Angular Boot Camp

Curriculum Notes

Oasis Digital Solutions Inc.

Updated January 2020

Welcome to the Angular Boot Camp curriculum notes. These notes serve to supplement and outline the information presented in class.

Because of the extensive experience of our instructors, a printed curriculum like this does not fully capture the important information presenters can offer, particularly in response to student questions.

# Introduction

## Angular Boot Camp

### Class objectives and roadmap

Students will learn the Angular basics, while building an Angular application. We believe this curriculum covers the basics well, and positions students to use Angular effectively and efficiently.

Our philosophy:

* Introduce one topic at a time.
* Learn to combine them by building an application.
* Share experiences from building real software.
* Encourage and answer myriad questions.

### Oasis Digital background

“Developers who sometimes train”: We are developers who also train, not isolated-from-the-real-world trainers.

### Student experience and expectations

Ask students:

* Background?
* What have they done, what are they doing?
* What do they need to be able to do?
* How much JS, frontend dev work, etc?
* Server-side experience?
* Experience with Angular? Gone through tutorial?

## Preparation

### Prerequisites review

Students should have experience with HTML, JavaScript, and jQuery or the DOM manipulation in general, perhaps by having worked in server- side web development (in Java, JSP, PHP, ASP, etc.) in the past.

Students should have:

* A laptop (PC, Linux, or Mac)
* Chrome web browser. Angular works well with all major browsers, but there are extra tools for Chrome during development.
* Text editor (JavaScript-aware) or IDE, such as VS Code.
* We recommend Visual Studio Code.
* Ability to install software, including Node, NPM, and packages.
* Optional: experience with AngularJS 1.x.

### What is Angular?

Angular is for:

* Web “SPAs” - Single-Page Applications
* Desktop HTML hybrid applications (Electron)
* Mobile HTML hybrid applications (Ionic, others)
* Desktop Native - coming soon?
* Mobile Native (NativeScript)
* Progressive mobile web apps (PWAs)
* Server-side pre-rendered applications (Universal)
* More?

### Brief comparison to AngularJS 1.x

* Performance, and design changes needed to achieve it
* Accommodate the changing web development environment, including Web Components
* Embrace and benefit from JavaScript progress
* Replace AngularJS-specific features with newer, standardized features
* Dynamic/lazy loading
* Ease of use
* Scale up to complex applications
* Competition!

### Single page apps vs server-side web applications

Comparison between SPA and conventional web apps. Load a page once, then most of your app happens without “reloading.”

SPA doesn’t actually have to mean “single page” – can often be multiple SPAs working together.

“Anyone done SPA?”

### Why Angular?

How Oasis Digital got to Angular: around 2011 we evaluated numerous SPA frameworks, and have been deep in to Angular and SPA development since then. Key points for Angular:

* Testability
* Observables, amenable to Redux architecture
* Declarative: HTML as DSL
* Supported by Google, momentum because Google is pouring money into it.

Likely to be around, have some life. Significant enterprise adoption. Reasonably safe choice, certainly as much as any other choice in frontend development.

All MIT licensed. Very permissive license, no scary “viral” GPL, so okay to use in proprietary software.

Not a Google product, but Google has a large team (30+) full-timers at Google working on Angular.

Links:

### Where Angular fits

Good:

* CRUD apps
* Complex business interactions, far beyond CRUD
* Lots of screens
* Big things
* SEO (getting better, due to newer tools)
* Mobile

Not so good:

* Small things

Iffy:

* Graphics-heavy games (though that’s not stopping some people)

Great for apps that live behind a login screen, but can also be used for public-facing websites.

Good for interactive portions of larger sites.

“Made with Angular” page <https://www.madewithangular.com/> demonstrates that Angular is widely used, applicable to many different types of applications. > abc

# Starting with Angular

## JavaScript, TypeScript, Angular

Instructor - consider drawing a grid on the whiteboard.

For classes interested in the differences between ES5, ES2015, TypeScript, ES.next, draw a grid on the board containing categories for each technology in the stack.

### ES5

Historically this was the JavaScript supported by browsers; but today most current browser support at least ES2015.

### ES2015

### ES.next

### TypeScript

Along the way of teaching Angular, we will also teach “just enough TypeScript” to understand and work on the code.

Since Angular itself is coded in TypeScript, and since TypeScript appears to be most prominently featured in the examples, we believe that the great majority of Angular development will happen in TypeScript. Therefore we also use it here, for the remainder of the class.

We have been using TypeScript even before Angular at Oasis Digital, and are very happy with the development process benefits. We even write most Node code with TypeScript.

## Simplest Angular application

Here is close to the simplest Angular app - just a few libraries and a bit of JS code.

### Appearance of the in-class examples - CSS

In this instruction kit, we are using a “class-light” CSS framework that improves the appearance of HTML pages and elements without requiring significant changes to the HTML code itself.

Although we often use Angular Material in our own projects, we intentionally don’t use it here, as it can be difficult to understand what aspects are Angular versus Angular Material when first learning Angular.

### Development tooling for class use

You might be curious what is inside the file abc2.js and the related files in the minimal\_lib directory. Sometimes in class we colloquially describe it as “magic”, but actually it is a combination and use of these technologies:

* TypeScript compiler
* SystemJS
* Angular
* RxJS
* NgRx
* Lodash
* minor libraries and miscellaneous pieces

abc2.js is most similar to the configuration provided by the “quick start” described and used and linked from Angular.io. It serves the needs in class quite well, because it minimizes any need for server-side tooling while we all make it through Angular itself.

Lodash tips:

npm install lodash-es @types/lodash-es

import { whatever } from 'lodash-es';

This enables use of the ES6 module packaging of lodash, for effective tree shaking during bundling.

## The TypeScript module system

### Dividing a program into modules / files

* TypeScript Modules, which are files
* Imports from the same project
* Imports from libraries
* Exports > abc

## Templates and bindings

### Template binding expressions

{{variable}} works as with AngularJS.

[prop] is a one-way binding to the named DOM property.

Specifically, this means a (“JavaScript”) property, not an HTML attribute.

Compared to other common syntaxes used by other libraries, this is:

* short
* stands out more

### Null coalescing / safe traversal

By default, an expression like this will fail if x is null/undefined:

{{ x.y }}

To avoid this, the following operator performs automatic null coalescing:

{{ x?.y }}

#### textContent vs innerText

[textContent] is preferable for performance reasons, according to many online sources.

#### TypeScript

In this chapter, students are first exposed to various TypeScript constructs, such as fields and methods. Take time to explain them. > abc

### Events and event bindings

This shows several ways to bind between controls.

Key ideas:

refs: #name or var-name. Either way it makes a new variable. If used in a string, it is making a new JS variable for repeater or similar. If used in HTML, it makes a new variable whose value is the DOM element it was used on.

No such thing as ng-click etc. anymore. One generic replacement to numerous specific things. In retrospect, it is so obvious that one wonders why they ever did the old thing.

In the template you can pass $event to get the raw event.

The important thing about the alternate syntax is that it lets you use things like Jade, and the “data-” is valid in HTML5.

#### TypeScript

The type that appears in a function definition parameter list is checked statically, but not dynamically. > abc

## Multiple Components

Any nontrivial Angular application will be composed of many components. Components are the primary chunk out of which an application is constructed, so it’s very important to become comfortable composing them together.

