npm install -g typescript

npm install -g @angular/cli@1.0.0-rc.4

ng new --ng4 angular-hello-world

ng generate component hello-world

ng serve

Notas dia #1

Preprocesadores

lass

a

Smart component

Como traerse los datos y como modificarlos

**Workshop ES6**

//clusters

(function() {

const asd = 2;

console.log(asd);

})()

//let vs var

//let cambia valor cuando se declara en distintos bloques

//var conserva el valor

//includes

//starts with

//endWith

//repeat

//Multiline String

//${variable}

// Default Parameters

//

// Rest Parameters, no lo entendi

// Tiene que ir de último

(function() {

function sum(first, ...values) {

console.log(first)

console.log(values)

console.log(arguments)

}

sum(1,2,3)

})()

// Spreat operator

// split para cuando pasamos los valores

//

//Arrow functions

//Argumento, flechita, cuerpo

//

//Objetos

//Shorthand

// return {name, age}

//

//Object.assign

//

//Destructuring

//Class declaration

//

//

//Array functions

//of

//from

//find and findinde

//fill

//

//promise.all()

//despues de terminar todas

//

//promise.race()

//la primer promesa en terminar

//link

//https://www.youtube.com/watch?v=FQE\_I96tOSk

//

//libro

//https://leanpub.com/understandinges6/read

Códigos libro

https://github.com/codeyu/ng-book-code

**Preguntas día 2,**

1- la carpeta vendor se crea de alguna forma?

**TypeScript Chapter**

* Typescript
  + Types
  + Annotations
    - Input
    - Output
    - HostBinding
* ES6
  + Classes
  + Modules
* ES5

Transpilers// Transcompiler

* Traceur created by Google
* Babel created by JavaScript Community

**What do we get with TypeScript?**

* Types
* Classes
* Decorators
* Imports
* Language utilities (e.g. destructuring)

**Types**

* Prevent bugs
* Clarify your code

**TSUN : Typescript Upgrated Node**

**REPL :Read–eval–print loop**

npm install -g tsun

The default initial value for an enum is 0

Having a variable of type any allows it to receive any kind of value:

<https://angular.io/guide/styleguide>

**Angular Standards**

1. 400 lines per file
2. Single Responsability Principle
3. Functions 75 line
4. feature.type.ts = type= .model, .component, .module, .service, , .pipe, .directive
5. Do append the symbol name with the conventional suffix (such as Component, Directive, Module, Pipe, or Service) for a thing of that type.
6. A few terms are unambiguously services. They typically indicate agency by ending in "-er". You may prefer to name a service that logs messages Logger rather than LoggerService. Decide if this exception is agreeable in your project. As always, strive for consistency.
7. Avoid putting app logic in main.ts. Instead, consider placing it in a component or service.
8. Do use a hyphenated, lowercase element selector value (e.g. admin-users).
9. upper camel case when naming classes.
10. export const mockHeroes = ['Sam', 'Jill']; // prefer
11. private \_toastCount: number; underscore for private attributes
12. Consider leaving one empty line between third party imports and application imports.
13. Consider listing import lines alphabetized by the module.
14. Feature modules
15. Do declare all components, directives, and pipes in the SharedModule. // Avoid providing services in shared modules.
16. Consider calling the application-wide core module, CoreModule. Add Services here
17. Avoid importing the CoreModule anywhere except in the AppModule.
18. Avoid allowing modules in sibling and parent folders to directly import a module in a lazy loaded feature.
19. Do use dashed-case or kebab-case for naming the element selectors of components.
20. Avoid input and output aliases except when it serves an important purpose.
21. Do place private members after public members, alphabetized.
22. Do limit logic in a component to only that required for the view. All other logic should be delegated to services.
23. Do name events without the prefix on.
24. Do name event handler methods with the prefix on followed by the event name.
25. Do put presentation logic in the component class, and not in the template.
26. Do use attribute directives when you have presentation logic without a template.
27. HostListener/HostBinding decorators versus host metadata Not clear.
28. Do use the @Injectable() class decorator instead of the @Inject parameter decorator when using types as tokens for the dependencies of a service.

