# ANGULAR 2

**Lazy Loading a Module**

Another advantage of using modules to group related pieces of functionality of our application is the ability to load those pieces on demand. Lazy loading modules helps us decrease the startup time. With lazy loading our application does not need to load everything at once, it only needs to load what the user expects to see when the app first loads. Modules that are lazily loaded will only be loaded when the user navigates to their routes.

To show this relationship, let's start by defining a simple module that will act as the root module of our example application.

*app/app.module.ts*

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { EagerComponent } from './eager.component';

import { routing } from './app.routing';

@NgModule({

imports: [

BrowserModule,

routing

],

declarations: [

AppComponent,

EagerComponent

],

bootstrap: [AppComponent]

})

export class AppModule {}

So far this is a very common module that relies on the BrowserModule, has a routing mechanism and two components: AppComponent and EagerComponent. For now, let's focus on the root component of our application (AppComponent) where the navigation is defined.

*app/app.component.ts*

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<h1>My App</h1>

<nav>

<a routerLink="eager">Eager</a>

<a routerLink="lazy">Lazy</a>

</nav>

<router-outlet></router-outlet>

`

})

export class AppComponent {}

Our navigation system has only two paths: eager and lazy. To know what those paths are loading when clicking on them we need to take a look at the routing object that we passed to the root module.

*app/app.routing.ts*

import { ModuleWithProviders } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { EagerComponent } from './eager.component';

const routes: Routes = [

{ path: '', redirectTo: 'eager', pathMatch: 'full' },

{ path: 'eager', component: EagerComponent },

{ path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }

];

export const routing: ModuleWithProviders = RouterModule.forRoot(routes);

Here we can see that the default path in our application is called eager which will load EagerComponent.

*app/eager.component.ts*

import { Component } from '@angular/core';

@Component({

template: '<p>Eager Component</p>'

})

export class EagerComponent {}

But more importantly, we can see that whenever we try to go to the path lazy, we are going to lazy load a module conveniently called LazyModule. Look closely at the definition of that route:

{ path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }

There's a few important things to notice here:

1. We use the property loadChildren instead of component.
2. We pass a string instead of a symbol to avoid loading the module eagerly.
3. We define not only the path to the module but the name of the class as well.

There's nothing special about LazyModule other than it has its own routing and a component called LazyComponent.

*app/lazy/lazy.module.ts*

import { NgModule } from '@angular/core';

import { LazyComponent } from './lazy.component';

import { routing } from './lazy.routing';

@NgModule({

imports: [routing],

declarations: [LazyComponent]

})

export class LazyModule {}

If we define the class LazyModule as the default export of the file, we don't need to define the class name in the loadChildren property as shown above.

The routing object is very simple and only defines the default component to load when navigating to the lazy path.

*app/lazy/lazy.routing.ts*

import { ModuleWithProviders } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { LazyComponent } from './lazy.component';

const routes: Routes = [

{ path: '', component: LazyComponent }

];

export const routing: ModuleWithProviders = RouterModule.forChild(routes);

Notice that we use the method call forChild instead of forRoot to create the routing object. We should always do that when creating a routing object for a feature module, no matter if the module is supposed to be eagerly or lazily loaded.

Finally, our LazyComponent is very similar to EagerComponent and is just a placeholder for some text.

*app/lazy/lazy.component.ts*

import { Component } from '@angular/core';

@Component({

template: '<p>Lazy Component</p>'

})

export class LazyComponent {}

[View Example](https://plnkr.co/edit/vpCqRHDAj7V6mlN1AknN?p=preview)

When we load our application for the first time, the AppModule along the AppComponent will be loaded in the browser and we should see the navigation system and the text "Eager Component". Until this point, the LazyModule has not being downloaded, only when we click the link "Lazy" the needed code will be downloaded and we will see the message "Lazy Component" in the browser.

We have effectively lazily loaded a module.

**Module VS Component**

In general, we say that module is something which is made from multiple sub modules.

Similarly, a module in Angular2 is something which is made from components, directives, services etc. One or many modules combines up to make an Application.

Now Components in Angular2 are classes where you write your logic for the page you want to display.

Consider an example,

You have to create an application for College Management. You divide your project into multiple parts like Admin, Students, Staff etc. These are modules.

Now in one of the module like Admin, you will have Add, Delete, Update User Info. These will be Components that will combine up and create a module.

