Angular 2

An architectural overview

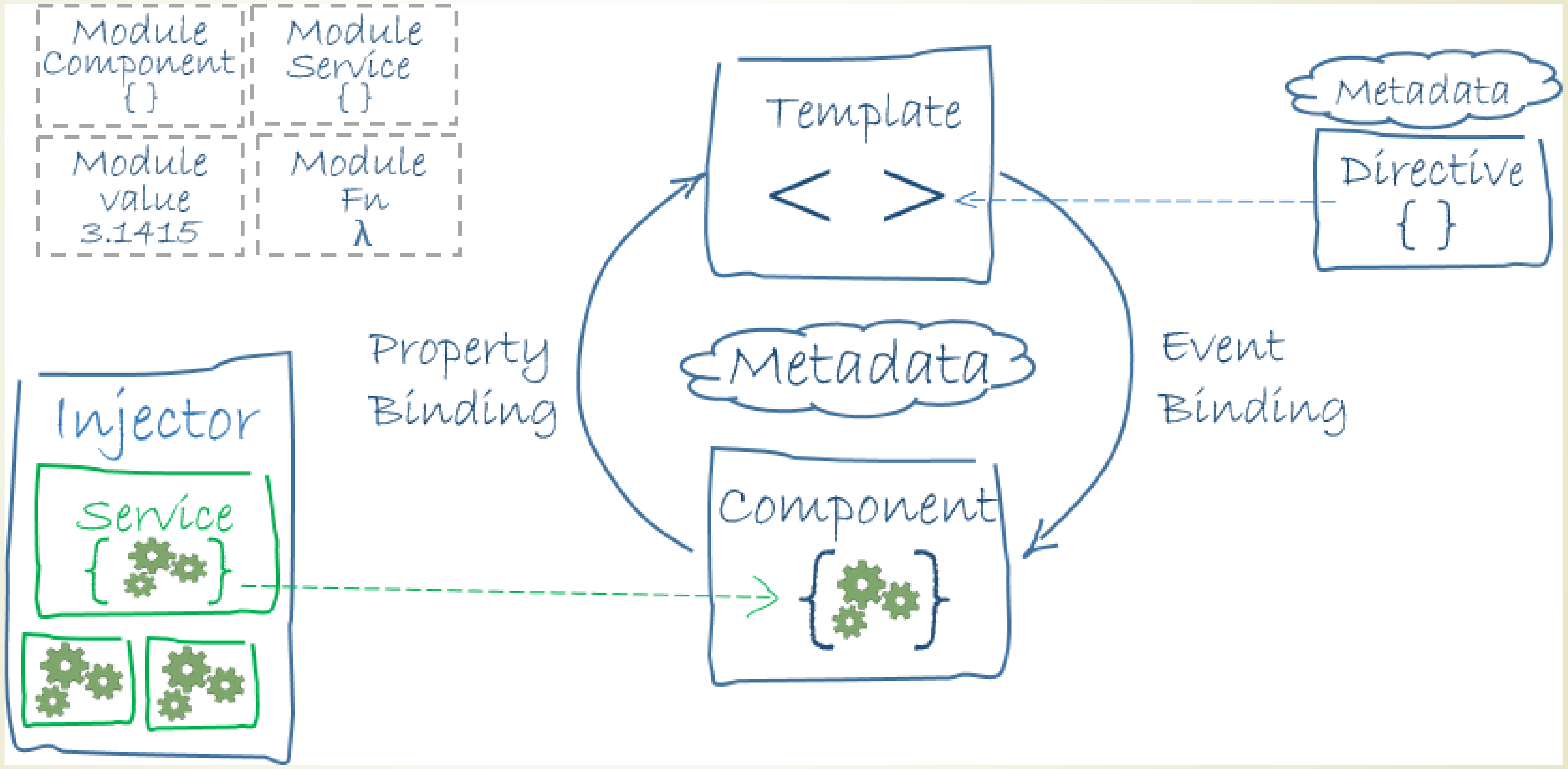
Background

- SPA is the trend and JavaScript is the language of choice
  - SPAs are more closer to desktop app than traditional web app
- Many front-end JavaScript frameworks to choose from
  - Angular (1 vs 2)
  - React
  - Ember
  - Backbone
  - and more …
- Angular 1.x is already a very popular framework
- Angular 2 improved on that functionality and made it faster, more scalable and more modern (Component Based)
- Angular 2 is build on five years of community feedback.
- Angular 2 is not backward compatible (it is built ground up)

