

Angular Training
Steve Y Lin
2021/02



Agenda

- Develop Tools
- Project Architecture
- Angular Structure
- Component Introduction
- Data Binding
- Structural Directive
- Template Variables
- Component Communication
- Form
- Router and Navigate
- Other Schematic
- Library & UI Framework
- Atomic Design
- Guideline & Design Pattern
- Unit / E2E Test
- Gitlab CI
- Matomo

Develop Tools



npm

Open-source developers use npm to share software. npm is the world's largest Software Registry. The registry contains over 800,000 code packages.



Angular CLI

The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications.



git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.



VS Code

Visual Studio Code is a code editor redefined and optimized for building and debugging modern web and cloud applications.



Node.js

Node.js is an open source server environment. Node.js allows you to run JavaScript on the server.



Will寶哥 Extension

This extension pack packages some of the most popular (and some of my favorite) Angular extensions.

Develop Tools

Angular CLI 優點

1. 量身打造
2. 最好的Starter模板
3. 自動產生目錄結構及開發所需的檔案
4. 支援程式碼產生
5. 包含單元、整合測試
6. 程式碼最佳化
7. 統一開發體驗

- 新建專案

```
> ng new <專案名稱>
```

- 建立Component、Directive、Service、Pipe、Guard等Schematic

```
> ng g <schematic> <schematic名稱>
```

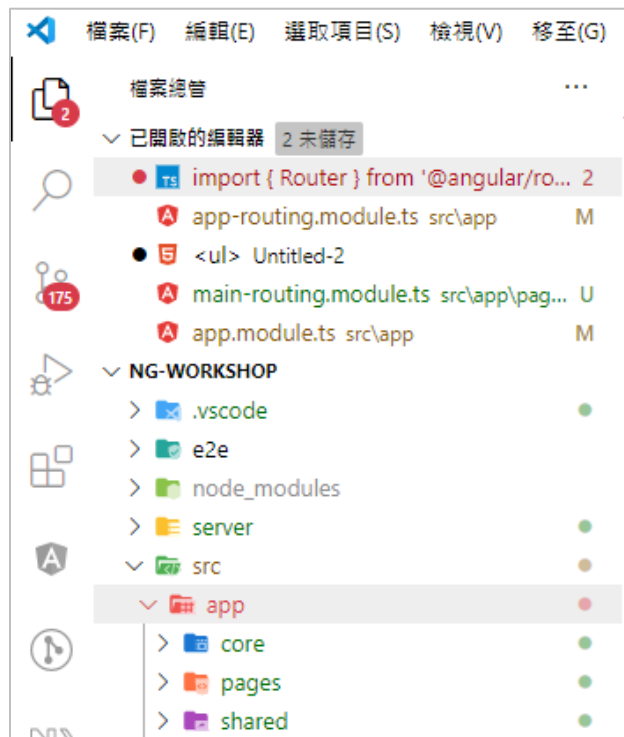
- 啟動Server

```
> ng serve
```

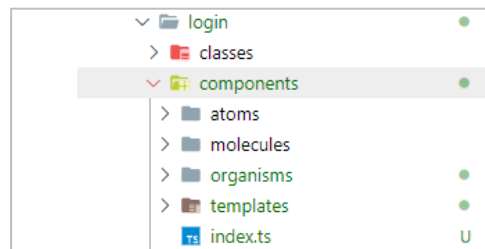
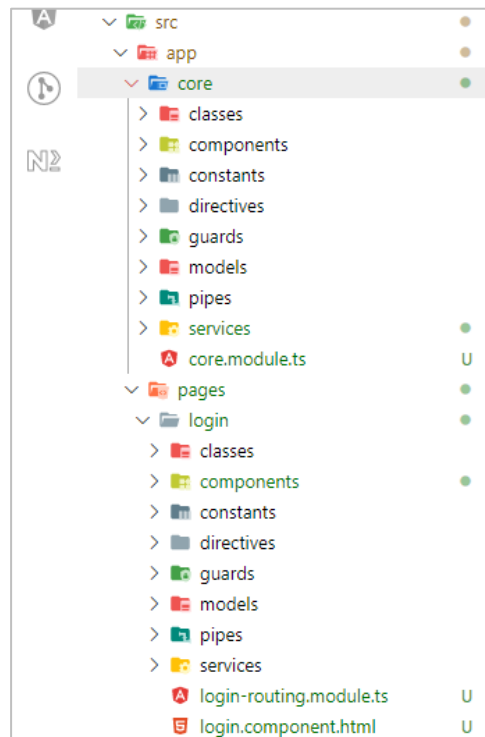
- 匯出專案




```
> ng build
```

