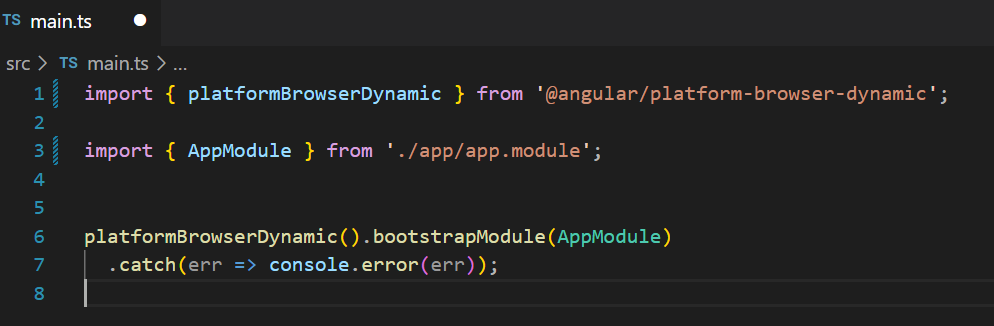
NgModules

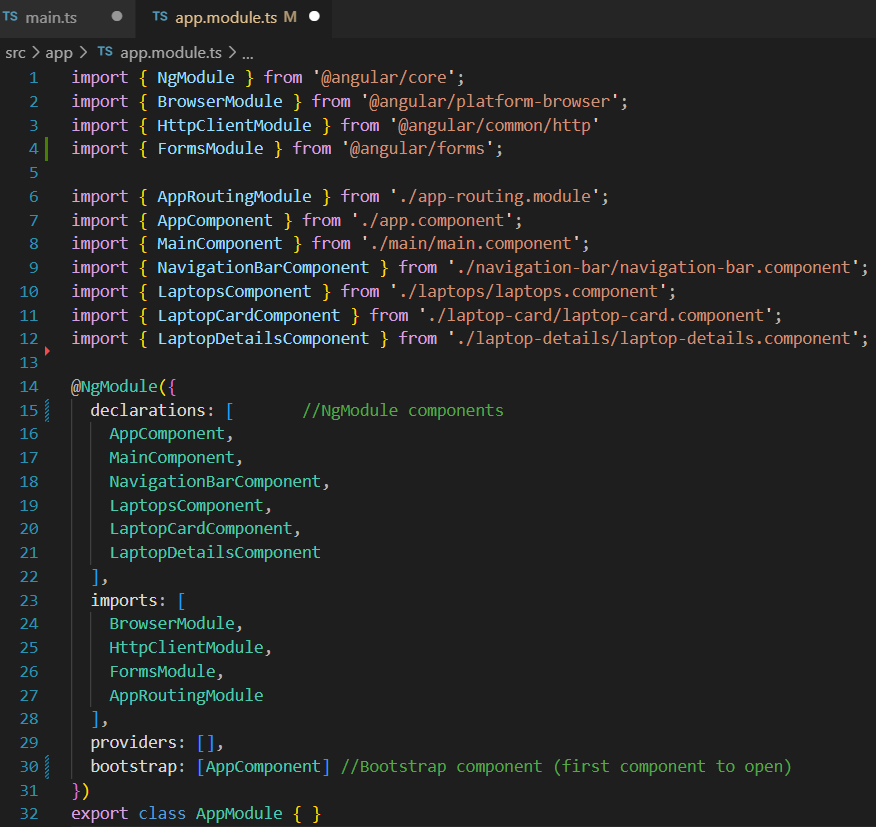
Other NgModules Services for DI

Component Component

Every application has a **Root Module** that enables bootstrapping, and typically has many more feature modules. A NgModule can associate its components with related code, such as services, to form functional units. NgModules are loaded by lazy-loading.

