UNIVERSIDAD TECNOLOGICA NACIONAL

DELIVEREAT!

Servicio de Delivery

Documento de estilo de código

Curso: 4K3

Grupo: 1

Integrantes:

Barrionuevo, Natalia – Legajo: 87834

Becerra Bonnet, Abril – Legajo: 85835

Pregal, Santiago – Legajo: 81838

Ricapito, Juan Pablo – Legajo: 85381

Urzagasti Molina Víctor Nicolas – Legajo: 86007

Vega, María Sol – Legajo: 85526

TABLA DE CONTENIDOS

INTRODUCCIÓN

- - - - X

Cuando se escribe un documento de convenciones de estilo de código para un proyecto de Angular, es importante considerar varios aspectos, como la estructura de archivos, las convenciones de nomenclatura, las importaciones de módulos, los ganchos de ciclo de vida, las pruebas y los estilos de componentes.

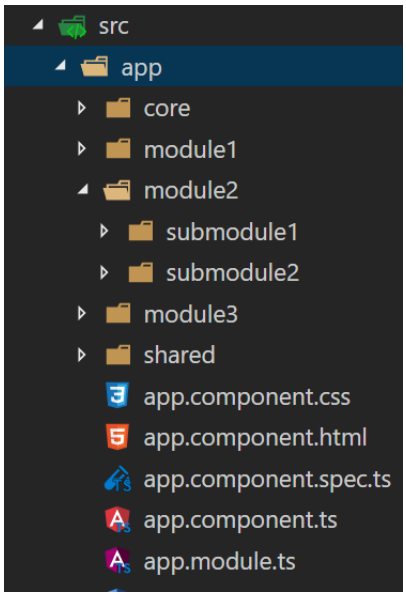
A continuación, se encuentran las guidelines y recomendaciones seguidas para la creación del código de implementación de la User Story “Realizar un Pedido de ‘lo que sea’”. Estas guidelines utilizan Angular con Typescript como referencia base.

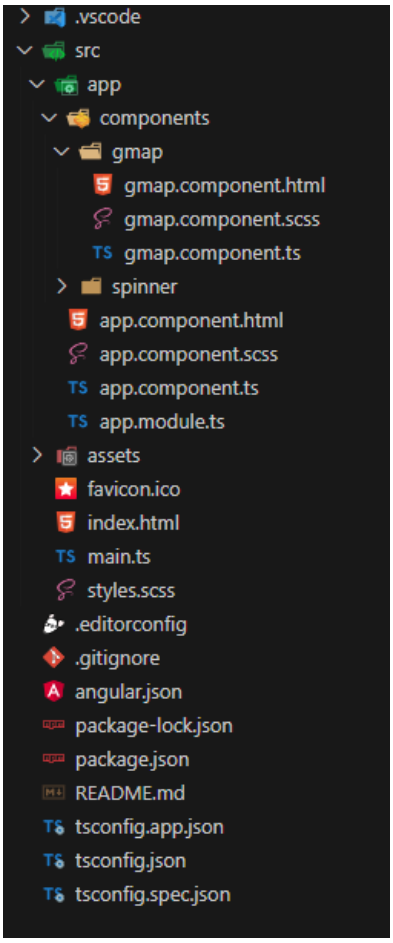
CONVENCIONES GENERALES

- - - - X

1. Convención de la estructura de archivos

Cada elemento que constituye un componente (html, css, ts), tendrá el mismo nombre (por ejemplo: hero.component.html, hero.component.css, hero.component.ts) y se mantendrán bajo la misma carpeta, lo que vuelve más legible y entendible la estructura de archivos. A la derecha nuestra estructura al momento de escribir esta guía; a continuación, un esquema general:





2. Definición de Responsabilidad

Regla del 1: Definir un componente o servicio por archivo y limitar las líneas de código a 500.

- Un componente por archivo lo vuelve más legible, mantenible y evita colisiones con el equipo en el control de fuente.
- Un componente por archivo evita bugs escondidos que emergen cuando combinamos componentes que comparten variables en un archivo, creando cierres, acoplamiento y dependencias no deseadas.