Project Architecture



展開

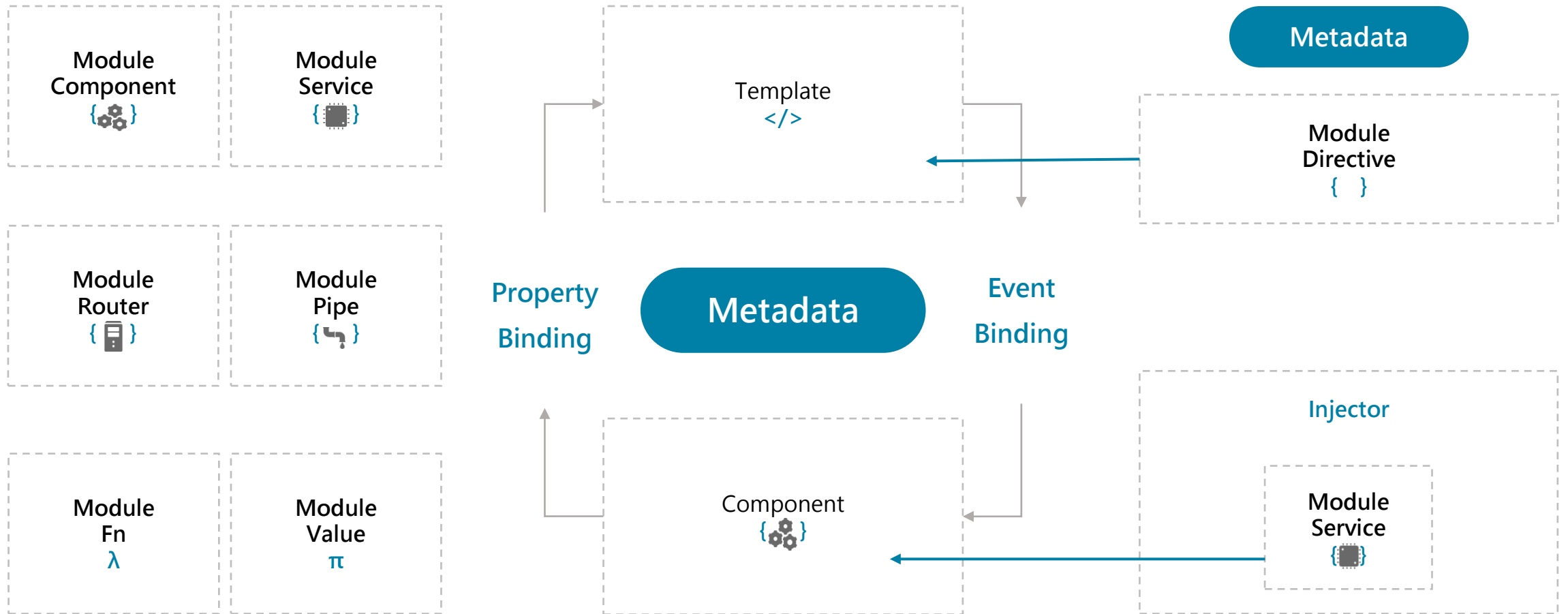


-  **core**: 共享單例資源
-  **shared**: 共享多個實例資源
-  **pages**: 依照頁面劃分module

-  **classes**: 供該Module使用的類別
-  **models**: 供該Module使用的介面或資料模型
-  **constants**: 供該Module使用的常數資源
-  **components**: 供該Module使用的元件
-  **services**: 供該Module使用的服務
-  **directive**: 供該Module使用的元件指令
-  **pipes**: 供該Module使用的管道
-  **guards**: 供該Module使用的路由守衛