### **Why NgModule ?**

It’s done automatically with [Angular CLI](https://github.com/angular/angular-cli), but the first thing you have to do in Angular is to load a root [NgModule](https://angular.io/docs/ts/latest/guide/ngmodule.html" \t "_blank) :

**The purpose of a NgModule is to declare each thing you create in Angular**. There is two kind of main structures :

* **“declarations” is for things you’ll use in your templates : mainly components** (~ views : the classes displaying data), but also directives and pipes ;
* **“providers” is for services** (~ models : the classes getting and handling data).

Why do we need to register everything ? NgModule was a structure introduced lately in the RC phase of Angular 2. First, it seemed unnecessary complexity, as it feels redundant with ES6 imports. But now we’re used to it, you can rejoice : **it allows**[**Ahead of Time (AoT) compilation**](https://angular.io/docs/ts/latest/cookbook/aot-compiler.html)**, which is amazing for performance**. And it may seems heavy at first, but it actually saves you many lines of imports : in beta versions of Angular 2, you needed to import your components and directives every time you used them (good luck if you use UI modules like Material).

### **NgModule and scopes / visibility**

The confusion starts with **declarations and providers not having the same scope / visibility** :

* declarations / **components are in local scope** (private visibility) ;
* providers / **services are in global scope** (public visibility).

It means the **components you declared are only usable in the current module**. If you need to use them outside, in other modules, you’ll have to export them :

On the contrary, **services you provided will be available / injectable anywhere in your app**, in all modules.

**When to import a NgModule ?**

The difference of scope between components and services is an important point to know, but for now it’s still OK. Things get messy because, of course, as in any framework and app, you won’t just have one module, but many of them. Angular itself is subdivided in different modules (core, common, http and so on).

So another main thing you do in an Angular module is to **import other NgModules** you need :

Problem is : **you need to know why you import these other modules.**

* Is it to use components (or other template-related things, like directives and pipes) ?
* or is it to use services ?

Why ? Because given the difference of scope between components and services :

* **if the module is imported for components, you’ll need to import it in each module** needing them;
* **if the module is imported for services, you’ll need to import it only once**, in the first app module.

If you fail to understand this, you’ll have errors on components not being available, because you forgot to import their module again.

Or if you import a module for services more than once, you’ll create several instances of the same service in your app, which can lead to errors, but more importantly is bad for performance.

### **When to import main Angular modules ?**

**A good knowledge of Angular modules is then required, to know how many times you need to import them**. Here is an helpful summary.

#### **Modules to import each time you need them**

* **CommonModule** (all the basics of Angular templating : bindings, \*ngIf, \*ngFor…), except in the first app module, because it’s already part of the BrowserModule
* FormsModule / ReactiveFormsModule
* MatXModule and other UI modules
* RouterModule **without forRoot() or forChild()**(for routerLink)
* any other module giving you components, directives or pipes

#### **Modules to import only once**

* **HttpClientModule**
* BrowserAnimationsModule or NoopAnimationsModule
* any other module providing you services only.

### **The SharedModule good practice**

**That’s why with**[**Angular CLI**](https://github.com/angular/angular-cli)**, CommonModule is automatically imported when you create a new module.**

**If you use other components-related modules like animations, flex layout or Material, it will be tiring to have to import their modules every time. So a good practice is to create a [SharedModule](https://angular.io/docs/ts/latest/guide/ngmodule.html" \l "!#shared-module" \t "_blank) to factorize.**

**But be careful how you manage this. It won’t work if you just import the modules :**

**Why ? Still a scope problem : the components will be available in the SharedModule itself. But it’s not where you’ll use them, it’s in other modules which include this SharedModule. So you need to export them :**

**If your SharedModule includes other shared things (like the app menu), stay on the previous code. But if your SharedModule is just here to factorize and doesn’t include anything else, you can simplify by skipping the imports property :**

### **Mixed NgModules**

It can get messier : how to manage modules with components and services at the same time ?

You know one of them : the **RouterModule**. It gives you a component (<router-outlet>) and a directive (routerLink), but also services (ActivatedRoute to get URL params, Router to navigate…).

Fortunately, the mess is managed by the module itself. Routing files are automatically generated by [Angular CLI](https://github.com/angular/angular-cli), but you may have noticed there is a subtle difference between the routing of your first app module and the routing of submodules.

