Angular 2 Compiler Pipeline

**Attention: This is a public document!**

Github issue: [#3605](https://github.com/angular/angular/issues/3605)

Tobias Bosch ([tbosch@google.com](mailto:tbosch@google.com))

8/21/2015

[Purpose](#h.ex7vex8gfzmb)

[Non goals](#h.7xu4b3d9456d)

[Compile overview](#h.n0ibft6d3l4e)

[Read directive metadata](#h.m5bh9k0i74k)

[Load the template](#h.9eorvhro79x6)

[Parse the template](#h.6xv4fe6eebnf)

[Load and extract styles](#h.5qprkl5w7ape)

[Encapsulate styles](#h.hwsykzn2j9bo)

[Preanalyze the template of nested components for <ng-content>](#h.m3x2b47huy1k)

[Analyze the template for errors](#h.rmrq0ar81x1e)

[Create template commands](#h.umghnz20shuc)

[Create change detectors](#h.6krsgn9zo4jz)

[Create the template registry](#h.2v5vmm2zqro0)

[Public API for tools](#h.nlblu2nafuod)

[Runtime overview](#h.7ph4lwgfpqt4)

[PostCompiler](#h.u3kugjyl6qr0)

[ViewManager](#h.z0wqp2o8zh5j)

[Renderer](#h.j8wxchwr850i)

[SharedStylesHost](#h.238y2vkuph86)

# Purpose

This document is an overview of the compiler that integrates [the proposal for offline compilation](https://docs.google.com/document/d/11r8IuS4xDyhVSEBp7fDYo7aiLYsLEXKs4lPd36umUGM/edit#) to dramatically improve start up speed.

## Non goals

Let the template parser answer questions from editors and other tools like

* "what symbols can be completed at this position"
* "give me the changes since I last got the TemplateAst from you"

This is on purpose to be able to land the changes in this design doc in a reasonable amount of time and implement the performance benefits of the prototype compiler mentioned above.

# Compile overview



The Angular compiler can run as a build step but should also support running in the application itself during development.

## Read directive metadata

Creates an instance of DirectiveMetadata for every class that has an @Directive or @Component annotation with the corresponding data. The following code outlines the required data, split up into sub interfaces to show which part of the compile pipeline needs which parts of the metadata.

This is done in two different ways:

* in transformers / build step: use the appropriate mechanisms of the build environment to read annotations of source files
* compilation during runtime: use reflection (JS and Dart Dev mode only)

|  |
| --- |
| interface DirectiveMetadata {  }  interface TypeDirMeta extends DirectiveMetadata {  type: Type; // for compilation during runtime  typeName: string; // for codegen  typeImportPath: string; // for codegen and for relative paths  }  interface MatcherDirMeta extends DirectiveMetadata {  selector: string;  compileChildren: boolean;  }  interface ChangeDetectionDirMeta extends DirectiveMetadata {  events: string[];  properties: string[];  hostListeners: Map<string, string>;  hostProperties: Map<string, string>;  changeDetection: string;  }  interface TemplateDirMeta extends DirectiveMetadata {  viewEncapsulation: ViewEncapsulation;  html: string;  htmlUrl: string;  styles: string[];  styleUrls: string[];  }  enum ViewEncapsulation {  NONE; // just merge all styles  EMULATED; // emulated encapsulation by changing styles  NATIVE; // use a native ShadowRoot  } |

## Load the template

Loading a template is done using a ResourceLoader. It is important to keep the absolute url from which the template was loaded so that we can resolve relative paths within styles later on.

For inline templates (i.e. TemplatedirMeta.template), we use typeImportPath as the TemplateSource.templateAbsUrl.

For templates that are defined by an url themselves (i.e. TemplateDirMeta.templateUrl) we need to resolve relative template paths using typeImportPath and store the resolved absolute template url in TemplateSource.templateAbsUrl.

|  |
| --- |
| loadTemplate(loader:ResourceLoader,  directive:? extends TemplateDirMeta,TypeDirMeta):Promise<TemplateSource>;  interface ResourceLoader {  loadAsString(url:string):Promise<string>;  }  class TemplateSource {  template: string;  templateAbsUrl: string; // needed to later resolve urls in styles  } |