Architecture Overview

- MVC? or MVVM? MVW (Whatever)
- Framework consists of several libraries, some core and some optional
- You write angular application by
  - composing HTML templates with Angularized markup
  - writing **component** class to manage those templates
  - adding application(wide) logic in services
  - and boxing components and services in modules (NgModule)
- Then you launch the app by bootstrapping the root module
- Of course, there is more to it than this
- For now, focus on the big picture on next slide

Architecture Overview



Building blocks of an Angular 2 app

- The architecture diagram identifies the eight main building blocks of an Angular 2 application:
  - Modules
  - Components
  - Templates
  - Metadata
  - Data binding
  - Directives
  - Services
  - Dependency Injection

Modules

- Angular has its own modularity system called Angular modules or NgModules.
- Every Angular app has at least one module, conventionally named AppModule.
- Most apps have many more *feature module*.
- An angular module, weather a root or feature module, is a class with an `@NgModule` decorator
  - Decorators are functions that modify decorated JavaScript class in some way
  - Angular has many decorators that attach metadata to classes (we will see soon)
- NgModule is a decorator function that takes a single metadata object whose properties describe the module.
  - declarations – the view classes that belongs to this module
    - components, directives and pipes
  - exports – (the subset of declarations) that should be visible and usable in the component templates of other modules

Modules cont…

- imports – (other modules) whose exported classes are needed by component templates declared in this module
- providers- creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app
- bootstrap – the main application view, called the root component
- see example module
- Launch an application by bootstrapping its root module
  - `AppModule` in a `main.ts` file like this one
- JavaScript also has its own module system. It's completely different and unrelated to the Angular module system
  - In JavaScript each file is a module and all objects in the file belong to that module
- There are two different and complementary module systems.

Library modules

- Angular ships as a collection of JavaScript modules
  - you can think of them as library modules
- Each Angular library name begins with the `@angular` prefix
- You install them with `npm` package manager and import parts of them with JavaScript import statements

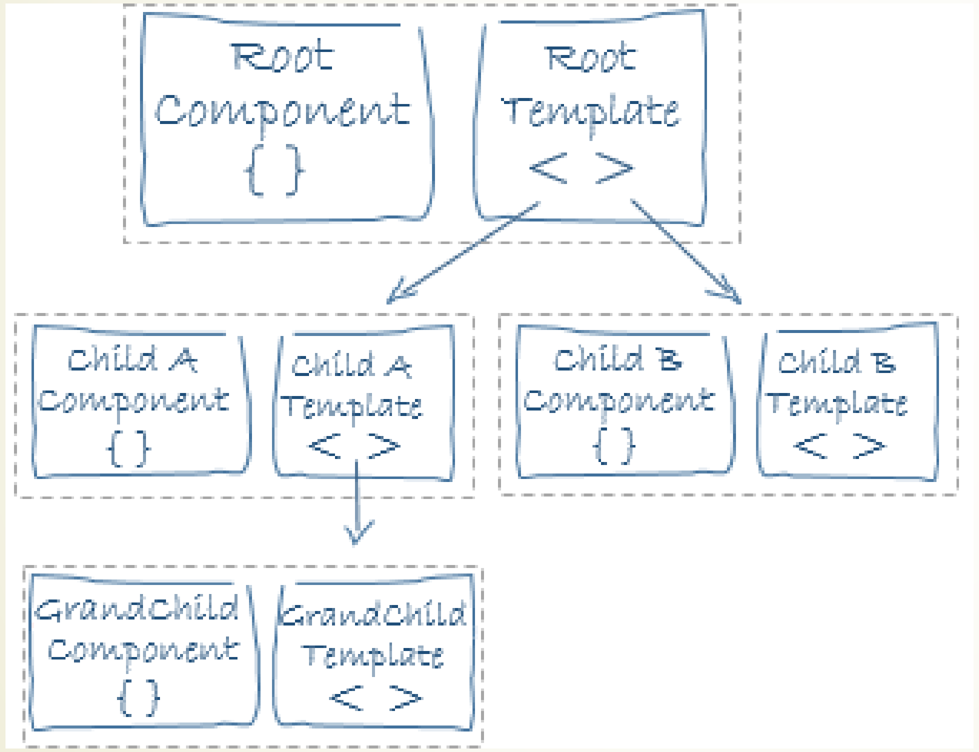
Components

- A component controls a patch of screen called a view
- Define a component's application (view) logic inside a class
- The class interacts the view through API of properties and methods
- Angular creates, updates, and destroys components as the user moves through the application.
- Your app can take action at each moment in this lifecycle through optional lifecycle hooks, like `ngOnInit()` (see example)