For the AppModule, it does :

For submodules, it does :

Why ? Because the first time in app module, forRoot() will give the router components and provide the router services. But the next times in submodules, forChild will only give the router components (and not providing again the services, which would be bad).

### **Lazy-loaded modules**

Last complication : if you lazy-load a module, which is now easy with [Angular CLI](https://github.com/angular/angular-cli).

As it will be a different bundle and module, loaded only on demand by default, **it’s not included in the global scope your app**.

**For components, it doesn’t change anything** : you need to import again the CommonModule or your SharedModule, like in any submodule.

For services, there is a difference :

* **you’ll still have access to services already provided in the app** (like HttpClient and your own services) ;
* but **the services provided in your lazy-loaded module will only be available in this lazy-loaded module**, not everywhere in your app.

### **Conclusion : why ?**

Now you know everything about Angular modules, you may ask : why this mess ? Well, it may be difficult for beginners, but there is a good reason :

* services are mostly just ES6 classes : they are imported/exported, so in their namespaces, so no risk of collision !
* components create… components, ie. new HTML tags : if they were global, loading two librairies creating components with the same name would create conflicts.

[**CONFIGURING DEPENDENCY INJECTION IN ANGULAR**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/configuring/)

* [Learning Objectives](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_learning_objectives)
* [Setup](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_setup)
* [NgModule.providers](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_ngmodule_providers)
* [Component.providers](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_component_providers)
* [Component.viewProviders](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_component_viewproviders)
* [Summary](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_summary)
* [Listing](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_listing)

We can configure injectors in Angular by:

1. providers on NgModule.
2. providers on Components and Directives.
3. viewProviders on Components.

So the question is where do you configure your provider?

Understanding where to configure your provider is a key piece of understanding how to architect your application, so we are going to explain this via a real practical example.

[**Learning Objectives**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_learning_objectives)

* Know the difference between configuring a provider on an NgModule, a component or directives providers property and a components viewProviders property.

[**Setup**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_setup)

We create a class called SimpleService which has one property called value which holds a string.

Copyclass SimpleService {

value: string;

}

We also have a component called ParentComponent which has a child component called ChildComponent.

Copy@Component({

selector: 'child',

template: `

<div class="child">

<p>Child</p>

{{ service.value }}

</div>

`

})

class ChildComponent {

constructor(private service: SimpleService) { }

}

|  |  |
| --- | --- |
|  | We use *string interpolation* to bind to the value property of SimpleService. |
|  | We *inject* an instance of SimpleService into the constructor. |

Copy@Component({

selector: 'parent',

template: `

<div class="parent">

<p>Parent</p>

<form novalidate>

<div class="form-group">

<input type="text"

class="form-control"

name="value"

[(ngModel)]="service.value">

</div>

</form>

<child></child>

</div>

`

})

class ParentComponent {

constructor(private service: SimpleService) { }

}

|  |  |
| --- | --- |
|  | We use *two way data binding* to bind to the value property of SimpleService. |
|  | We render the ChildComponent inside this ParentComponent. |
|  | We *inject* an instance of SimpleService into the constructor. |

The ParentComponent has just one input box which reads and writes to the SimpleService valueproperty using two way ngModel binding, the ChildComponent just renders the value to the screen with {{ }}.

We render two *side by side* <parent> tags in our root AppComponent module, like so:

Copy@Component({

selector: 'app',

template: `

<div class="row">

<div class="col-xs-6">

<parent></parent>

</div>

<div class="col-xs-6">

<parent></parent>

</div>

</div>

`

})

class AppComponent {

}

We set up our NgModule and bootstrap it, like so:

Copy@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, ParentComponent, ChildComponent ],

bootstrap: [ AppComponent ]

})

class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);

In the end when we run our application we should end up with something that looks like this:

#### **Note**

We have also added some css styles on our component which has been removed from the above code, the full code can be found in the listing at the end of this lecture.

[**NgModule.providers**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_ngmodule_providers)

We’ll first configure our SimpleService on the root NgModule, like so:

Copy@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, ParentComponent, ChildComponent ],

bootstrap: [ AppComponent ],

providers: [ SimpleService ]

})

class AppModule { }

|  |  |
| --- | --- |
|  | We’ve configured our NgModule with a class provider of SimpleService. |

In this configuration the service has been injected onto our applications root NgModule and therefore is in our root injector.