### Using inspector tooling to understand the component hierarchy

As your application has more and more components, understanding how they are nested together can grow more difficult. To help with that problem and assist with many other areas of analysis and debugging, consider installing the Angular DevTools plug-in:

https://angular.io/guide/devtools

abc

## Lazy loading

### Routes as the unit of lazy loading

#### Why?

TODO: Update this wording to reflect LL as default loading approach

The purpose of lazy loading is to defer loading the source code for portions of an application until they are needed. It makes little difference on a small application, but makes a very big difference on a large application.

For optimal user experience, lazy load nontrivial subsections of an application - by default. Lazy loading works well enough that you can “build it right the first time”.

#### How?

This example shows the particular variation of routing syntax needed to work with the CLI lazy loading support. The syntax can be slightly different if you are using a different tooling approach.

### Preloading: eager lazy routes

Initially, lazy loading is often perceived as a feature only for the largest, most complex applications. However, it may be best seen as a default way of loading portions of an application, suitable for nearly all applications.

To understand the benefit, consider the additional feature of pre-loading: with this setting, Angular will begin (asynchronously) loading modules configured for lazy loading, even before the user needs them. The sequence of operations is then:

* Load the page, with the minimum of code needed to start.
* Render the page, so the user can see and interact with it.
* Asynchronously, while no user is waiting, load other code that might be needed.
* When the user needs more of that code, when they navigate to such a route, it is immediately ready.

For a deeper explanation, see this blog post by a core Angular team member:

https://vsavkin.com/angular-router-preloading-modules-ba3c75e424cb#.fay3a7ak5

#### Lazy loading during class

Although most of our in-class example don’t lazy load, for workshop code (and for real work) we recommend lazy loading. That way, you get practice with the details. > abc

# Building with Components

## Development tooling

### Angular CLI - the powerful default choice

Angular CLI is the official, Angular core team endorsed mechanism for building projects.

The CLI provides an army of benefits. Most of these benefits can be achieved from other build systems, however it would require developers to be intimately familiar with the inner workings of JavaScript module loading, build tool chains, cache busting, asset management, deployment environment setup, etc.

### Webpack - for custom builds

Many of the Angular seed projects use Webpack, with good results. However, it is surprisingly tedious to configure this toolset for a project from scratch, and even with a seed project there is extensive complexity sitting in your project. For an easier start…

### SystemJS - for browser-only development

The Angular web based examples (on the Angular website, using Plunker, etc.) as well as the development tooling we use during class, use the System module loader and associated tooling.

## Starting a project with Angular CLI

Angular CLI Homepage: <https://cli.angular.io/>

Angular CLI Github Project: <https://github.com/angular/angular-cli> > abc

### Creating your component hierarchy

The purpose of this step is to show how easily students can get up and running with a realistic component hierarchy. To do so, focus primarily on the nesting of the components, not on their contents.

Continue with the nesting of components, but we go beyond just putting a couple of them together and start stacking the multiple components deep.

Students might initially find that this nesting complexity seems overwhelming. But that is why it is so important to become very comfortable with the syntax and semantics. In a real application you will have many more components than this.

### Visual wireframing

Angular components can be used as a skeleton on which to build your applicant’s visuals; and we recommend this for real projects. Often a surprisingly large amount of progress can be made, and understanding can be gained, using just components with HTML and CSS in them. > abc

## Essential built-in directives

### Branching with \*ngIf

### Iteration with \*ngFor

abc

## Pipes

### Using the built-in pipes

Pipes built in to Angular: <https://angular.io/api?type=pipe>

Using default/built in pipes

Chaining pipes

If AngularJS developers ask what happened to filter and orderBy, refer them to this explanation in the Angular docs:

<https://angular.io/guide/pipes#appendix-no-filterpipe-or-orderbypipe>:

Angular doesn’t provide pipes for filtering or sorting lists. Developers familiar with AngularJS know these as filter and orderBy. There are no equivalents in Angular.

This isn’t an oversight. Angular doesn’t offer such pipes because they perform poorly and prevent aggressive minification.

### When are pipes the right solution?

We generally recommend sparing use of Pipes - most of the time it makes more sense to manipulate data in to a form suitable for display before it reaches your Angular template. This enables more rapid development and easy unit testing. > abc

## Pipe Locales

Internationalization (i18n) is a large topic with many parts, but one part that can be implemented relatively simply is adding locales to the built-in pipes.

DecimalPipe, CurrencyPipe, PercentPipe, and DatePipe are all localized. This means that they will appear in the format appropriate for en-US (American English), but that this is overridable.

In an NgModule, we can use Providers to change the default locale from en-US to another locale that the pipes will use instead. We can also change the default currency from USD to another currency that the CurrencyPipe will use instead.

Additionally, the locale can be changed in the template separately from the app/feature default.

In order for either type of locale definition to work (Provider-level or point-of-use), the locales must be registered with registerLocalData() - see app.module.ts for this in action.

Note: Providers are a part of Angular’s Dependency Injection system. We will learn more about Dependency Injection later. > abc

## Class and style bindings

### [class.x] bindings

[class.fat-border]="n > 7"

This means put the class on the element IFF expression is truthy

This is a shortcut built in to Angular; not a DOM property.

### [style.y] bindings

[style.width.px]

This is also an Angularism, not a bare DOM property.

### [ngClass] for more complex class selection

abc

## Angular Animations

### Angular animation introduction

Must install @angular/animations to use

Import BrowserAnimationsModule and any animation functions

animations metadata property goes inside @Component decorator

trigger: defines the trigger word used to call an animation and the transitions that occur in an animation transition: defines the states which the animations occur between and those animations keyframes: identifies the styles which are used during an animation; offset defines how much time between styles

### Why Angular animations instead of CSS Transitions?

Some actions are made trivial with Angular animations (ie: keyframes) Adds simplicity to some animations Easy access to reusable animations Declarative API Built directly into Angular itself

abc

## Component data binding with @Input()

abc

## Component event binding with @Output()

abc

# Services, reactivity, and HTTP

## Dependency injection

### Introducing injection with HTTP

The tough part of this is explaining why dependency injection is a good idea. People who have used platforms that offer it generally arrive eager and ready to proceed. Those who have never heard of it are generally skeptical. They wonder, why should I do this, why should I care about this? That is the hardest thing to explain in class quickly.

### Why dependency injection?

* So that someone else writes the code to instantiate all the different pieces of an application and wire it up.
* So that dependencies are more easily understood and edited across large teams with fewer conflicts.
* So you can swap out different dependencies during testing.

### What can be injected?

The Angular dependency injection system has many services available out-of-the-box. You can consult the documentation to read about numerous built-in services. In this example, we use the built-in HttpClient service to load the list of employees from the JSON Server API.

Incidentally, parent components are automatically available for injection. (This should be described elsewhere; it is not a very good introductory topic.) > abc

## Creating services

Dependency Injection

* Annotations and Metadata
* Nested injectors
* Scope / lifetime
* Asynchronous / Lazy injection
* Nested DI as a replacement for nested scopes

https://angular.io/guide/dependency-injection

Point out something about how this would work in ES5 or ES6.