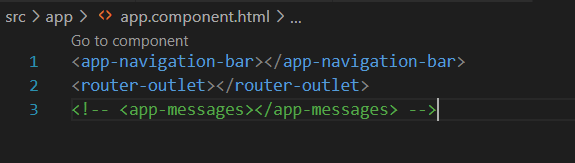
Main file:

Root Module:



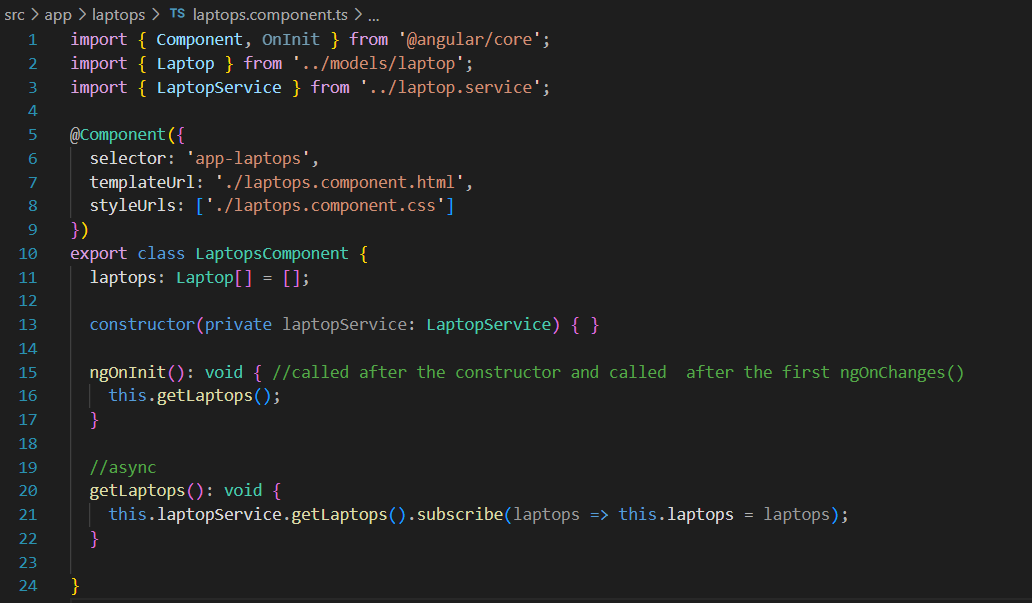
In Components you can add other components in `imports: ` (not in `declarations: `) if imported component is standalone (standalone: true, )

In provider you can enter services for DI (remember about imports)

Root module has rout ngModule named AppRoutingModule and components. App Component has

app-navigation-bar is a component, router-outlet is a directive from RouterModule, that is imported in AppRoutingModule:

The decorators like @Component() determine the type of file. Decorators allow to attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

Laptop component:

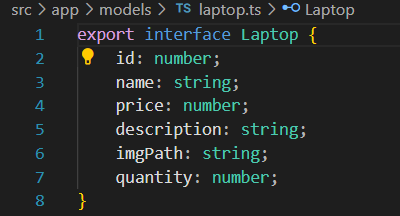
export class LaptopsComponent {

allows to do import and use class methods:

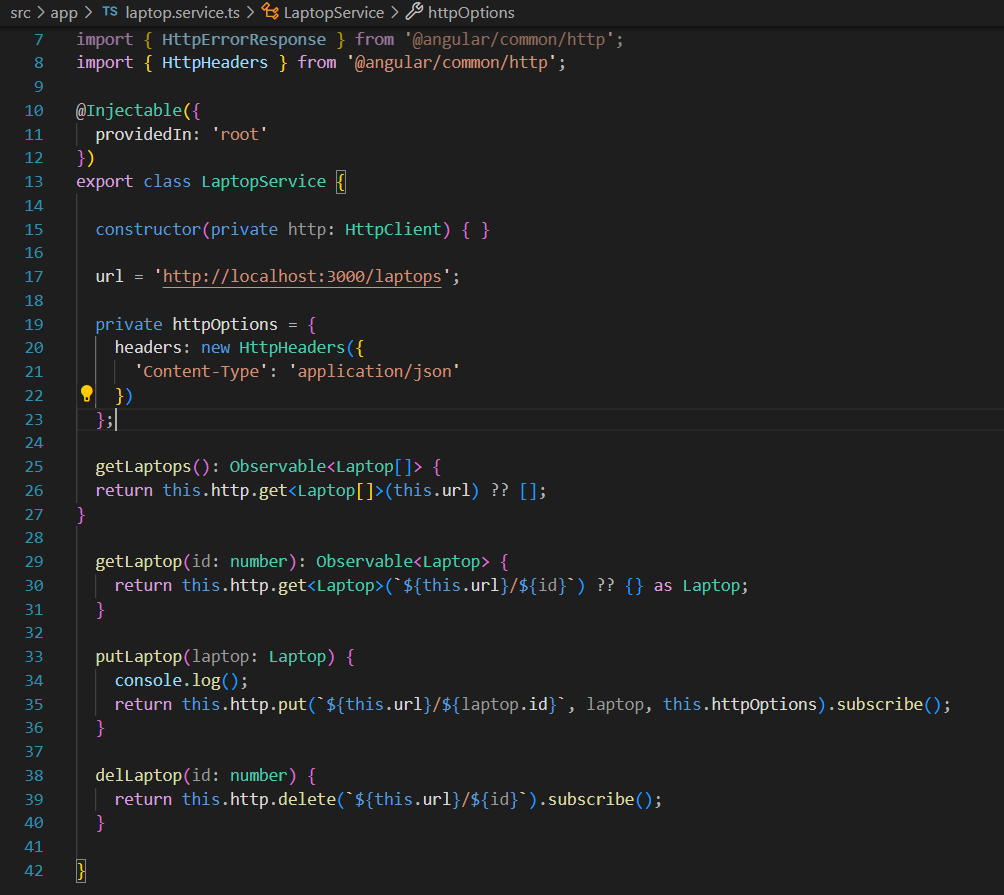
import { LaptopsComponent } from './laptops/laptops.component';

