What is DependenciesScanner in NestJs and how it is working

# Explain as simple as possible

In nest Dependency Scanner is class that go through each module module recursively from main module (AppModule) and register all modules in NestContainer and then for each module register all it’s providers and enhances (from providers controllers and methods).

# What is use of DependenciesScanner

1. Register core module named InternalCoreModule
2. Register all modules of our app
3. For each module register all it’s injectable
4. For each module register it’s controllers
5. Set distance (deph) property for each module
6. Assign all global modules to list of imports of each module
7. Add each module providers to serialize-graph

# How it works

1. First of all we create instance of DependencyScanner in nest-factory.ts file

const dependenciesScanner = new DependenciesScanner(

container,

metadataScanner,

graphInspector,

config,

);

1. Then we call our main method of **DependencyScanner**, method scan where we provide our main module (AppModule)
2. In scan method first we create **InternalCoreModule** and save it in **NestContainer** but we will not focuse on it right now
3. Second method is **scanForModules** to which we pass our main module, hear is what he is doing:
   1. Adds current module to NestContainer
   2. Also will generate a token (hash) for module
   3. Add module in **ctxRegistry** array with which we will check if module haven’t been added earlier
   4. Get current module imports and register all imported modules to nest container by calling **scanForModules** (which is current method)
   5. It means **scanForModules** method gose recursively through all modules and their imports (which are also modules and also could have other imported modules)
4. Once all modules are stored nest calls method named **calculateModulesDistance** which assigns to each module property distance which means deepness of module in reference to our main module (AppModule), also is going through each module recursively (as in 4.d.)
5. Then we call **scanModulesForDependencies** which main goal is to get all injectable (providers and enhancers) of module and register them under current module, hear it is how it works:
   1. First is I call **reflectImports** which takes all imported modules and assign them to their main module by calling **mainModule.addRelatedModule(subModule)** which adds to **mainModule** in field \_**imports** their **subModules,** while adding each imported module nest generates a hash for each module and store it in **NestContainer**, if module was already cached it is just returned from **NestContianer**
   2. Once all modules linked one between others it is time to get store providers and enhancers for each module, to do that we are calling **reflectProviders** on each module, this method takes all providers from module decorator and execute two methods under each provider
      1. First method is **insertProvider** it’s main goal is to create instance of provider and to add it to current module in \_**providers** Map  
         But if provider is enhancer or it’s scope is Request or Transcendent it is the same added to module but in field \_injectable
      2. Second method is much interesting, the method **reflectDynamicMetadata**  
         1. This method calls **reflectInjectables** over each enhancer which takes it from metadata of current provider, once current enhancers (for example guards) of provider are taken.  
         1.2. Next step is to go through each method of provider and to get is’ enhancers  
         1.3. After all enhancers of provider are taken is time to register them by calling **insertInjectable** over provider and it’s methods enhancers  
         1.4. While executing **insertInjectable** first we are storing enhancers under their modules and then store each enhancer in **graphInspector** class (don’t focus right now)  
         2. When all enhancers types are registered we call **reflectParamInjectables** which takes current provider and get’s it’s methods and go through each method args (which are always pipes) and register them by calling **insertInjectable (look above 1.4)**
   3. Once all provider enhancers are registered exactly the same thing is done over all controllers unique difference is that controllers are stored in \_**controllers** Map but other actions are done the same as for providers
   4. The last part is to store all exported providers under current module
6. Next method is very simple, **calculateModulesDistance()** main goal is to take our main module (AppModule) and to go through each of it’s sub modules recursively and to set them distance property which describes depth of module in reference to main module, this property helps nestjs in resolving problems with circular dependency
7. Next method **addScopedEnhancersMetadata()** takes all global enhancers and Request/Transcendent providers then get their instance of from their module and all for each module take all it’s controllers and providers instances and link them to each of enhancers by assigning to their metadata current enchancer
8. The last method **bindGlobalScope()** assigns to each module imports all global modules.
9. Once scanning process is done **dependenciesScanner** calls second method named **applyApplicationProviders,** It’s main goal is to go through each provider and to store it in applicationConfig under it’s enhancer type