.NET DI Examples: - http://www.ninject.org/ - https://blogs.msdn.microsoft.com/webdev/2014/06/17/dependency-injection-in-asp-net-vnext/

### Making a service available for dependency injection

Mark it @Injectable(), though this is not always necessary.

### Inject HTTP into a service instead of a component

This is a usually a better practice than directly using HTTP from components. Among other reasons, this reduces the need for (tedious) testing of components, shifting the testing to (easier) services.

(For those who remember Angular JS injection, TypeSctipt metadata removes the need for positional injection.) > abc

## Observables

We continue to refine the application, this time by hiding the details of the Response object inside the service. That means using a .map() to handle the parsing of the JSON inside the Response object, so the subscribe in the component just gets data as its payload.

There’s still that annoying .subscribe() in the component, though. We’ll deal with it next.

### The least you need to know about Observables

Although rarely recommended, it is possible to build a high quality, large Angular application with minimal use of Observables. Thus the least you truly need to know is that an Observable can, in simple cases, work like a Promise, and be easily converted to a Promise.

### Observables as the pipeline of data flow in Angular.

abc

## The async pipe

### Observables as a source of tedium and boilerplate

Used naively, it is easy to end up with a lot of boilerplate code subscribing and unsubscribing to observables.

### Automatic subscription and unsubscription

The async pipe automatically subscribes and unsubscribes. We find in substantial, complex projects that nearly all manual subscriptions can be replaced with this mechanism. > abc

# Component-local State

Component local state is the first of three parts in our state management series. This is the core of building with Angular, stacking together components into routes, and as much as possible using the built-in facilities to manage their state. Then adding your own code for additional state beyond which can be managed in the box.

Stepping aside from observables briefly, let’s look back to what we saw when we first put data fields into components: simply keeping the state of a component, in its fields, then passing that state up or down using bindings and events.

This approach is actually quite workable in many situations, and is a good default starting point until a more complex needs arrives.

## Suitable uses for component-local state

To help understand that boundary, study the “do” and “don’t” examples here. See that in the positive example, the state is contained quite locally and passed up and down only a little bit.

## Where component-local state becomes problematic

In the negative example though, the state is passed up and down more extensively, treating the undesirable “bucket brigade”. When you see this, the code is looking for a more comprehensive state solution, which we will see later on. > abc

## Reactive forms

It’s easy to initially think of forms as an occasional feature; maybe your application has a form or two. But we have found that enterprise applications typically have numerous forms, and even other kinds of applications often have screens which, upon careful consideration, are best developed as a form even if they don’t look subjectively like a form.

### Control groups

Controls can be grouped in to control groups, hierarchically. A form is a control group, but not all control groups are forms.

### FormBuilder

FormBuilder is a high level, concise API to construct form groups etc.

The documentation is less bad now, and incidentally Cory recently published a nice easy example:

https://coryrylan.com/blog/angular-2-form-builder-and-validation-management

https://medium.com/@jecelynyeen/building-model-driven-form-with-latest-technique-in-angular-2-rc-4-b825ef780b8e#.ko656r30x

### ngSubmit

* Always use ngSubmit
* It prevents submitting the form as HTML form – which will happen if you use a naïve (submit) event talk and your code errors.

### invalid vs !valid

Defensively use !control.valid vs control.invalid for functionality like disabling buttons. This is due to the possible asynchronous nature of these boolean properties; so, if a validator is ‘pending’ it will enable the button which would be unintended. If the button relies on ‘!valid’ then it will stay disabled until the form is valid (and/or not pending a validation). > abc

## Nested reactive forms

Reactive Forms are useful for more than single-screen forms; they are also very powerful state management tools by themselves. For example, a wizard-like app might use one FormGroup to contain the state of the entire application, with a separate sub group for each page within the wizard.

With a state design like this, the Form Group itself provides all the state management and reactive hooks that an Angular application would need. > abc

## The smart/display component pattern

### Motivations

### Implementation

Async pipe plus smart/display is nice.

For the moment, we recommend caution with this pattern, because it means you make much more extensive use of untyped and unchecked data binding. > abc

## Observable composing and chaining

### Interactive search with observability

Observable operators, and data flow are not ancillary, advanced features of Angular. Rather, they are right at the heart of the tools to use frequently and regularly to create typical applications greens.

Here we introduce various additional operators, and show how they can be composed to form a realistic screen design, reminiscent of a search screen that might appear in a typical enterprise application.

(Unlike such an application, this one is especially well-suited to displaying cat pictures, one of the fundamental purposes of the Internet.)

In this code, various old and new operators are combined to call an API in an effective way. It works roughly as follows:

Set up a form control for each control on the screen.

Consider first the subreddit control:

* Get the observable stream of changes to the value.
* Prepend the initial value, because we want the search screen to work even before the user interacts with this control.
* Trim any text at the beginning and end of the screen, because we don’t expect the server to accept such text or for it to mean anything.
* Filter out any empty values, because the search should never run for empty values.

Consider second, the search criteria control. Treat it the same as above.

Next, combine these two observable streams, into a criteria object with corresponding fields.

Finally, that criteria can be used to run the search:

* Debounce so that we don’t run searches too frequently while user is typing.
* SwitchMap, because we need to call an observable service inside our observable chain.
* In the result, always start each search result with an empty array, so that the old search results disappear while the new search is running.
* Delegate the details of the actual search to service code.
* Retry the service up to three times if it fails.

This code shows a good example of dividing functionality between user interface and service. The component contains observable manipulation code that has to do with the behavior of the user interface. But the code that actually calls the backend and understands HTTP etc, is in a service class.

This example can also be used to demonstrate cancelable HTTP. To do so, comment out the “debounceTime”, then quickly typed several letters into the search criteria. Study the network tab of the developer controls to see that obsolete search HTTP calls have been canceled at the browser/network level.

This post contains a fairly good explanation of the switchMap operator and when to use it:

http://shuheikagawa.com/blog/2016/05/01/incremental-search-with-rxjs/

To understand observable operators, RxMarbles can help:

RxMarbles.org

Excellent, long introduction to observables:

https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

## Which Map Operator?

In this step we introduce switchMap. There is a lot of information around the Internet suggesting different RxJS map operators. Here is a concise summary of one to use which kind of map operator.

### Loading data to the same display area

This is the most common scenario, and usually encountered first. A user makes choices about data they wish to see, and as they make those choices, a backend operation is executed. The results of that operation are then displayed, with each new improved display data replacing the previous display data. That is what is happening in this example.

For the scenario, switchMap is the right choice. It cancels any prior operation in progress, when a new operation starts.

### Saving data, when the save order doesn’t matter

The second most common type of operation involved saved data. We don’t have an example included here of saving data, but the observable data flow for saving is similar to that for loading.

With saving, switchMap is almost always wrong - because starting to save a second piece of data should never cancel the saving of the first piece of data. Instead, mergeMap usually makes sense, and it allows multiple save operations to proceed concurrently.

### Saving data, when the save order DOES matter

Lastly, sometimes the order of data save matters; perhaps there is some data dependency between each piece of data as it is added. For these cases, the desired behavior is for the client application to complete each save operation before proceeding with the next.

For this, use concatMap. It allows each underlying observable operation to complete before proceeding with the next.