So every request to resolve and inject the token SimpleService is going to be forwarded to our single root injector.

Therefore since we only have one injector which is resolving the dependency, every-time we request an instance of SimpleService to be injected into one of our components it’s *always* going to inject the *same* instance.

#### **Important**

Remember if we request the same token from the same injector we get the same instance.

Since we’ve bound the input field directly to the simple service value field **and** it’s the same instance of simple service used everywhere, then when we type into one input control it automatically updates the other input control and also the child components.

#### **Tip**

If we want to share one instance of a service across the entirety of our application we configure it on our NgModule.

#### [**Component.providers**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_component_providers)

Let’s now see what happens when we configure our SimpleService *additionally* on the ParentComponentvia the providers property.

Copy@Component({

selector: 'parent',

template: `...`,

providers: [ SimpleService ]

})

class ParentComponent {

constructor(private service: SimpleService) { }

}

Now *each* ParentComponent has it’s *own* child injector with SimpleService configured, like so:

We can see from the running the code above that if we type into one parent component only *that* parent component and it’s child component automatically updates, like so:

Each instance of ParentComponent now has it’s *own* instance of SimpleService, so state is not shared globally but only between a ParentComponent and it’s child components.

That’s because each instance of ParentComponent has it’s own child injector with SimpleServiceconfigured as a provider.

#### **Important**

Remember when we request the same token from different injectors we get the different instances.

When we configured the SimpleService on the parent component it created a child injector, and when we tried to inject SimpleService into the parent component constructor it resolved and created an instance of SimpleService from it’s own injector.

#### **Tip**

If we want to have one instance of a service per component, and shared with all the components children, we configure it on the providers property on our component decorator.

## [Component.viewProviders](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_component_viewproviders)

If we now configure the SimpleService provider on the viewProviders property on the ParentComponent nothing changes, we still get the functionality we had before.

But lets use content projection and the ng-content component to change the child component from being a view child of parent to to being a content child of parent. i.e. lets pass in <child></child> to the parent component like so:

Copy<**parent**><**child**></**child**></**parent**>

So we change the AppComponent template to pass in child to the parent component, like so:

Copy <**div** class="row">

<**div** class="col-xs-6">

<**parent**><**child**></**child**></**parent**>

</**div**>

<**div** class="col-xs-6">

<**parent**><**child**></**child**></**parent**>

</**div**>

</**div**>

Change the ParentComponent template to project the passed in content to the same place the child component used to be, like so:

Copy<**div** class="parent">

<**p**>Parent</**p**>

<**form** novalidate>

<**div** class="form-group">

<**input** type="text"

class="form-control"

name="value"

[(ngModel)]="service.value">

</**div**>

</**form**>

<**ng-content**></**ng-content**>

</**div**>

|  |  |
| --- | --- |
|  | We use content projection to insert the ChildComponent where it used to be hard coded. |

Now even though child is still rendered under parent, it’s considered a content child and not a view child.

Lets now change the configuration of ParentComponent to use viewProviders instead.

Copy@Component({

selector: 'parent',

template: `...`,

viewProviders: [SimpleService ]

})

class ParentComponent {

constructor(private service: SimpleService) { }

}

Now when we type into the ParentComponent the child component doesn’t update automatically.

That’s because when using viewProviders the component creates an injector which is **only** used by the current component and any view children.

If you are a content child, as our child component now is, then it uses the injector in NgModule to resolve the dependency.

#### **Tip**

If we want to have one instance of a service per component, and shared with only the components view children and not the components content children, we configure it on the viewProvidersproperty on our component decorator.

[**Summary**](https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/#_summary)

We can configure the DI framework in Angular in three main ways.

We can configure a provider on the NgModule, on a component or directives providers property and on a components viewProviders property.

Deciding where to configure your provider and understanding the different is key in understanding how to architect an Angular application.

If we want an instance of a dependency to be shared globally and share state across the application we configure it on the NgModule.

If we want a separate instance of a dependency to be shared across each instance of a component and it’s children we configure it on the components providers property.