Templates

- You define a component's view with its companion template
- A template looks like regular HTML, with few differences
- Here is a template for products component
  - Although this template uses typical HTML elements like `<h2>` and `<p>`, it also has some differences (we will learn more about template syntax later)
  - In the last line of the template, the `<product-detail>` tag is custom element that represents a new component, `ProductDetialComponent`
  - `ProductDetialCompoent` is a child of the `ProductsComponent`
  - See component tree diagram on next slide
- Notice how `<product-detail>` rests comfortably among native HTML elements.
- Custom components mix seamlessly with native HTML in the same layout.

Component tree

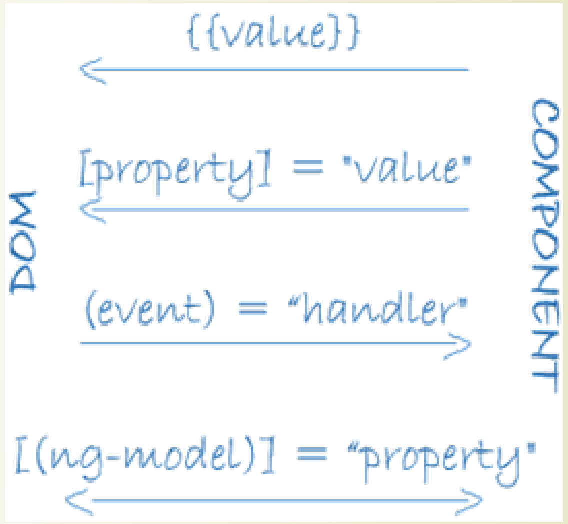


Metadata

- Metadata tells Angular how to process a class
- The `ProductsComponet` we saw earlier is just a class
  - It's not a component until we *tell angular* about it.
- To tell Angular that it is a component, attach metadata to the class.
- In TypeScript, we attach meta data using decorator
- Here is some metadata for `ProductsComponent`.
  - `@Component` decorator takes a required configuration object with the information Angular needs to create and present the component and its view
  - Few of the possible `@Component` configuration options
    - selector – CSS selector
    - templateUrl – path to component's HTML template
    - directives – array of the components of directives this template requires
    - providers – array of dependency injection providers for services that the component requires