#### Future Considerations

http://api.duckduckgo.com/?q=DuckDuckGo&format=json // This has a nicer api but requires jsonp requests // This means we would have to use this instead of http in the example https://angular.io/docs/js/latest/api/http/JSONP\_PROVIDERS-let.html

More good stuff:

http://blog.jhades.org/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/

https://coryrylan.com/blog/angular-2-observable-data-services

Is this a good time to mention broader reactive programming question?

https://vsavkin.com/the-taxonomy-of-reactive-programming-d40e2e23dee4 > abc

## Making User Actions Observable

While the previous step showed how to use APIs that return Observables already, this step shows us how to handle user events that are not given to us as Observables.

As the user is clicking employee names, it is important that we publish those events into an observable Subject instead of handling them directly. This prevents results from arriving out of order if the user clicks around. > abc

## Filtered list via Observables

While the previous step show just a single layer of user interaction causing server-side data loading, this step now shows two layers, more typical of real enterprise application.

Both the left and right portions of the screen are populated with data from a server-side API; unlike much of what we have seen to this point, the data is not mocked nor preloaded in the user interface code.

Important features of this realistic example:

* As you change the employee name filter, changes are de-bounced to avoid excessive server requests.
* Requests to load data from the server are automatically kept in order by the observable stream mechanism.
* You can experiment with variable API latency (jitter) in employee-loader.ts to see this happen.
* As you click on an employee, their details are fetched; these requests are also kept in order similarly. > abc

## Reactive forms with control fields

This work-in-progress shows a different approach, in which you keep a reference to each form control is a component field. It shortens the syntax in both the component and the form, although it also makes the form slightly less idiomatic to construct.

There is an open question about which approach we as a group actually recommend.

We initially thought this approach might be more amenable to type checking with AOT. But AOT type checks either way. > abc

# Router State

Router state is the second of three parts in our state management series. This is the core of building with angular, stacking together components into routes, and is much as possible using the built-in facilities to manage their state. Then adding your own code for additional state beyond which can be managed in the box.

## Route as navigational state

In this example, the use of nested routes for different views of the data illustrate the idea that the route is the Angular provided tool for capturing this aspect of navigational state. Using the router for this is much superior to an ad hoc bit of state for this view selection:

* Better UI/UX
* More idiomatic
* More concise

### Nested routes

Nested routes are simply additional layers of the same routing technology. In the example here they are nested only 2 deep, but in a real application of scale can easily be deeper.

### Routing to components

For those coming from the older AngularJS, the idea of a route destination as its own top level entity isn’t around anymore; instead you always route to a component (either directly, or as we saw earlier, via a lazy loaded module).

### Configuring the router

For a basic explanation of pathMatch: 'full', see <https://angular.io/guide/router#redirecting-routes>.

For a lot more, see <http://vsavkin.tumblr.com/post/146722301646/angular-router-empty-paths-componentless-routes>.

### HTML5 vs hash routes

New projects should use native HTML 5 routes; but hash-based routes are still supported for applications which much exist in a server or integration environment in which the necessary server-side support for HTML5 routes is unavailable. > abc

### Route-driven observable data loading

Having put the navigational state (pointing to a specific entity) and the routing system, it is now possible to concisely wire this up to Observable-based data loading.

#### Component reuse

If only the route parameters change, the existing components (already up and running) will simply receive new route parameters via the observable mechanism. This is fast and preserves other less explicitly manage aspects of state (such as state uniform user is filling in) automatically, rather than discarding everything by reloading a page or re-creating all the components on the page. > abc

## Route Query Parameters

This step makes use of data filters/sorting and query parameters in the URL.

Query Parameters provide a human readable, GET style of adding data to the URL, without needing to wire up/change any routes.

This step also shows a sample of sorting/filtering data in a service to be provided to the view through an observable. > abc

## Managing Query Parameters

abc

## Route Guards

Route guards determine if a user is allowed to navigate from the current route to a new route. This example demonstrates two of the most commonly-used guards, CanActivate and CanDeactivate.

### CanActivate

This is the guard typically used to determine if a user is authorized to access a specific feature of an application. In the example, there’s an AuthService that’s a stand-in for a real authentication service, with a method that returns true if the current user is allowed to access the requested route. (There’s no actual authentication happening here; this is just a stub for the sake the of the demo, and the method always returns true until it’s changed in the source code.)

Note: This is a good chance to remind students that code running in a browser can’t offer any real security for an application – that has to happen on the backend. “Security” in a web client application is really more about user experience.

### CanDeactivate

CanDeactivate seems a little strange at first glance – why would you want to prevent a user from leaving a route? It makes sense when you consider that a user might have done something in the current route that they don’t want to lose, like fill out a long, complicated form. Before the user is allowed to navigate away, it’s a good idea to give them a chance to save their changes, and that’s what the CanDeactivate route guard allows.

## Other route guards

The other guards not demonstrated here, CanActivateChild and CanLoad, are discussed in the documentation.

Resolve can also be considered a kind of route guard. > abc

### Route guards - protecting routes

### Resolve - loading data during routing

abc

# Sharing state with a Service

State in services is the third of three parts in our state management series. This is the core of building with angular, stacking together components into routes, and is much as possible using the built-in facilities to manage their state. Then adding your own code for additional state beyond which can be managed in the box.

## Data fields in a Service

The most obvious way to store state and the service is to:

* Put the data in the field of the service
* Inject the service to your components
* Bind the data directly from the service variable onto the template

Unfortunately this technique is not advisable; it requires whole application change detection, and the scales poorly with size. It also doesn’t provide any good place to put code that responds to data change.

## Observable data in a Service

However, by putting data in an Observable in a service, the above problems are alleviated;

* The async pipe can be used to make data binding approximately is easy
* The observable mechanism provides a good place to put code which responds to data change
* The changes affirmatively communicated to the underlying Angular mechanisms, so that efficient “on push” operation (to be described a few steps down) works quickly even in very large applications. > abc

## NgRx Store - Elm/Redux Architecture

### An industry-wide idea

State in a central place

immutable data

control of change

### History

Elm

Redux

### Angular implementation

Angular is neutral as to what overall application architecture you adopt, other than encouraging a component-based approach, and urging one-way dataflow.

http://victorsavkin.com/post/137821436516/managing-state-in-angular-2-applications

It is possible to use the “Redux” architecture, for example. You can read more about it below.

https://medium.com/google-developer-experts/angular-2-introduction-to-redux-1cf18af27e6e#.e4h744w8t

The key piece of tooling to make this easy is NgRx; it is not part of the Angular project nor is it a Google product, but it is developed partly by people on the Angular team.

ngrx/store - RxJS powered state management for Angular applications, inspired by Redux: https://github.com/ngrx/store

Comprehensive Introduction to @ngrx/store: https://gist.github.com/btroncone/a6e4347326749f938510

Rob Wormald

http://onehungrymind.com/build-better-angular-2-application-redux-ngrx/

“My buddy Rob Wormald created an awesome Redux implementation using RxJS called @ngrx/store. This gives us all the power of Redux combined with the power of observables which makes for a very, very powerful stack.”