使用Atomic Design的方式分類Component中的目錄，
待Atomic Design章節再進行詳細的分類基準

Angular Structure



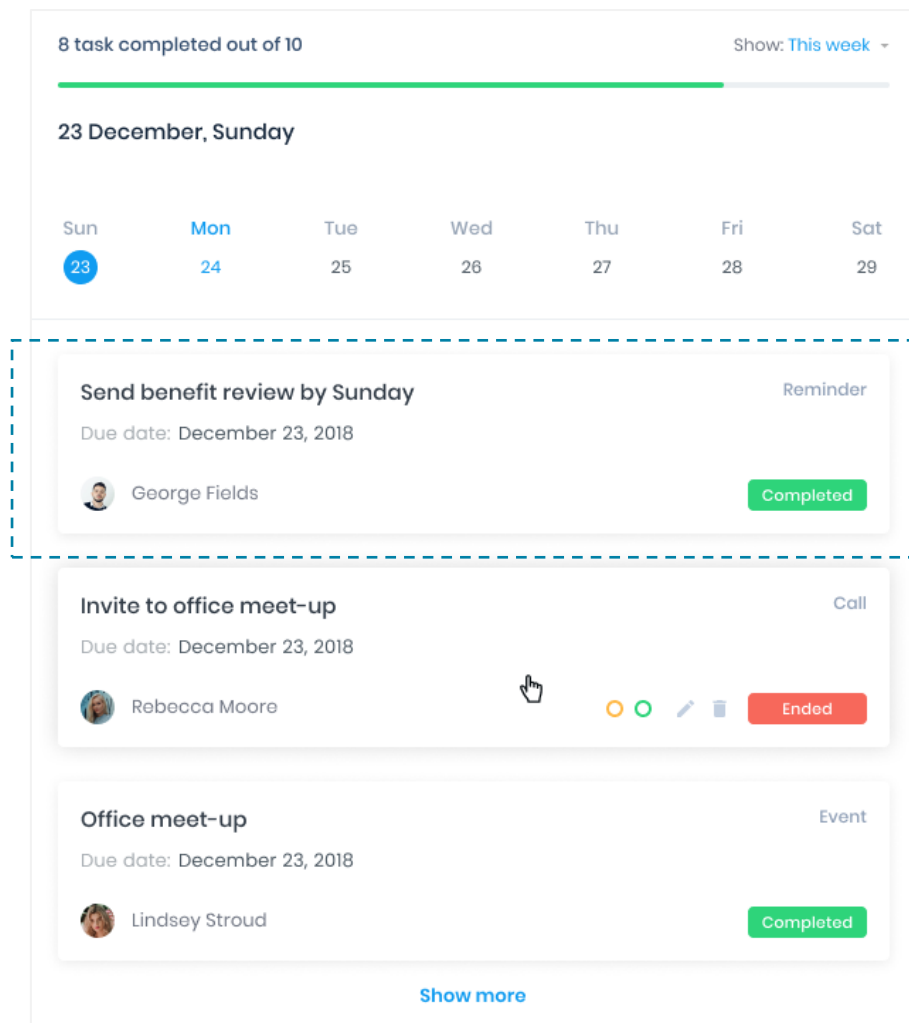
Component Introduction

何謂Component ?