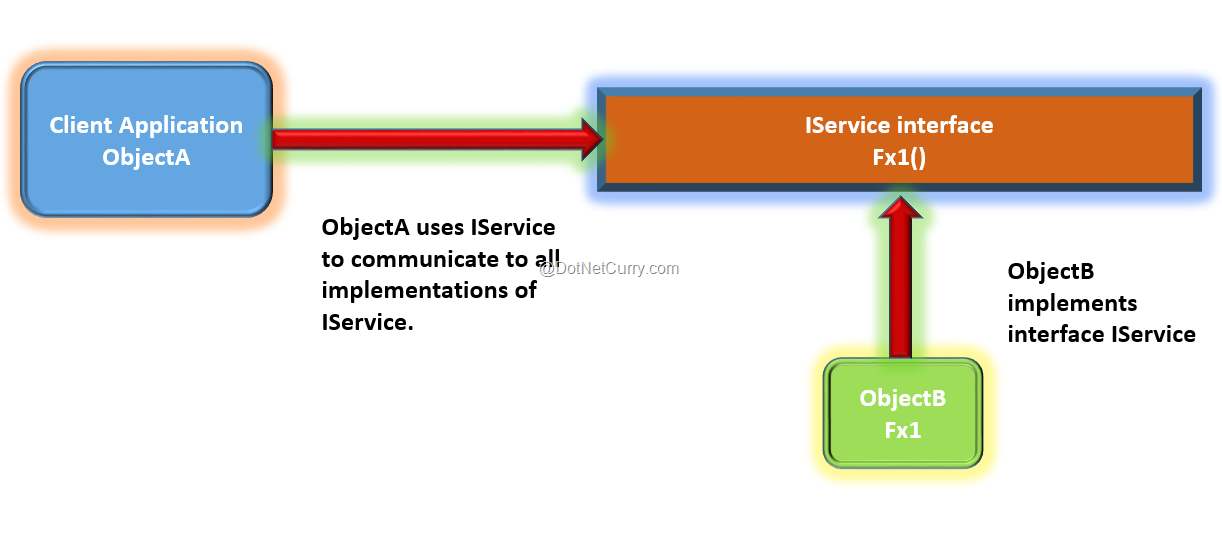
If we want a separate instance of a dependency to be shared across each instance of a component and only it’s view children we configure it on the components viewProviders property.

**What is Dependency Injection (DI)?**

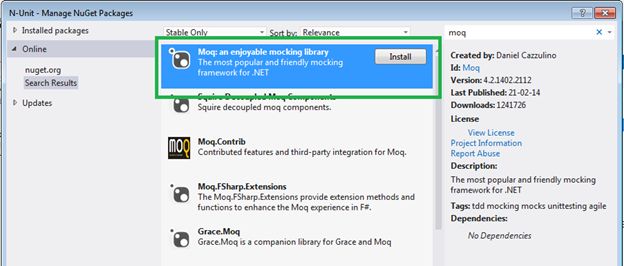
Technically, **Dependency Injection or DI** is defined as a software design pattern which implements Inversion of control (IOC) for resolving dependencies across objects. The dependency is an object (or a service object), which is passed as dependency to the consumer object (or a client application). This is a pattern using which decoupling (or loose-coupling) across components can be implemented easily.

When ObjectA wants to access methods (or operations) of ObjectB, then instead of directly instantiating ObjectB in ObjectA, ObjectB is *injected* into ObjectA as a dependency. In this case, ObjectB implements an interface, which is passed to ObjectA. Hence ObjectB is not instantiated using new in ObjectA. The advantage of this approach is that if in future ObjectB has any modifications, ObjectA need not be refreshed.

The following image (not a class diagram) provides an overview of the above scenario.



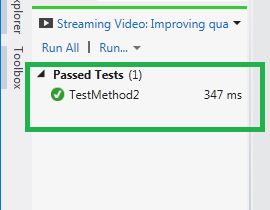
**Unit Test Mocking**

**What is mocking?**  
  
Let's think that one application is being develop and many developers are working in this project and each one is assign to develop a function. Let's think that I am developing a function that will insert one employee information into the DB; if it is not present in the DB then fine and one of my fellow developer is developing the function to check the existence.  
  
And I have completed my function but this guy has not, as he has a little bit of a workload, haha.. Now, as I completed my task, I wanted to test my function but for that I need to depend on the checking function that is still not developed.  
  
So, how I will do that? I need to create mock object that will bypass the checking function. The point to make here is that there are many mocking frameworks to implement the mock object. In this article we will use MOQ as a mocking framework.  
  
So, let's create one unit test application and pass this library as a reference of the application from the Nuget Package Manager.  
  
  
  
Here is our code that we will test using the unit test application.