Mike Ryan’s advice on using ngrx/store:

https://twitter.com/MikeRyanDev/status/972160260000616448

If some piece of state meets these conditions put it in the Store:

1. The state will be shared with other components or services
2. Actions from other components or effects should cause the state to change
3. The state must be retrieved via a side effect (like a service call)

Otherwise leave it in your component or service. Don’t put everything in your global state container!

abc

### Immutable data with NgRx Store

### Managing complexity and boilerplate

In addition to what we have discussed and seen already, there are numerous libraries both from the NgRx team and outside, which add additional features or reduce boilerplate.

abc

## NgRx Effects

This step builds upon the ngrx/store introductory step to show a simple effect. In this example, we add two effects:

1. ackAll$ simply asks the user for confirmation (using window.confirm and the RxJS filter operator).
2. init$ uses @ngrx’s initialization mechanism to load the initial values before displaying the app. > abc

# Less-used Angular features (topics depend on time)

## ng-container

<ng-container> is a grouping syntax recognized by the Angular template parser. It’s useful when there’s no existing element to attach a structural directive to, or when adding a new <div> or <span> might cause other problems.

Despite its appearance, it isn’t a component or directive. After Angular has parsed the template, all that’s left behind in the DOM is a comment.

### Sentence example

The slider controls the “descriptiveness” – that is, the number of adjectives used – in the following two sentences, one created with <span>s and the other with <ng-container>s. The logic in both is the same; only the choice of <span> or <ng-container> is different.

Try commenting out the p > span rule in the CSS file to show how the apparently-innocuous <span>s affect the first sentence, but the <ng-container>s don’t harm the second.

### “Glossary of Useful Terms” example

<ng-container> is also useful when there’s not a single obvious element where an \*ngFor or \*ngIf can be attached. In this case, it’s a <dl> (what we used to call a “definition list”):

The HTML <dl> element encloses a list of groups of terms and descriptions. Common uses for this element are to implement a glossary or to display metadata (a list of key-value pairs).

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dl>

The odd thing about <dl> elements is that, unlike, say, <ul> or <ol>, their list items aren’t single HTML elements. They’re usually pairs of elements, a <dt> (“term”) and a <dd> (“definition”). That means that if you’re trying to use an \*ngFor to generate such a list, there’s not a single element to attach it to. <ng-container> solves this problem easily, as the example shows.

### “Important Programming Languages” example

Although Angular’s template parser will do its best to help you get away with it, you’re not supposed to just wrap a <div> around a group of table rows to turn them on or off. It’s illegal HTML, and can cause the browser to render the table in bizarre (and inconsistent) ways.

There’s no such concern with <ng-container>, since the only thing left behind after the template parser finishes processing an <ng-container> is a comment.

### Resources

Structural Directives - Group sibling elements with <ng-container> <https://angular.io/guide/structural-directives#group-sibling-elements-with-ng-container> > abc

## Providers

A provider is a tool of the dependency injection system that tells how to obtain the value of a dependency

abc

## Complex Animations

For more information about how to install and use Angular animations, check the Basic Animations step

### Stagger and query functions

* query: looks for inner elements meeting a certain criteria within the animated element; takes a condition (like, a state), and defines the action taken on it
* stagger: adds a time delay between animated elements; used with the query function

### State function

* state: defines different states to be defined with each animation; includes a name and a style definition
* the wildcard state ’\*’ is a placeholder for any state
* transition function defines the transition between two states; the state’s name or the wildcard state can be used in transition

### Parameterized animations

Syntax to add parameters to animations is demonstrated in the silicon wafers example and extended upon in the CMS examples.

More information: https://medium.com/@danieltamirr/parameterized-angular-animations-fa73a2727158

When the parameters are changed while an animation is action, the end state will twean towards the newest parameter values. Once the animation is completed, it resets to the beginning state and awaits a new trigger call. A way to overwrite this default functionality is to use the AnimationBuilder API

### Animation Builder

Docs for animation builder: https://angular.io/api/animations/AnimationBuilder Docs for animation player: https://angular.io/api/animations/AnimationPlayer

### Route Animations

Docs for route animations: https://angular.io/guide/route-animations > abc

## Complex forms

### Custom validators

Writing a custom validator is straightforward, it is just a function. See example code here for details.

Blog post with some info:

http://blog.thoughtram.io/angular/2016/03/14/custom-validators-in-angular-2.html

The validators in this step cannot yet be used as directives.

Because this example uses form builder, it uses the keys to refer to specific fields in the form level validator. If you are willing to use the form control group manual construction technique, you can use direct control references instead for slightly better IDE assistance. > abc

### Custom async validators

Blog post with some info:

http://blog.thoughtram.io/angular/2016/03/14/custom-validators-in-angular-2.html > abc

## Complex routing

### Secondary routes

Idea is multiple components on the same page with independent routing from each other.

Note especially the syntax to create a route link.

### Lazy Loading

Secondary routes do not work with lazy loading. Modules containing secondary routes/outlets must not be lazy-loaded. > abc

## Route Query Parameters

This step makes use of data filters/sorting and query parameters in the URL.

Query Parameters provide a human readable, GET style of adding data to the URL, without needing to wire up/change any routes.

This step also shows a sample of sorting/filtering data in a service to be provided to the view through an observable. > abc

## Complex forms

### Custom form controls

Implement the ControlValueAccessor interface to create a custom form control directive that integrates with Angular forms. > abc

### Reusing Observable data

This came up from a problem seen in class then again “in the wild”. There was a suggestion that observables made it hard/impossible to use the data again later without reloading it.

This shows that it is only slightly harder, just an extra operator. > abc

## Reactive forms

### Standalone reactive controls

abc

### Observable error recovery

TODO Update this text

Show how the observable in poll2() continues to run even as every other GET request fails. Contrast with poll1(), where the observable dies immediately after the first bad HTTP GET request. Even though the error in poll1() is detected and handled, the observable still stops.

Explain that poll2() keeps executing because it handles the error via a .catch() in the separate observable “error channel” (to use Ben Lesh’s term) inside getEmployees().

TODO Demonstrate .retry() and .retryWhen() operators

TODO Still need a lot more > abc

### Observable error handling

Show how the observable in poll2() continues to run even as every other GET request fails. Contrast with poll1(), where the observable dies immediately after the first bad HTTP GET request. Even though the error in poll1() is detected and handled, the observable still stops.

Explain that poll2() keeps executing because it handles the error via a .catch() in the separate observable “error channel” (to use Ben Lesh’s term) inside getEmployees().

TODO Demonstrate .retry() and .retryWhen() operators

TODO Still need a lot more > abc

abc

## The component lifecycle

Kyle has a lifecycle hook image somewhere.

http://stackoverflow.com/questions/33462532/using-resolve-in-angular2-routes http://stackoverflow.com/questions/34289761/angular-2-equivalent-to-router-resolve-data-for-new-router

constructor(myService: MyService, …) { … } The class constructor is called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here. ngOnChanges(changeRecord) { … } Called after every change to input properties and before processing content or child views. ngOnInit() { … } Called after the constructor, initializing input properties, and the first call to ngOnChanges. ngDoCheck() { … } Called every time that the input properties of a component or a directive are checked. Use it to extend change detection by performing a custom check. ngAfterContentInit() { … } Called after ngOnInit when the component’s or directive’s content has been initialized. ngAfterContentChecked() { … } Called after every check of the component’s or directive’s content. ngAfterViewInit() { … } Called after ngAfterContentInit when the component’s view has been initialized. Applies to components only. ngAfterViewChecked() { … } Called after every check of the component’s view. Applies to components only. ngOnDestroy() { … } Called once, before the instance is destroyed.

abc

## Change detection strategies - and why it matters

### How change detection works

* Each Component has a Change Detector
* Change Detection follows the component hierarchy
* Always top to bottom
* One pass - no loop, no Angular 1 digest “cycle”

In Angular, data always flows from top to bottom, and change detection works from top to bottom. This is why it is important, as we have talked about before earlier, to push changes upward using events.