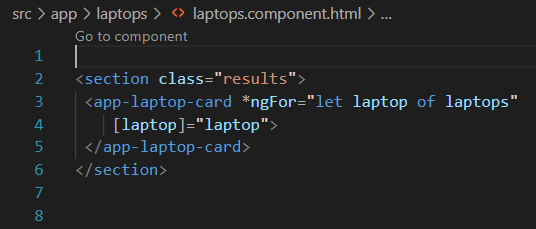
Injection the laptopService:

constructor(private laptopService: LaptopService) { }

Laptop:

laptopService: Services has decorator @Injectable for Dependency Injection



laptops.component.html represents laptops (declared in the .ts file) using directive \*ngFor and represents each laptop as app-laptop-card component:

(class=”results” is css class)

How to create and run angular app:

> ng new App-name

> cd App-name

> ng serve --open

The ng serve command:

* Builds the application
* Starts the development server
* Watches the source files
* Rebuilds the application as you make changes

The --open flag opens a browser to http://localhost:4200

Create a component:

> ng generate component *name* { --skip-tests || --inline-template }

Service/interface:

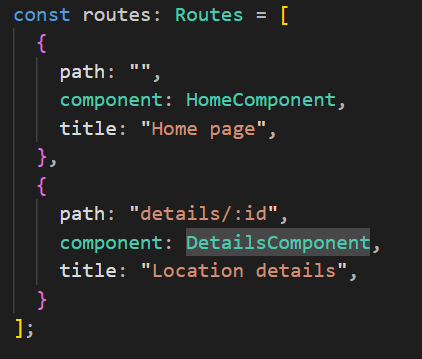


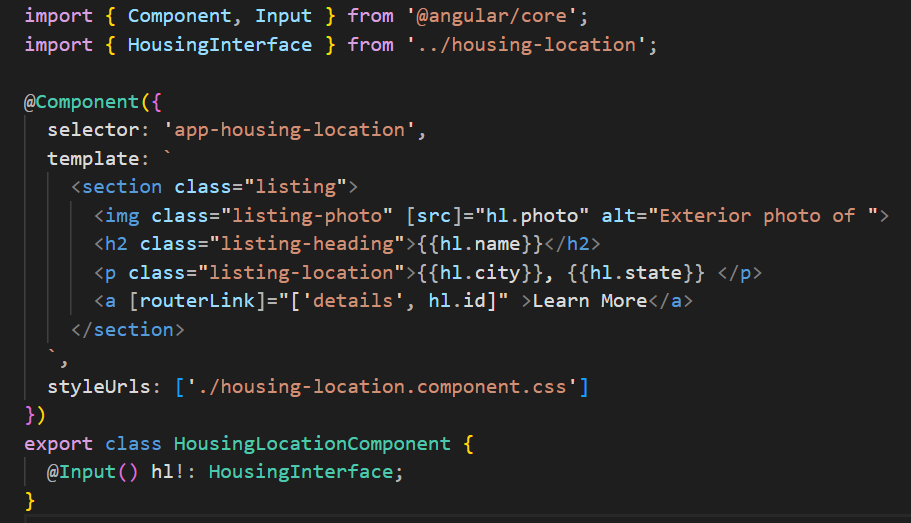
Figure 1 Routes

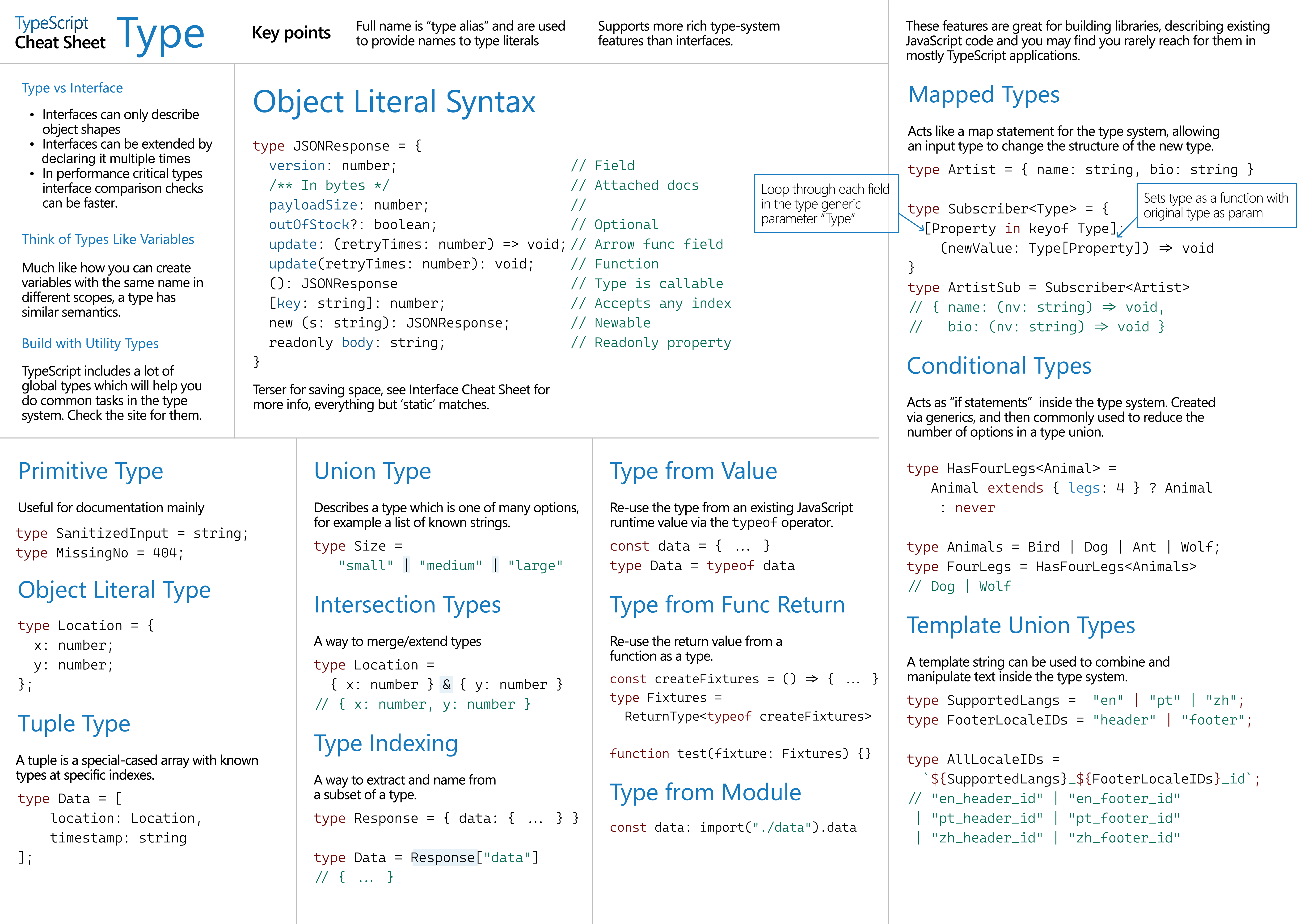
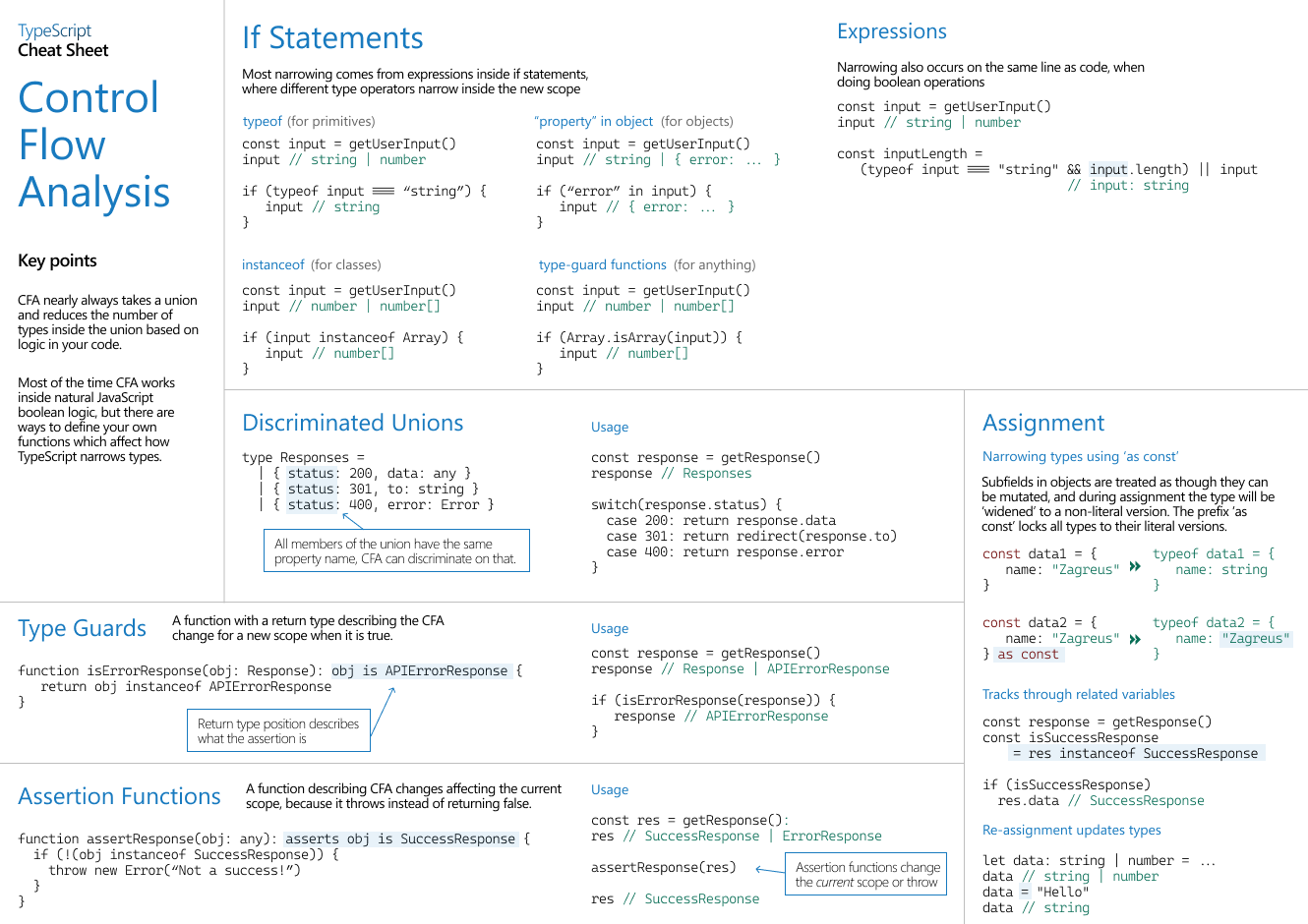
>ng generate service *name*