Continue reading page 161

**Dependency Injection**

• **the Provider** (also often referred to as a binding) maps a token (that can be a string or a class)

to a list of dependencies. It tells Angular how to create an object, given a token.

• **the Injector** that holds a set of bindings and is responsible for resolving dependencies and

injecting them when creating objects

• **the Dependency** that is what’s being injected

**Using a class**

{ provide: UserService, useClass: UserService }

**Using a value**

providers: [

{ provide: 'API\_URL', useValue: 'http://my.api.com/v1' }

]

constructor(@Inject('API\_URL') apiUrl: string) {

*// works! do something w/ apiUrl*

}

Continue with Routing chapter

**Routing**

Hash based routing

• Routes describes the routes our application supports

• RouterOutlet is a “placeholder” component that shows Angular where to put the content of

each route

• RouterLink directive is used to link to routes

Continue page 250

**location strategy**: The way the Angular application parses and **creates paths** from and to route definitions. In **Angular 1 this is called routing modes** instead

The **default strategy is PathLocationStrategy**

**Nested routes** is the concept of containing routes within other routes.

ng generate guard logged-in

**Data Architecture**

1. **MVW / Two-way data binding: Model-View-Whatever** is a term used⁷² to describe **Angular 1’s default architecture**.
2. **Flux**⁷³: uses a unidirectional data flow. In Flux, **Stores hold data, Views render what’s in the**

**Store, and Actions change the data in the Store**. There is a bit more ceremony to setup Flux,

but the idea is that because **data only flows in one direction**, it’s easier to reason about.

1. **Observables**: **Observables give us streams of data**. We subscribe to the streams and then

perform operations to react to changes. **RxJs is the most popular reactive streams library for JavaScript.**

**Reactive programming**

* Reactive Programming is a way to work with asynchronous streams of data.
* Observables are the main data structure we use to implement Reactive Programming.
* Anything can be a stream: variables, user inputs, properties, caches, data structures, etc
  + What can you do with a stream?
    - Used as a input to another one
    - Filter
    - Map
    - Merge
* The "listening" to the stream is called **subscribing**.

**Introduction to Redux with TypeScript**



1. What’s a reducer?

A reducer must be a **pure function:**

* 1. The function always evaluates the same result value given the same argument value(s).
  2. Evaluation of the result does not cause any semantically observable [side effect](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) or output.
  3. nor can it depend on any external input from I/O devices.
  4. It must **not mutate the current state directly**
  5. It must **not use any data outside of its arguments**
  6. **pure function will always return the same value, given the same set of arguments**
  7. **won’t call any functions which have an effect on the outside world, e.g. no database calls, no HTTP calls, and no mutating outside data structures**
  8. **Reducers should always treat the current state as read-only. A reducer does not change the state instead, it returns a new state**

Example:

* 1. length(s), returning the size of a string s

**Inpure functions**

* 1. inc(x): x + a
  2. random() is impure because each call potentially yields a different value
  3. printf() and similar functions are impure because it causes output to an I/O device as a side effect.

1. An Action, which defines what to do (with optional arguments)
2. The state, which stores all of the data in our application
3. The Reducer which takes the state and the Action and returns a new state.

**Remember that in Redux we do not modify the state. Instead, we create actions which instruct the reducer on how to generate a new state.**

**In Redux, we keep our state in the store. The store has the responsibility of running the reducer**

**and then keeping the new state.**

This is an important principle of Redux: dispatching actions is a “fire-and-forget” maneuver.

Dispatching actions is not a direct manipulation of the state, and it doesn’t return the new

state.

When we dispatch actions, we’re sending off a notification of what happened. If we want to know

what the current state of the system is, we have to check the state of the store.