When you change to the OnPush detection strategy, what you really mean is the change detection will only run when the shallow value/reference of an input changes. Angular will not, with OnPush, look any deeper inside.

### Change detection modes

* Default
* OnPush
* Others are for internally tracking what happens during change detection, and are not relevant at the programming level.

#### Documentation description of change detection settings

The documentation offers the following descriptions, which are not really all that helpful on their own.

OnPush: the change detector’s mode will be set to CheckOnce during hydration.

Default: the change detector’s mode will be set to CheckAlways during hydration.

### Zone.JS

* Came from Dart
* Under consideration to be included in a future JS version (TC39).
* Hooks browser events

<http://victorsavkin.com/post/110170125256/change-detection-in-angular-2>

<http://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html>

<https://angular.io/api/core/ChangeDetectionStrategy>

<https://blog.angular.io/3-tips-for-angular-runtime-performance-from-the-real-world-d467fbc8f66e> > abc

## Change detection strategies - and why it matters

### How change detection works

* Each Component has a Change Detector
* Change Detection follows the component hierarchy
* Always top to bottom
* One pass - no loop, no Angular 1 digest “cycle”

In Angular, data always flows from top to bottom, and change detection works from top to bottom. This is why it is important, as we have talked about before earlier, to push changes upward using events.

When you change to the OnPush detection strategy, what you really mean is the change detection will only run when the shallow value/reference of an input changes. Angular will not, with OnPush, look any deeper inside.

### Change detection modes

* Default
* OnPush

#### Documentation description of change detection settings

The documentation offers the following descriptions, which are not really all that helpful on their own.

OnPush: the change detector’s mode will be set to CheckOnce during hydration.

Default: the change detector’s mode will be set to CheckAlways during hydration.

### Zone.JS

* Came from Dart
* Under consideration to be included in a future JS version (TC39).
* Hooks browser events

<http://victorsavkin.com/post/110170125256/change-detection-in-angular-2>

<http://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html>

<https://angular.io/api/core/ChangeDetectionStrategy>

<https://blog.angular.io/3-tips-for-angular-runtime-performance-from-the-real-world-d467fbc8f66e> > abc

abc

## Decorator directives

### Introduction

### Examples

All of the Components we have seen were directives. This unit is about other kinds of directives.

https://github.com/angular/angular/blob/master/modules/angular2/docs/core/02\_directives.md

Class field decorators for directives and components import {Input, …} from ‘angular2/core’;

@Input() myProperty; Declares an input property that we can update via property binding (e.g. <my-cmp [my-property]=“someExpression”>).

@Output() myEvent = new EventEmitter(); Declares an output property that fires events to which we can subscribe with an event binding (e.g. <my-cmp (my-event)=“doSomething()”>).

@HostBinding(‘[class.valid]’) isValid; Binds a host element property (e.g. CSS class valid) to directive/component property (e.g. isValid).

@HostListener(‘click’, [’event).

@ContentChild(myPredicate) myChildComponent; Binds the first result of the component content query (myPredicate) to the myChildComponent property of the class.

@ContentChildren(myPredicate) myChildComponents; Binds the results of the component content query (myPredicate) to the myChildComponents property of the class.

@ViewChild(myPredicate) myChildComponent; Binds the first result of the component view query (myPredicate) to the myChildComponent property of the class. Not available for directives.

@ViewChildren(myPredicate) myChildComponents; Binds the results of the component view query (myPredicate) to the myChildComponents property of the class. Not available for directives. > abc

## Infinite Scrolling

This example demonstrates infinite scrolling, using a custom directive and rxjs operations. As a person scrolls to the bottom of a page, a new set of data is appended to the end of the current list of data.

Notes on Directives: https://angular.io/api/core/Directive

This example is based on the code found at: https://netbasal.com/build-an-infinite-scroll-component-in-angular-a9c16907a94d

In order to observe when a person has reached the bottom of a page of information, we use an Intersection Observer: https://developer.mozilla.org/en-US/docs/Web/API/Intersection\_Observer\_API The observer is set to detect intersections between the element that the directive is applied to and a defined “trigger” element. When this intersection occurs, a trigger event is sent from the directive.

When the EmployeesComponent receives the trigger event, it calls next on a Trigger Subject. The subject uses pagination techniques to request a chunk of employees (via the http call in the EmployeesService). The list of employees received in the request is then piped into two observables, which give us the list of employees and a boolean stating whether there are still data points to be loaded on the server. > abc

## Content Projection

abc

### Multiple Ng-Content

abc

## ngSwitch

This step is optional - only do it if the class is moving along quickly.

Should we care about a switch statement? Can’t you just use a bunch of IF statements instead?

Of course it is possible to just use IF and never use switch. But it is also good to have this switch in your toolbox. Switch more explicitly expresses to a human reader what you are trying to say. > abc

## Sanitization - secure handling of user-provided data

abc

## Component Inheritance

### Reusing code with a different template

Component Inheritance was introduced in Angular 2.3 While it may be a feature now supported by the framework we should be cautious to warn students of its use. There are a few good use cases for this feature that help component library authors. We should encourage students to rather then rely on Component Inheritance use well defined small components that compose our UI.

This example we have a complex data view component with the same Inputs and Outputs. We use Component Inheritance to easily allow different template views to display the same data. > abc

## Multi Providers

Using the multi: true option in a provider object allows for a service to be provided multiple ways

### Motivation for using multi providers

abc

## Custom pipes

By default, custom pipes are “pure” – they only re-run when they detect a change to a primitive or to an object reference (not to the properties within the object or array).

Impure pipes are run during every change detection cycle, and therefore get called far more often. To mark a pipe as impure, set “pure” to false in the pipe definition.

Discussion:

https://angular.io/guide/pipes#pure-and-impure-pipes

abc

## Custom pipes

By default, custom pipes are “pure” – they only re-run when they detect a change to a primitive or to an object reference (not to the properties within the object or array).

Impure pipes are run during every change detection cycle, and therefore get called far more often. To mark a pipe as impure, set “pure” to false in the pipe definition.

Discussion:

https://angular.io/guide/pipes#pure-and-impure-pipes

abc

## Unit testing

This is our introduction to unit testing. Here we unit test just a service in a very simple way. You can see all of the machinery necessary to do basic Angular testing.

Use CLI/Nx to generate and run tests with Jest.

### Simple spec

simple.spec.ts shows basics of a Jest test.

### Service Tests

* hello.service.spec.ts Simple service that can be tested without Angular > abc

### Jest mocks

Jest doesn’t include an “in the box” tool to make a mocked replacement for a class to use in testing. We think this might be because Jest’s heritage is from environments where classes aren’t mocked very often, but we don’t know for sure.