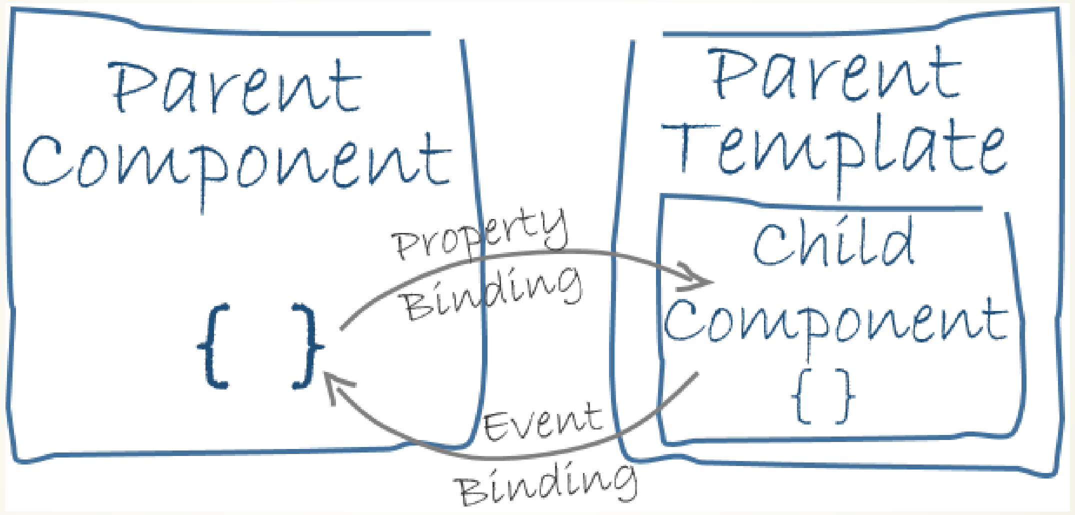
Data binding

- A mechanism for coordinating parts of a template with parts of component
- Add binding markup to the template HTML to tell Angular how to connect both sides
- There are four forms of data binding
  - interpolation
  - property binding
  - event binding
  - two-way data binding
    - combines property and event binding in a single notation
- Angular processes all data binding once per JavaScript event cycle
  - starting from the root of the application component tree through all child components
- Data binding plays important role in communication between a template and its component.



Data binding

- Data binding is also important for communication between parent and child components



Directives

- Angular templates are dynamic
  - When Angular renders them, it transform the DOM according to the instructions given by **directives**.
- A directive is a class with directive metadata.
  - In Typescript, apply `@Directive` decorator to attach metadata to the class.
- A component is a *directive-with-a-template*
  - `@Component` decorator is actually a `@Directive` decorator extended with template-oriented features
- Two other kinds of directives exist
  - structural directive (fro e.g. *ngFor, *ngIf, ngSwitch)
  - attribute directive (for e.g. ngModel, ngStyle, ngClass)
- Both of above directives tend to appear within an element tag as attributes
  - Structural directives alter layout by adding, removing, and replacing elements in DOM
  - Attribute directives alter the appearance or behavior of an existing element.

Services

- *Service* is a broad category encompassing any value, function, or feature that your application needs.
- Almost anything can be a service
- A service is typically a class with a narrow, well-defined purpose
  - It should do something specific and do it well. (e.g. logging service, data service)

- Angular has no definition of a service
  - There is no service base class, and no place to register a service
- Yet services are fundamental of Angular application
  - Components are big consumers of services
  - Here's is an example of a service that logs to the browser console
  - Here's is ProductService that fetches products
- Services are the ones to handle the server communication grunt work

Services are everywhere

- Component classes should be lean
  - They don't (shouldn't)
    - fetch data from the server
    - validate user input
    - or log directly to console
  - They delegate such task to services
- A components' job is to enable the user experience and nothing more.
  - It mediates between the view (rendered by template) and the application logic (which often includes some notion of a *model*)
- Angular doesn't *enforce* these principles
  - It won't complain if you write "kitchen sink" component with 3000 lines
  - Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through *dependency injection*.

Dependency injection

- *Dependency injection* is a way to supply a new instance of class with fully-formed dependencies it requires.
  - Most dependencies are services.
  - Angular uses DI to provide new components with services they need
- Angular can tell which services a component needs by looking at the types of its constructor parameters.
  - For e.g., the constructor of `ProductListComponent` needs a `ProductService`
    - `constructor(private service: ProductService){}`
  - When Angular creates a component, it first asks an **injector** for the services that the component requires.
- An injector maintains a container of service instances that it has previously created.
  - If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular.

Dependency injection

- But how does injector know to make a service instance?
  - In brief, we must register a **provider** of the service with the injector.
- A provider is something that can create or return a service, typically the service class itself
  - in other words, a *provider* is a recipe for creating a service
- You can register providers in modules or in components
  - In general, add providers to the root module so that the same instance of a service is available everywhere.
  - Alternatively, register at a component level in the `providers` of a `@Component` metadata
    - Registering at a component level means you get a new instance of the service with each new instance of the component,

Main points

- Modules, Components, Templates, Metadata, Data Binding, Directives, Services and Dependency Injection, these are the eight main building blocks of an Angular application.
- Module encapsulates related components, directives and pipes and also controls the access of these components from outside.
- Component encapsulate view and logic together with template and component class, where @Component decorator and its metadata makes the job easy (declarative)
- Data binding makes communication possible between view and logic within a component and also between parent-child components.
- Directive either change the DOM structure or alters the appearance or behavior when applied to regular HTML tags or the custom angular tags.
- DI seamlessly provides all the required services for a component in a declarative way.