## Parse the template

|  |
| --- |
| parseTemplate(source:TemplateSource, directives:(MatcherDirMeta)[]):ParsedTemplate;  class Node {  sourceInfo: SourceInfo;  }  class SourceInfo {  start: number;  end: number;  url: string;  }  class ParsedTemplate extends Node {  rootNodes: Node[];  sourceUrl: string;  }  class EmbeddedTemplate extends Node {  index: number;  attrs: Map<string,string>;  variables: Map<string, string>;  bindings: Map<string,Expression>;  directive: MatcherDirMeta;  children: Node[];  }  class Attr extends Node {  name: string;  value: string;  }  class Variable extends Node {  name: string;  value: string;  }  class PropertyBinding extends Node {  name: string;  value: Expression;  }  class EventBinding extends Node {  name: string;  value: CallableExpression;  }  class Element extends Node {  name: string;  attrs: Attr[];  variables: Variable[];  props: PropertyBinding[];  events: EvenBinding[];  directives: MatcherDirMeta[];  children: Node[];  }  class NgContent extends Node {  select: string;  }  class InlineStyle extends Node {  content: string;  }  class UrlStyle extends Node {  // might be relative  url: string;  }  class Text extends Node {  value: string;  binding: Expression;  }  // the Nodes for expressions are omitted to keep this  // design doc small. See <https://github.com/angular/angular/blob/master/modules/angular2/src/change_detection/parser/ast.ts>  class Expression extends Node { … } |

## Load and extract styles

After a template has been parsed, all styles are loaded and extracted, including nested styles imported via @import statements in stylesheets. This will use the styles defined in the TemplateDirMeta as well as the styles defined in the template itself. The styles in the template will be removed from the parsed template.

|  |
| --- |
| loadAndExtractStyles(loader:ResourceLoader, component:TemplateDirMeta, parsedTemplate:ParsedTemplate): Promise<ParsedTemplateWithStyles>;  interface ResourceLoader {  loadAsString(url:string):Promise<string>;  }  class ParsedTemplateWithStyles {  template: ParsedTemplate;  styles: string[];  } |

## Encapsulate styles

Adds attributes \_ngcontent\_… and \_nghost\_... to elements and styles

Need TemplateDirMeta for the component of the template to read the view encapsulation as well as TypeDirMeta to get the type name

Needs TemplateDirMeta for all directives in the ParsedTemplate, if they are components

* to know which elements are components with ViewEncapsulation.EMULATED

|  |
| --- |
| encapsulateStyles(component:? extends TemplateDirMeta,TypeDirMeta, template:  ParsedTemplateWithStyles):ParsedTemplateWithStyles; |

## Preanalyze the template of nested components for <ng-content>

We need the NgContent.select for the templates of all nested components to precalculate content projection, i.e. for every light dom node find the index in the <ng-content>s of the component at this place. Note that via the commands we have a place to store this meta information also for text nodes (see later)!

* this requires to load the nested templates and to a quick scan for <ng-content> elements, but does not require to parse these nested templates
* I.e. to compile one template we need the templates of nested components, but not transitively all templates of all nested components.

|  |
| --- |
| findNgContentSelectors(loadedTemplate:LoadedTemplate):Map<DirectiveMetadata, string[]>; |

## Analyze the template for errors

Once the template is parsed, we can analyze it for errors.

E.g.

* detect property bindings to not existing properties on html elements
* detect expressions that use non existing properties of components

|  |
| --- |
| analyzeTemplate(component:DirectiveMetadata, template:ParsedTemplate):AnalyzeResult[];  class AnalyzeResult {  severity:Severity;  node: Node;  message: string;  }  enum Severity {  DEBUG;  INFO;  WARNING;  ERROR;  } |

## Create template commands

Template commands are used for rendering templates as well as for DI. We are creating an array of TemplateCmd for the ParsedTemplate as well as for each EmbeddedTemplate node.

Needs TemplateDirMeta and TypeDirMeta for all directives in the ParsedTemplate

* to know which elements are components with ViewEncapsulation.NATIVE
* to be able to reference directives in the generated code

Needs NgContent.select for the templates of all directives in the ParsedTemplate to to precalculate content projection

Template commands have the nice benefit that they are DOM independent and combine the UI with metadata together. This has the following benefits:

* create AppProtoViews lazily as the template can be used on the application layer as well (not just on the render layer)
* stamping out of ui elements from these commands is very efficient
* no need for ElementBinders (i.e. keep meta data separate to the html string)

|  |
| --- |
| createTemplateCommandsCodeGen(encapsulatedTemplate:ParsedTemplate,  nestedComponentsNgContent:Map<DirectiveMetadata, string[]>):string;  createTemplateCommandsRuntime(  encapsulatedTemplate:ParsedTemplate,  nestedComponentsNgContent:Map<DirectiveMetadata, string[]>)  :TemplateCmd[];  E.g.  result = [  BeginElementCmd(name, attrs, variables, …);  TextCmd('hello');  EndElementCmd();  BeginComponentCmd(templateId, …);  TextCmd('projected', ngContentIndex:0);  EndComponentCmd();  ]; |

Note: This will also calculate the index of the matching ng-content element for all light dom nodes of a component and associate this index to the template command. This speeds up view instantiation later on.

