRE-LINK PHASE

// levelOne: pre link function is called on element instance

// levelTwo: pre link function is called on element instance

// levelThree: pre link function is called on element instance

// POST-LINK PHASE (Notice the reverse order)

// levelThree: post link function is called on element instance

// levelTwo: post link function is called on element instance

// levelOne: post link function is called on element instance

**Summary**

In retrospect we can describe the different functions and their use cases as follows:

Compile function

Use the compile function to change the original DOM (template element) before AngularJS creates an instance of it and before a scope is created.

While there can be multiple element instances, there is only one template element. The ng-repeat directive is a perfect example of such a scenario. That makes the compile function the perfect place to make changes to the DOM that should be applied to all instances later on, because it will only be run once and thus greatly enhances performance if you are stamping out a lot of instances.

The template element and attributes are passed to the compile function as arguments, but no scope is available yet:

/\*\*

\* Compile function

\*

\* @param tElem - template element

\* @param tAttrs - attributes of the template element

\*/

function(tElem, tAttrs){

// ...

};

<http://jsfiddle.net/mrajcok/QkXcH/>

Pre-link function

Use the pre-link function to implement logic that runs when AngularJS has already compiled the child elements, but before any of the child element's post-link functions have been called.

The scope, instance element and instance attributes are passed to the pre-link function as arguments:

/\*\*

\* Pre-link function

\*

\* @param scope - scope associated with this istance

\* @param iElem - instance element

\* @param iAttrs - attributes of the instance element

\*/

function(scope, iElem, iAttrs){

// ...

};

Post-link function

Use the post-link function to execute logic, knowing that all child elements have been compiled and all pre-link and post-link functions of child elements have been executed.

This is the reason the post-link function is considered the ***safest*** and ***default*** place for your code.

The scope, instance element and instance attributes are passed to the post-link function as arguments:

/\*\*

\* Post-link function

\*

\* @param scope - scope associated with this istance

\* @param iElem - instance element

\* @param iAttrs - attributes of the instance element

\*/

function(scope, iElem, iAttrs){

// ...

};

**Conclusion**

By now you should hopefully have a clear understanding of the differences between thecompile, pre-link and post-link function inside directives.

If not and you are serious about AngularJS development, I would highly recommend reading the article again until you have a firm grasp of how it works.

Understanding this important concept will make it easier to understand how the native AngularJS directives work and how you can optimize your own custom directives.

And if you are still in doubt and have additional questions, please feel free to leave a comment below.

Have a great one!