Yet mocking out a class, like an Angular service is extremely common in Angular testing, so we’re added on a tool to make it concise and type safe. We’ve chosen the jest-createspyobj package:

https://github.com/unlight/jest-createspyobj

import { createSpyObj } from ‘jest-createspyobj’;

There is also a more Angular-centric tool for the purpose, to consider instead:

https://github.com/hirezio/auto-spies/tree/master/packages/jest-auto-spies

import { Spy, createSpyFromClass } from ‘jest-auto-spies’; > abc

### Angular Testbed Module

abc

## More unit testing

### Unit testing pipes

pipes.spec.ts

### HTTP Mocking

* Mocking HTTP responses: sw.service.spec.ts > abc

### Asynchronous testing

* Mocking with synchronous observables: app.component.spec.ts
* Truly asynchronous tests: sw.service.spec.ts
* Manually stepping through time: app.component.spec.ts

abc

### Unit testing component templates

abc

## Component styling

### Scoped CSS

Each component uses a different style technique.

styles: []  
styleUrls: []  
styles tags

### ViewEncapsulation

Use the last (third.ts) component to use the different ViewEncapsulation modes and demonstrate the different side effects.

@Component({  
 encapsulation: ViewEncapsulation.Emulated, // default  
 encapsulation: ViewEncapsulation.None // Global  
})

Developer Guide - Component Styles: <https://angular.io/guide/component-styles>

### Targeting the component itself

You can use the :host selector to target styles to apply to the component itself, as it is an HTML element. It is often necessary to apply a display style as well, to make this element behave as you would expect. By default it does not behave like a DIV. > abc

abc

## Custom Preload Strategies

This step shows how to create an injectable class that is a custom implementation of the PreloadingStrategy (angular provided) abstract class.

There are 4 main route paths: \* Home: default path (lazily loaded as well) \* Lazy: lazily loaded (non-default path) \* Preloaded: preloaded behind the scenes immediately after the current path’s load \* Delayed: same as Preloaded but has a 2000 MS delay before it is loaded

To allow for the injectable to have the needed information to decide what to do, we can add a data object property to the Route objects.

Two main properties in the data object we set in this step: \* preload: boolean for whether to preload the module \* delay: combined with preload: true, allows for a delayed preloaded module

The custom injectable can be assigned to the preloadingStrategy property of the ExtraOptions object of the forRoot() function in the RouterModule.

Related Articles: https://coryrylan.com/blog/custom-preloading-and-lazy-loading-strategies-with-angular https://www.concretepage.com/angular-2/angular-custom-preloading-strategy > abc

## NgRx Feature Modules

abc

abc

abc

## Publishing Components Example

This example shows how to set up an Angular library for distribution, either as a Angular module or as a standalone web component.

To build a package suitable for distribution on npm as an Angular Element (a web component), run

npm run build-element

Relevant doc page: <https://angular.io/guide/elements> > abc

# Upgrading, third-party code, examples, and legacy features - extra material for Q&A

## Upgrading from AngularJS 1.x

Link to and explain the various tools, mostly those in ngUpgraders, project to unify upgrading.

https://github.com/ngUpgraders

Core team said something like this:

* Use A1 with component pattern.
  + Remove $scope
  + TODO add more from a slide in a talk Paul did
* Upgrade to 1.5
* Make 1.5 code look like 2.0, using ngForward
* Add A2 to the page
* Convert one component at a time
* Remove A1

However, we think many companies will not follow this process, rather they will keep old apps on A1 and build new apps on A2.

### ngUpgrade

A common approach is a kind of deep hybrid upgrade process where you can closely intermix old and new code. This is the official solution supported by the Angular team.

With this approach you upgrade one component at a time, resulting in AngularJS and Angular running simultaneously on the same page.

Some applications are suitable for that and some not.

#### A good set of videos from a AngularConnect 2017:

* [From Donkey to Unicorn, A new approach to AngularJS migration – Asim Hussain – AngularConnect 2017](https://www.youtube.com/watch?v=JxDuEwLfeGc)
* [Lightning Talk – Migrating a Portfolio of Front-End Apps to Angular – Craig Shipton & Mike Smith](https://www.youtube.com/watch?v=l2-FpuiQ3p4)
* [GCP: Migrating a 1M+ LOC project from AngularJS to Angular – Connie Wan & Jen Bourey](https://www.youtube.com/watch?v=xI7cP99BEnA)

#### Other advice from the Google team:

1. Use only a little bit of ng-upgrade; this approach is called something like “upgrade lite”. Make the two parts of your application touch mostly by running the two routers side-by-side. Some routes served by one, some route served by the other. A little bit of glue code so that if the user ends up in neither they get a helpful error message.
2. Use even less of it. Build your new application with Angular and in the places where you need to embed some old functionality, use an iFrame.
3. Of course, there are presumably various organizations following the official, much more deeply integrated path, where you can freely intermix services, components, etc. between Angular and AngularJS; but Oasis don’t actually know of a concrete case of this happening.

Also, https://github.com/thomsonreuters/FEF/tree/master/ng-rosetta provides an example of an application that was converted from AngularJS 1.x to componentize AngularJS to Angular 2.0

### ngForward

https://github.com/ngUpgraders/ng-forward

This looks like it might be the most complete / solid alternative to make A1 code like A2 code.

import { Component } from 'ng-forward';  
  
@Component({  
 selector: 'nested',  
 template: '<h3>Nested</h3>'  
})  
class Nested{ }

import { Injectable } from 'ng-forward';  
  
@Injectable()  
class TestService{  
 getValue(){  
 return new Promise(resolve => {  
 window.setTimeout(() => resolve('Async FTW!'), 3000);  
 });  
 }  
}

import { Component, Inject } from 'ng-forward';  
  
@Component({  
 selector: 'inner-app'  
 directives: [Nested],  
 template: `  
 <h2>Inner app</h2>  
 <p>ES7 async resolved value: {{ innerApp.num || 'resolving...' }}</p>  
 <nested></nested>  
  
 <h4>Event</h4>  
 <button (click)="innerApp.triggerEventNormally()">  
 Trigger DOM Event  
 </button>  
 <button (click)="innerApp.triggerEventViaEventEmitter()">  
 Trigger Emitted Event  
 </button>  
  
 <h4>One Way String from Parent (read-only)</h4>  
 <p>{{innerApp.msg3}}</p>  
  
 <h4>One Way Binding from Parent (read-only)</h4>  
 <input ng-model="innerApp.message1"/>  
  
 <h4>Two Way Binding to/from Parent (read/write)</h4>  
 <input ng-model="innerApp.message2"/>  
 `  
})  
@Inject(TestService, '$element')  
class InnerApp{  
 @Input() message1;  
 @Input() message2;  
 @Input('message3') msg3;  
  
 @Output() event1 = new EventEmitter();  
 @Output('event2') evt2 = new EventEmitter();  
  
 constructor(TestService, $element){  
 this.$element = $element;  
 this.TestService = TestService;  
 this.resolveValue();  
 this.evt2 = new EventEmitter();  
 }  
  
 async resolveValue(){  
 this.num = await this.TestService.getValue();  
 }  
  
 triggerEventNormally() {  
 this.$element.triggerHandler('event1');  
 }  
  
 triggerEventViaEventEmitter() {  
 this.evt2.next()  
 }  
}

### Where did 1.x features go?

* modules
* directives, including components and decorators
* controllers
* templates
* filters
* services / factory
* scope
* scope hierarchy
* controller as
* transclusion - view port

https://github.com/tolemac/Ng2Emulation

### Additional Resources

[Upgrading AngularJS - online video service](https://www.upgradingangularjs.com/)

abc

### Loading data with retry

This step explores and presents a production grade way to load data from a backend service; it automatically retries failed data loads, and present status to the user about the progress of data loading.