## Create change detectors

We are generating a separate change detector class for the ParsedTemplate as well as for each EmbeddedTemplate node

Needs ChangeDetectionDirMeta and TypeDirMeta for all directives in the ParsedTemplate

* to generate code that sets properties on directives
* to be able to reference directives in the generated code

|  |
| --- |
| createChangeDetectorsCodeGen(config:ChangeDetectorGenerationConfig, component:ComponentMetadata, parsedTemplate: ParsedTemplate): string;  // returns a factory function  createChangeDetectorsRuntime(config:ChangeDetectorGenerationConfig, component:ComponentMetadata, parsedTemplate: ParsedTemplate): Function; |

## Create the template registry

The template registry is the central place in the Angular runtime that contains all information about templates. If the compile pipeline runs as a build step, it needs to generate code that will instantiate a TemplateRegistry which will return PrecompiledTemplate instances.

|  |
| --- |
| class PrecompiledTemplate {  changeDetectorFactory: Function;  templateCmds: TemplateCmd[];  styles: string[];  }  class TemplateRegistry {  getTemplate(templateId:string):PrecompiledTemplate;  } |

## Public API for tools

|  |
| --- |
| compileComponentCodeGen(component:DirectiveMetadata, directives: DirectiveMetadata[], config:CompileConfig):Promise<string>;  compileHostComponentCodeGen(component:DirectiveMetadata, directives: DirectiveMetadata[], config:CompileConfig):Promise<string>;  compileComponentRuntime(component:DirectiveMetadata, directives: DirectiveMetadata[], config:CompileConfig):Promise<PrecompiledTemplate>;  compileHostComponentRuntime(component:DirectiveMetadata, directives: DirectiveMetadata[], config:CompileConfig):Promise<PrecompiledTemplate>; |

The codegen version will output a dart library / Typescript module that has a single export NG2\_TEMPLATE which contains a PrecompiledTemplate.

This allows to compile a component, given the metadata of its directives. We also need to load the templates of nested components, but don't need to parse them (only look for <ng-content> tags).

This allows to create an incremental compiler: if a template or directive is changed, we only need to recompile the component itself and all components that are using this component directly, but not transitively.

The compileHost… methods need to be called in addition to compileComponent… methods for all components that are used for bootstrap or together with the DynamicComponentLoader.

# Runtime overview



## PostCompiler

The PostCompiler takes in the template commands and creates the AppProtoViews. They contain information about bound elements, especially the ProtoInjectors and the ProtoChangeDetectors. The later is just read from the TemplateRegistry.

Note that the PostCompiler is lazy: it only creates AppProtoViews recursively until an ng-if / ng-for / … is encountered. For these it only creates the AppProtoView but not the contained ElementBinders. Whenever a new AppView for such a non initialized AppProtoView should be created, the PostCompiler continues creating AppProtoViews. This leads to better bootstrap performance as we only need to visit the parts of the application that are actually used.

## ViewManager

The ViewManager is the central entry point to instantiate a template. This is done by handing in an AppProtoView and returns AppViews. AppViews contain injectors which contain the instantiated directives as well as the instantiated change detectors.

## Renderer

The renderer stores the template commands in a RenderProtoView so that they don't have to be passed to the Renderer for every view instantiation (important e.g. for WebWorkers).

To create DOM elements, the Renderer just executes the instructions in the TemplateCommands. Note that the TemplateCommands are renderer independent and can be used to also create elements for NativeScript or ReactNative.

During view instantiation, the Renderer also does content projection. This is facilitated by associating indices of the corresponding ng-content element to the TemplateCommands in the ligth dom of components during the compilation phase.

## SharedStylesHost

The SharedStylesHost tracks the styles of all components in Angular. Note: This does not include the styles of components that use native shadow dom.

By default, the SharedStylesHost adds these styles to the <head> element and to all shadow roots of Angular components that use ViewEncapsulation.NATIVE.