- Es una mejor práctica redistribuir el componente y sus clases de soporte en sus propios archivos dedicados.

Definir funciones pequeñas.

- Las funciones pequeñas son más fáciles de testear, especialmente cuando hacen una sola cosa (cumpliendo el SRP) y sirven un solo propósito.
- Las funciones pequeñas promueven la reutilización, son más fáciles de leer y de mantener.
- Las funciones pequeñas ayudan a evitar bugs escondidos que traen las funciones grandes que comparten variables con el entorno, crean cierres indeseados, o acoplamiento con dependencias

Por ejemplo:

```
ObtenerCoordenadas(): google.maps.LatLngLiteral {
    this.geocoder.geocode({
      address: this.numero + " " + this.calle + " , " + this.ciudad + ' , AR'
    }).subscribe(({ results }) => {
      this.markerPosition = results[0].geometry.location.toJSON();
    });
    return this.markerPosition!;

  }
```

3. Nombres de Variables:

Las convenciones del nombrado son inmensamente importantes para la legibilidad y mantenibilidad del código, las guías recomiendan convenciones para los nombres de archivos y los símbolos.

Utilizar nombres consistentes para cada símbolo.

Las convenciones de nombre ayudan a proveer una manera consistente de encontrar contenido a primera vista. La consistencia en el equipo de desarrollo es importante. consistencia en una compañía provee una gran eficiencia.

Las convenciones de nombre deberían ayudar a encontrar el código deseado rápido y hacerlo fácil de comprender.

Nombres de directorios y archivos deberían transmitir claramente su intención. Por ejemplo, app/heroes/hero-list.component.ts podría contener un componente que maneja una lista de héroes.

Separar archivos con puntos y guiones

- Usar guiones para separar palabras en el nombre descriptivo
- Usar puntos para separar para separar el nombre descriptivo de su tipo
- Usar nombres de tipos consistentes para todos los componentes siguiendo un patrón que describe las funciones del componente y luego su tipo. Un patrón recomendado es funcion.tipo.ts
- Usar los nombres de tipo convencionales en angular, incluyendo .service .component .pipe .module y .directive, inventar tipos adicionales si es necesario pero se debe tener en cuenta no crear demasiados.

Los nombres de tipo proveen maneras consistentes de identificar rápidamente qué contiene un archivo.

Los nombres de tipo hacen fácil encontrar archivos específicos usando el buscador de un editor o de un IDE.

Tipos sin abreviaciones como `.service` son descriptivos y no ambiguos, mientras que abreviaciones como `.srv .svc y .serv` pueden ser confusas.

Nombres de tipo proveen matcheo de patrones para cualquier tarea automatizada

Evitar utilizar alias para inputs y outputs

Evitar alias de inputs y outputs, excepto cuando sirva a un propósito importante.

Tener dos nombres para la misma propiedad (uno privado y uno público) es inherentemente confuso.

Deberías usar un alias cuando el nombre de la directiva también es una propiedad de entrada y el nombre de la directiva no describe adecuadamente la propiedad.

Por ejemplo:

```
 mapDirectionsService: MapDirectionsService;
  constructor(httpClient: HttpClient, geocoder: MapGeocoder,
mapDirectionsService: MapDirectionsService) {
    this.mapDirectionsService = mapDirectionsService;
    this.apiLoaded =
httpClient.jsonp('https://maps.googleapis.com/maps/api/js?key=APIKEY',
'callback').pipe(
      map(() => true),
      catchError(() => of(false)),
    );
    this.geocoder = geocoder;
  }
```

4. Indentación:

- Utilice tabuladores para la indentación.
- Coloque un espacio en blanco alrededor de operadores y después de comas en listas o argumentos de funciones.
- Use llaves en la misma línea que la declaración de control de flujo (ejemplo: if (condición) {) y una sangría de una tabulación.

5. Estilos de Componente:

Evitar alias de inputs y outputs, excepto cuando sirva a un propósito importante.

Tener dos nombres para la misma propiedad (uno privado y uno público) es inherentemente confuso.