- 最小單位
- 包裝HTML、CSS(LESS or SCSS)及Typescript
- 可重複使用
- 處理View的操作
- 反應物件狀態

元件化設計

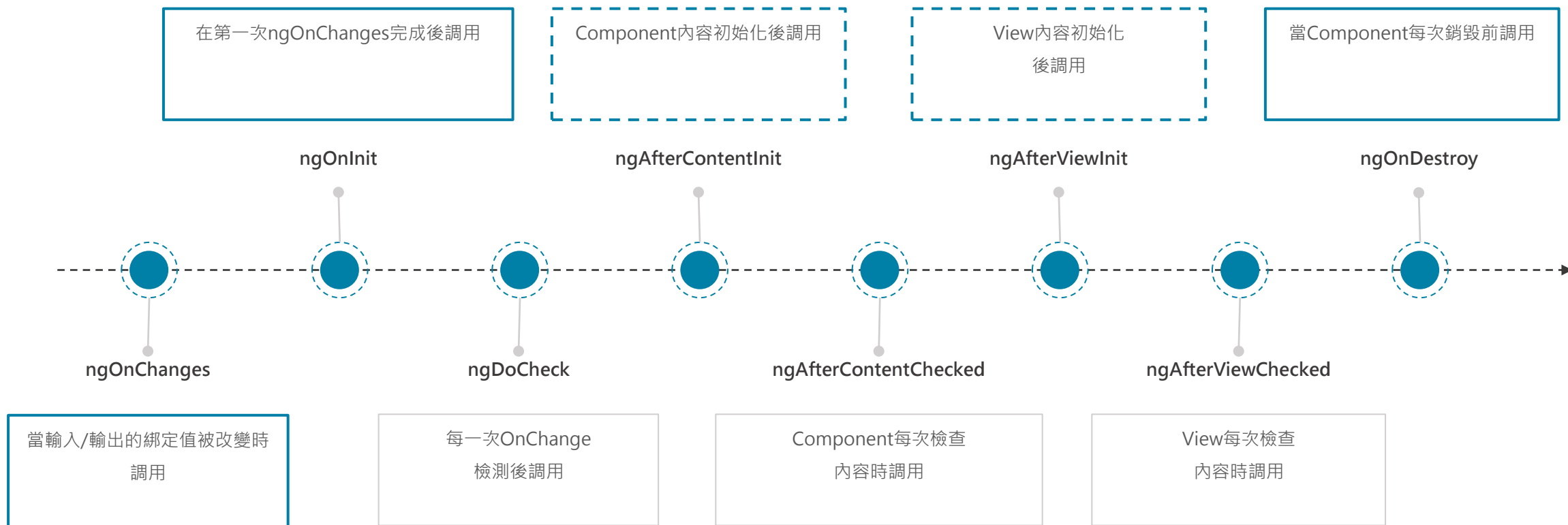
- 將畫面拆分成多個可重複使用的component
- 主要用來呈現資料
- 盡可能不處理運算方面的邏輯；僅處理View的呈現邏輯
- 所有的元件都由Component組合而成