This is a relatively advanced piece of code in the loader directory, but it should be within grasp of the more advanced students.

Production grade projects typically need to reinvent code like this if it is not available already.

https://github.com/angular/angular/issues/10165

abc

## Wrapping an API with observables

In this unit, we will attempt to show our understanding of how you are supposed to use observables pervasively in an angular application. We don’t really know yet, so first we need to figure it out. > abc

## Dynamic Reactive Forms

It’s easy to generate reactive forms dynamically.

In this example, some metadata (the example schema) is used to generate the labels and input elements for a form, and to attach the appropriate validation criteria to each input element. If any element is invalid after the user has touched it, the relevant error message(s) are shown next to the element.

A few things to note:

* “Schema” isn’t an Angular term; it’s just what we’re calling the collection of per-field details in this particular example.
* We’ve kept this example simple by including the schema in the source code, but in a real application, it would probably be fetched from an external source. The form would still be created the same way.
* We’re dynamically generating the entire form, but it’s also easy to generate just a portion, then specify the rest with the usual static elements.
* Although this example only shows input elements, the same techniques work for other kinds of controls like <select> and <textarea>, and even for custom Angular components.
* This can get a lot more complex if necessary. FormGroup objects can be nested, for example, and FormArray can be used to handle things like lists of elements in a specific order.

Dynamic Forms Guide: <https://angular.io/guide/dynamic-form> > abc

### Transclusion use case example

This step shows a simple use case for ng-content Transclusion

Note the use of the custom schema selectors for the announcements directive to show a sense of customization to the custom transclusion directives.

* CUSTOM\_ELEMENTS\_SCHEMA is being imported from @angular/core and being put in at the module level as a ‘schema’. It is required to allow for the custom selectors created by the Announcements directive
* NO\_ERRORS\_SCHEMA is somewhat a valid choice as well, but also more of a hack for this particular case of custom selectors. It is more of a hack in this case since it will stop all errors resulting from the selector element names > abc

## Template reference variables

“A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component.”

https://angular.io/guide/template-syntax#ref-vars

This is especially helpful when interacting with non-Angular code. For a good example in class, we could load up some jQuery plug-in.

We used to cover this topic in the early steps around how templates work, but found it was needlessly confusing and distracting there. It is an advanced topic that you should learn after getting through all of the basics, when you are ready to do integration between Angular and other things.

These vars are also use together with @ViewChild(). Add a template reference variable in that element, i.e. <img #myImg>, and then you can reference it in your component with

@ViewChild('myImg') myImage;

abc

## Centralized state management with hierarchical DI

### State in observable in a service

An Angular 2 component can ask to have any parent upward in the hierarchy, injected via the normal DI mechanism. This can often be considerably easier than a long bucket brigade of bindings. But it also makes the relevant components more closely coupled with each other.

This example shows the value of hierarchical dependency injection. We inject a service which manages state, at the right “level” of the application. Then all the components inside that level automatically get the same shared state, while other components get a different state.

This example also shows will be think will be an important truth of Angular:

If your application has more than a trivial amount of state, the right way to manage it is to either use ngrx/store or similar, or reinvent ngrx/store or similar.

To illustrate in class the effect of hierarchical DI, simply move the “providers” entry which provides the state service, to different places in the component hierarchy:

* Move it to the top, to illustrate that the state is then shared application wide.
* Move it down to the individual subcomponents, which illustrates that the state is then no longer even shared among the different parts of a car order widget. > abc

## Observable bindings

In this step, we work with observable data bindings (async pipe) in more detail.

### Observables in more depth

This part might not even need Angular:

* series of values
* cold

full of functional programming ideas

Starting point is:

http://reactivex.io/

http://reactivex.io/tutorials.html

http://reactivex.io/documentation/observable.html

But it points you to the old Rx:

https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md

https://github.com/Reactive-Extensions/RxJS/

“rx”, version 4.0.x

Angular 2 uses the “RxNext” major revamp, which is here:

https://github.com/ReactiveX/RxJS

For a while you needed to do this; not any more: npm install @reactivex/rxjs

version 5.0.x

RxMarbles.org

Also Victor on the Angular team wrote a nice introduction to the various kinds of reactive programming:

https://vsavkin.com/the-taxonomy-of-reactive-programming-d40e2e23dee4

abc

## Infinite scroll

### Background info

old, feb 2016: http://orizens.com/wp/topics/angular-2-attribute-directive-creating-an-infinite-scroll-directive/

Here is an off the shelf directive: https://github.com/orizens/angular2-infinite-scroll … but that misses the point.

The goal for this step is not to use any off-the-shelf code, but rather to implement, right here, infinite scrolling. It will be a good exercise, and show people how to effectively use the capabilities in angular to do this. Our implementation should, to the extent it reasonably can, be implemented using off-the-shelf bindings and observables, and only make use of ad hoc callbacks is necessary.

The server already present (JSON server)can do just enough querying to be a useful backend for this. We may need to ship another, additional data file which has many rows but only a small amount of data per row. But then don’t fetch the whole thing, nor one of the time, but fetch them in ranges so as to most effectively illustrate infinite scroll. > abc

## Dynamic Locale

abc

## Promises - still available, still useful

This shows how the Angular HTTP Observable can be easily turned in to a Promise, and then handled as we would in A1.

This is just one unit, because we will do much less depth of promises on an A2 class. Not the 2-3 steps we do with A2. Spend the time on Observables instead. > abc

## HTTP API access via promises

This is an example of calling a more complex API, using HTTP and firstValueFrom().

Note that we’re (finally) loading a new script in index.html.

abc

## Two-way binding with ngModel

You can still get minimal two-way binding in A2, with different syntax. It does not need a form or any other support.

This is the first time we need to use a directive, though.

Box of bananas: <http://www.bennadel.com/blog/3008-two-way-data-binding-is-just-a-box-of-bananas-in-angular-2-beta-1.htm>

Importantly, two-way binding is not in the Angular 2 “core”. Rather, it is in the “forms module”, which you must add to the project. The purpose of this design is that applications which do not need NgModel, a rather complex piece of code, do not pay the price including it. > abc

### ngModel with more control types

As of release candidate 3, the model binding support is still somewhat incomplete. It works for most input types but not all. But it is possible to work around this was just a little extra work, and many Angular 2 applications have grown quite large already without much trouble caused by this.

#### TypeScript

In earlier drafts of the curriculum, we introduced more typescript complexity of this step. That seems too early. > abc

## Template-driven forms

http://blog.ng-book.com/the-ultimate-guide-to-forms-in-angular-2/

http://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/

Here’s the summary from Ward:

ngModel and 2-way data binding https://docs.google.com/document/d/1Kr\_LmYc1fqnVcOQdSLiSCD6X4-SMqsKRLiZU6rY3wsY/edit > abc

# Appendix: Finished example app

This is a video stat tracker app, similar to what is sometimes constructed during class. > abc

# Afterword

Thank you for reviewing these curriculum notes. We hope they have been a helpful accompaniment to our class.