Deberías usar un alias cuando el nombre de la directiva también es una propiedad de entrada y el nombre de la directiva no describe adecuadamente la propiedad.

Poner la lógica de presentación en la clase componente

Se pondrá la lógica de presentación en la clase del componente, y no en la plantilla.

La lógica estará contenida en un solo lugar (la clase componente) en lugar de estar dispersa en dos lugares.

Mantener la lógica de presentación del componente en la clase en lugar de en la plantilla mejora la comprobabilidad, la mantenibilidad y la reutilización.

Secuencia de miembros

Se colocarán las propiedades en primer lugar, seguidas de los métodos.

Se colocarán los miembros privados después de los miembros públicos, en orden alfabético.

Colocar los miembros en una secuencia consistente facilita la lectura y ayuda a identificar instantáneamente qué miembros del componente sirven para qué propósito.

Por ejemplo:

```
export class GmapComponent implements OnChanges {
  @Input() ciudad: string = 'Villa Carlos Paz';
  @Input() calle: string = '';
  @Input() numero: string = '';
  @Input() desde: google.maps.LatLngLiteral | undefined;
  @Output() adress_component = new
EventEmitter<google.maps.GeocoderAddressComponent[]>();
  @Output() ruta = new EventEmitter<number>();
  @Output() coordenadas = new EventEmitter<google.maps.LatLngLiteral>();
  @Output() existe = new EventEmitter<boolean>();
  apiLoaded: Observable<boolean>;
  markerOptions: google.maps.MarkerOptions = { draggable: false };
  markerPosition: google.maps.LatLngLiteral | undefined;
  zoom = 16;
  geocoder: MapGeocoder;
  center: google.maps.LatLngLiteral = { lat: -31.4252716, lng: -64.4972074 };
  addMarker(event: google.maps.MapMouseEvent) {
    this.markerPosition = event.latLng!.toJSON();

    this.coordenadas.emit(this.markerPosition);
    this.center = event.latLng!.toJSON();
    this.MostrarCalle(event.latLng!.toJSON());
  }
```

COMENTARIOS Y DOCUMENTACION

- - - - X

6. Comentarios en el Código

- Se agregarán comentarios descriptivos cuando sea necesario para explicar la lógica o el propósito de un bloque de código.
- Se mantendrán los comentarios actualizados a medida que cambie el código.

BUENAS PRACTICAS

- - - - X

7. Evitar la Duplicación de Código:

- Se reutilizarán funciones y se evitará la duplicación innecesaria de código.

8. Seguir los Principios SOLID:

- Se seguirán los principios SOLID de diseño de software cuando sea apropiado.

9. Manejo de Excepciones:

- Se utilizarán bloques try-catch para manejar excepciones de manera adecuada y se evitará atrapar excepciones genéricas. Por ejemplo:

```
  mapDirectionsService: MapDirectionsService;
  constructor(httpClient: HttpClient, geocoder: MapGeocoder,
mapDirectionsService: MapDirectionsService) {
    this.mapDirectionsService = mapDirectionsService;
    this.apiLoaded =
httpClient.jsonp('https://maps.googleapis.com/maps/api/js?key=APIKEY',
'callback').pipe(
      map(() => true),
      catchError(() => of(false)),
    );
    this.geocoder = geocoder;
  }
```

CONTROL DE VERSIONES

10. Uso de Git:

- Se utilizará Git para el control de versiones y se seguirán las convenciones de nomenclatura de ramas y commits establecidas por el equipo.

CONCLUSIONES

Esta guía de estilo de código proporciona una base sólida para mantener la calidad y la coherencia en el código fuente de *DeliverEat!*. Todos los miembros del equipo deben seguir estas pautas de manera consistente para garantizar que el código sea limpio, legible y de alta calidad. De esta manera, se busca mantener código de calidad a lo largo de todo el ciclo de vida del producto.

REFERENCIAS

https://runebook.dev/es/docs/angular/guide/styleguide

https://docs.angular.lat/guide/styleguide

https://github.com/excelmicro/typescript