Component Introduction

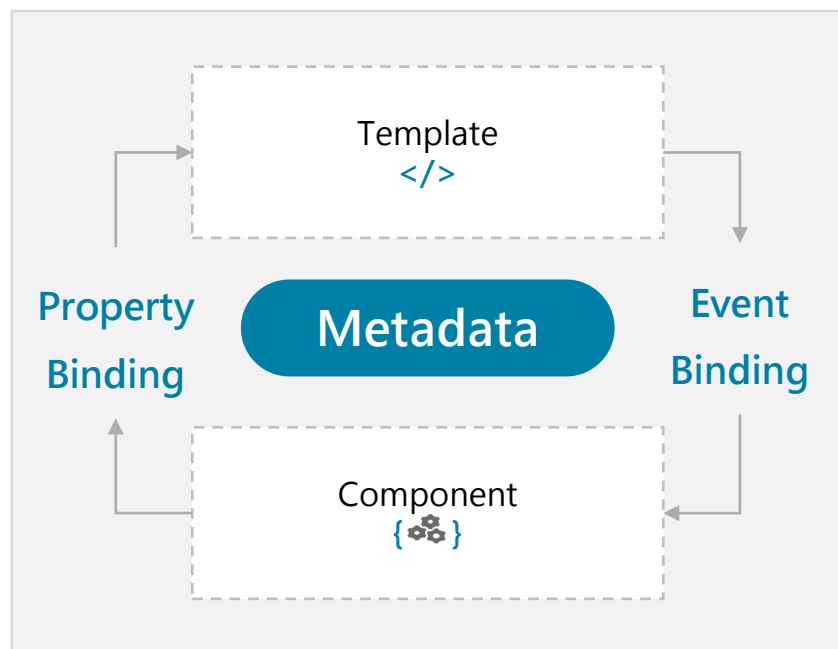
Component 生命週期

 常用  普通  較少



Data Binding

- 將資料呈現在畫面上
- 根據條件呈現
- 接收使用者輸入的資料
- 提供四種綁定方式插值、屬性綁定、事件綁定及雙向綁定



插值(Interpolation)

```
<span>{{ name }}</span>
```

屬性綁定(Property Binding)

```
<img [src]="imageSource" [attr.data-source]="imageSource" />
```

事件綁定(Event Binding)

```
<button (click)="onClickedButton('clicked')>Click Me</button>
```

雙向綁定(Two-way Binding)

```
<input type="text" [(ngModel)]="name" />
```

Structural Directive

結構型指令的職責是 HTML 佈局。它們塑造或重塑 DOM 的結構，比如新增、移除或維護這些元素。

注意：每個元素僅能使用一個結構指令

*ngIf

```
<div *ngIf="show">Show Me?</div>
```

*ngFor

```
<ul>  
  <li *ngFor="let item of items">{{ item.title }}</li>  
</ul>
```

ngSwitch

```
<p [ngSwitch]="">  
  <span *ngSwitchCase="true">TRUE</span>  
  <span *ngSwitchCase="false">FALSE</span>  
  <span *ngSwitchDefault>DEFAULT</span>  
</p>
```

Structural Directive

如果需要再*ngFor中，指定某個item不顯示，如何實作？

*ngFor

```
<ul>  
  <li *ngFor="let item of items" *ngIf="show">{{ item.title }}</li>  
</ul>
```

Wrong

*ngFor

```
<ul>  
  <li *ngFor="let item of items" [attr.hidden]="!show">{{ item.title }}</li>  
</ul>
```