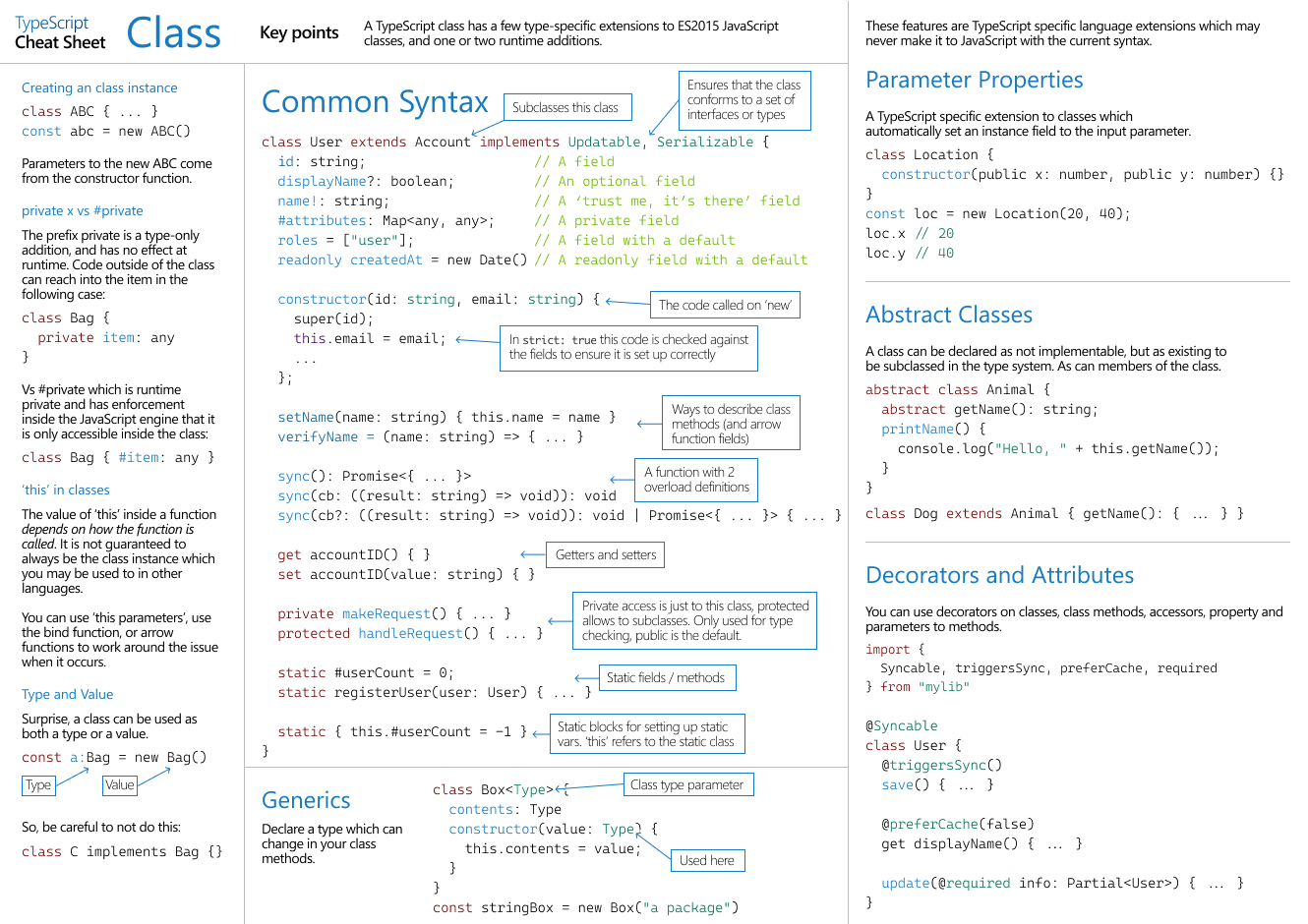
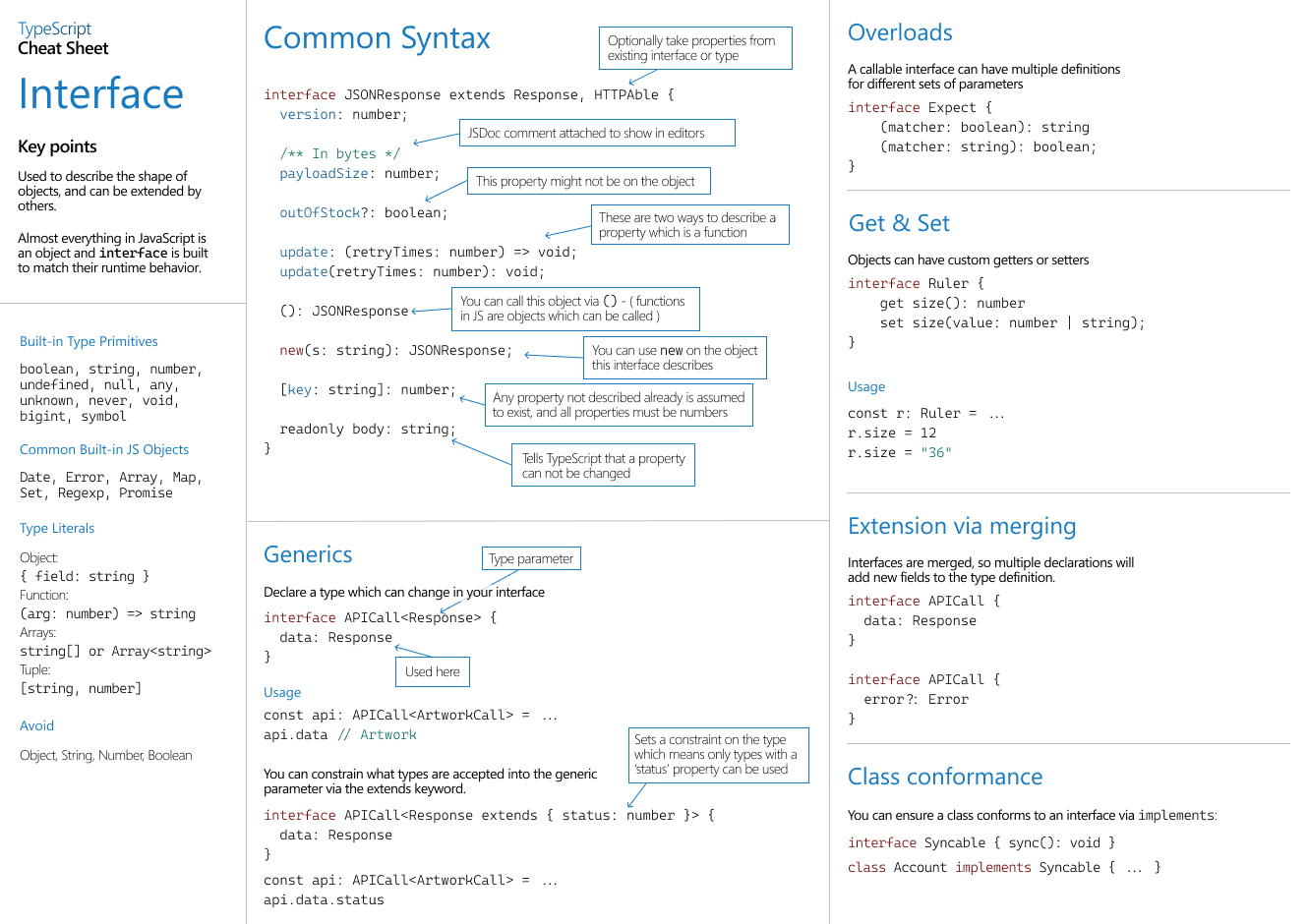
>ng generate interface *name*

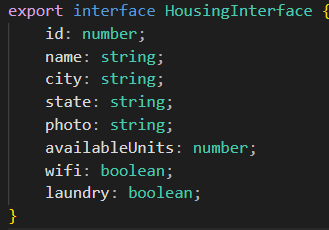
Injecting components into component:

Main component:

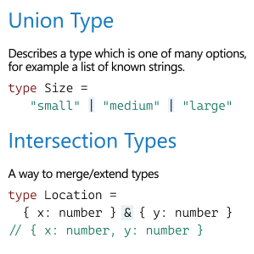
Subcomponents:

**TypeScript**



In TypeScript, both types and interfaces serve similar purposes and can often be used interchangeably for defining the shape of objects

Interfaces: Interfaces can be extended by other interfaces using the “extends” keyword and implemented by classes using the “implements” keyword. This allows you to build on existing interface definitions.



Types: Type aliases can be used to define unions, intersections, and more complex types, but they cannot be extended or implemented. They are generally more flexible for creating custom types.

Interfaces: Interfaces are generally used to describe the shape of objects, and their properties cannot be marked as **readonly**.

Types: Type aliases can be used with the **readonly** modifier to create **readonly** types, which enforce immutability for object properties.