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using System.Threading.Tasks;
7. namespace TestProjectLibrary
8. {
9. public class checkEmployee
10. {
11. public virtual Boolean checkEmp()
12. {
13. throw new NotImplementedException();
14. }
15. }
17. public class processEmployee
18. {
19. public Boolean insertEmployee(checkEmployee objtmp)
20. {
21. objtmp.checkEmp();
22. return true;
23. }
24. }
25. }

Now, see the implementation, the checkEmployee class contains a checkEmp() function that is still not implemented. And we are sending an object of the checkEmployee class to the insertEmployee() function to check whether the employee already exists before it is inserted into the DB.  
  
So, the concept is that since the checkEmployee class is not fully implemented , we will send a mock object of the checkEmployee class as an argument of the insertEmployee() function. Here is sample code of the implementation.

1. using System;
2. using Microsoft.VisualStudio.TestTools.UnitTesting;
3. using TestProjectLibrary;
4. using Moq;
6. namespace UnitTest
7. {
8. [TestClass]
9. public class UnitTest
10. {
11. [TestMethod]
12. public void TestMethod2()
13. {
14. Mock<checkEmployee> chk = new Mock<checkEmployee>();
15. chk.Setup(x => x.checkEmp()).Returns(true);
17. processEmployee obje = new processEmployee();
18. Assert.AreEqual(obje.insertEmployee(chk.Object), true);
19. }
20. }
21. }

Have a look at the first two lines of TestMethod2(). We are defining a mock object associated with checkCmployee class and in the next line we are setting the mock object.  
  
chk.Setup(x => x.checkEmp()).Returns(true);  
  
The preceding line is a bit interesting. Moq has a Setup() function by which we can set up the mock object. We need to use a lambda expression to point to a specific function. Here we are referring to the checkEmp() function and the Returns parameter value is true.  
  
This means that whenever the unit test application encounters the checkEmp() function it will always return true without executing it's code. So, ultimately, it will not execute at all and the result will be always true.  
  
Now, if we run the test then we will see it passes.  
  
  
  
**Conclusion**  
Mocking is very useful concept when the project is distributed among many team members. The fundamental idea behind mocking is to inject dependency and perform a unit test.

**Mocking Unit Testing Using Rhino**

## **Introduction**

Mocking is an integral part of unit testing. Although you can run your unit tests without use of mocking but it will drastically slow down the executing time of unit tests and also will be dependent on external resources. In this article we will explain mocking and different benefits that we can achieve by introducing mocking in our unit tests.

## **Real World Example**

We will focus on a real world example so that you will get a good grasp of mocking. Let’s say we are working on banking software which pulls out the customer balance information from some database. The software also makes a request to an external web service for authentication. The process of authentication is slow since it involves many different steps including discovering the web service, serializing the input and output parameters etc.

We want to test that if the customer balance returned is correct or not.

Here is our web service with the Authenticate method:

Hide   Copy Code

public class AverageJoeBankService : IAverageJoeBankService

{

public bool Authenticate(string userName, string password)

{

*// This is just simulate the time taken for the web service to authenticate!*

System.Threading.Thread.Sleep(5000);

return true;

}

}

As you can see, we are simply simulating the Authenticate method. We are assuming that it will take at least 5 seconds to authenticate the user. Of course in real application your Authenticate method will communicate with the real web service.

And here is the AccountBalanceService method:

Hide   Copy Code

public class AccountBalanceService

{

private IAverageJoeBankService \_averageJoeService;

public AccountBalanceService(IAverageJoeBankService averageJoeService)

{

\_averageJoeService = averageJoeService;

}

public double GetAccountBalanceByUser(User user)

{

*// the authenticate method below takes too much time!*

bool isAuthenticated = \_averageJoeService.Authenticate(user.UserName,

user.Password);

if (!isAuthenticated)

throw new SecurityException("User is not authenticated");

*// access database using username and get the balance*

return 100;

}

}

Here is our first attempt to write the unit tests:

Hide   Copy Code

[Test]

public void should\_be\_able\_to\_get\_the\_balance\_successfully\_without\_using\_mock\_objects()

{

User user = new User();

user.UserName = "johndoe";

user.Password = "johnpassword";

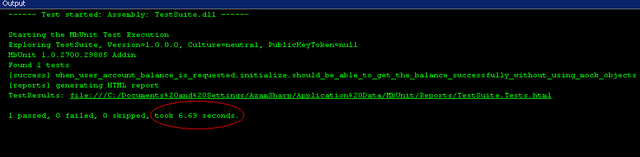
\_accountBalanceService = new AccountBalanceService(new AverageJoeBankService());

Assert.AreEqual(100, \_accountBalanceService.GetAccountBalanceByUser(user));

}

In the unit test above we are simply using the concrete implementation of the AccountBalanceService class which triggers the actual method for authentication.

When you run your tests you will get the following result:

[](https://www.codeproject.com/KB/tips/intro_to_mocking/IntroductionToMocking_Image001.PNG)

The test passed and you have a big smile on your face. Not so fast joy boy! Take a look at the time it took to run the test 6.69 seconds. This is too much time to run a single test. Apart from time there are other problems with the test. The test is dependent on the AverageJoeBankService. If the web service is not available then the test will fail. Unit tests should be independent and they should run fast. Let’s make this test faster by introducing mock objects.

Oh wait! We haven’t explained what mock objects mean in the context of unit tests. Mock objects are like real objects but they don’t do anything. We know the definition we just gave you is kind of crazy but you will know what we are talking about in a minute.

Here is the unit test which uses mock objects:

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

private AccountBalanceService \_accountBalanceService;

private MockRepository \_mocks;

[SetUp]

public void initialize()

{

\_mocks = new MockRepository();

}

[Test]

public void should\_be\_able\_to\_get\_the\_balance\_successfully()

{

User user = new User();

user.UserName = "JohnDoe";

user.Password = "JohnPassword";

var averageJoeService = \_mocks.DynamicMock<IAverageJoeBankService>();

\_accountBalanceService = new AccountBalanceService(averageJoeService);

using (\_mocks.Record())

{

SetupResult.For(averageJoeService.Authenticate(null,

null)).IgnoreArguments().Return(true);

}

using (\_mocks.Playback())

{

Assert.AreEqual(100,\_accountBalanceService.GetAccountBalanceByUser(user));

}

}

First, we created the MockRepository which is part of the Rhino Mocks frameworks. You might say “What the hell is Rhino Mocks?” Rhino mock is a mocking framework which provides you different mocking features and ways to mock your objects. There are several other mocking frameworks which include NMock, NMock2, TypeMock, MoQ.

Anyway, in the unit test above we are creating a mock object using the following code:

Hide   Copy Code

var averageJoeService = \_mocks.DynamicMock<IAverageJoeBankService>();

\_accountBalanceService = new AccountBalanceService(averageJoeService);

After passing the mocked AverageJoeService (not the real service) to the AccountBalanceService we put our expectations.

Hide   Copy Code

using (\_mocks.Record())

{

SetupResult.For(averageJoeService.Authenticate(null,

null)).IgnoreArguments().Return(true);

}

This means that when the averageJoeService.Authenticate method is fired then returns true so I can proceed further. You can also see that we are not concerned about the passed in arguments and that’s why we are using IgnoreArguments.

The next part is the Playback which is the code that will trigger the expectations and produce the desired result. Here is the Playback part:

Hide   Copy Code

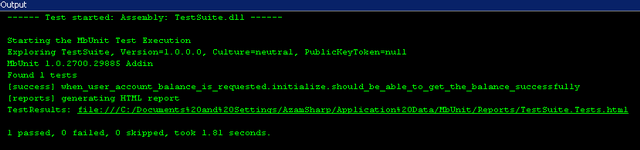
using (\_mocks.Playback())

{

Assert.AreEqual(100,\_accountBalanceService.GetAccountBalanceByUser(user));

}

Here is the output of the unit test:

[](https://www.codeproject.com/KB/tips/intro_to_mocking/IntroductionToMocking_Image002.PNG)

As you can see, now the unit test only takes 1.81 seconds. Not only is the test faster but now it is not dependent on the AverageJoeBankService. This means that even if the web service is down you will be able to run the above test.

## **Conclusion**

In this article we introduced the concepts behind Mocking. Mocking helps to improve unit tests by removing the outside dependencies which results in better, faster, independent unit tests.

We hope you liked the article, happy programming!

NOTE: We have also created a screencast on this subject which you can view using the following link: Screencast: [Introduction to Mocking](http://screencastaday.com/ScreenCasts/32_Introduction_to_Mocking.aspx)