Correct

雖然hidden可以將該element隱藏，但實際上該element依舊是**存在**，因此仍與*ngIf的結果有所差別

Template Variables

語法就是在任意的標籤裡面使用一個 # 字號加上一個變數名稱，如 `#name`

會在這個 Template 中建立一個名為 name 的區域變數，可以透過事件繫結將任意 DOM 物件傳回元件類別中

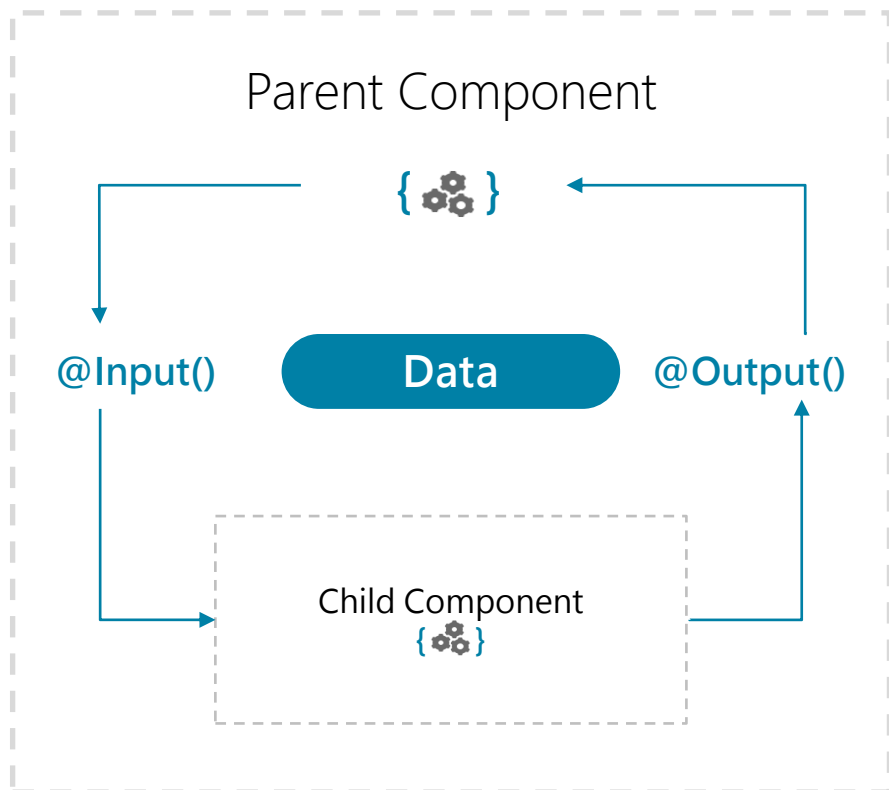
app.component.html

```
1 <input type="text" #demo>
2 <input type="button" (click)="send(demo)">
```

app.component.ts

```
1 export class AppComponent implements OnInit {
2     send(demo: HTMLInputElement): void {
3         console.log(demo);
4         console.log(demo.value);
5     }
6 }
```

Component Communication



- 透過組合Component完成功能
- 彼此使用Property和Event溝通
- 最外層的Component才有資料
 - Controller View
 - Child Component 負責顯示資料和觸發事件
- 讓元件更容易重複使用
- 僅讓Component進行垂直溝通，水平溝通請參照Service



Component Communication

`@Input()` 的目的是確認資料綁定 input 特性 (properties) 是可以在改變時被追蹤的。基本上來說，它是 Angular 將 DOM 藉由特性綁定 (property bindings) 直接在組件注入數值的方法。

app.component.html

```
1 | <app-editor [count]="100"></app-editor>
```

editor.component.ts

```
1 | import { Input, ... } from '@angular/core';
2 |
3 | ...
4 | export class EditorComponent implements OnInit {
5 |     @Input() public count: number;
6 |     constructor() {}
7 |     ...
8 | }
```



Component Communication

`@Output()` 是用來觸發組件的客製化事件並提供一個通道讓組件之間彼此溝通。

app.component.html

```
1 | <app-status (statusChanged)="getStatus($event)"></app-status>
```

status.component.ts

```
1 | import { Output, EventEmitter, ... } from '@angular/core';
2 |
3 | ...
4 | export class StatusComponent implements OnInit {
5 |     @Output() public statusChanged = new EventEmitter<boolean>();
6 |     constructor() {}
7 |     public onStatusChanged(status: boolean) {
8 |         this.statusChanged.emit(status);
9 |     }
10 | }
```

Component Communication


8 task completed out of 10 Show: This week ▾

23 December, Sunday

Sun	Mon	Tue	Wed	Thu	Fri	Sat
23	24	25	26	27	28	29

Send benefit review by Sunday

Due date: December 23, 2018


 George Fields

Reminder


Completed

Invite to office meet-up

Due date: December 23, 2018

 Rebecca Moore


Call



Ended

Office meet-up

Due date: December 23, 2018

 Lindsey Stroud

Event


Completed

Show more


How to design ?

Invite to office meet-up

Due date: December 23, 2018

 Rebecca Moore

Call



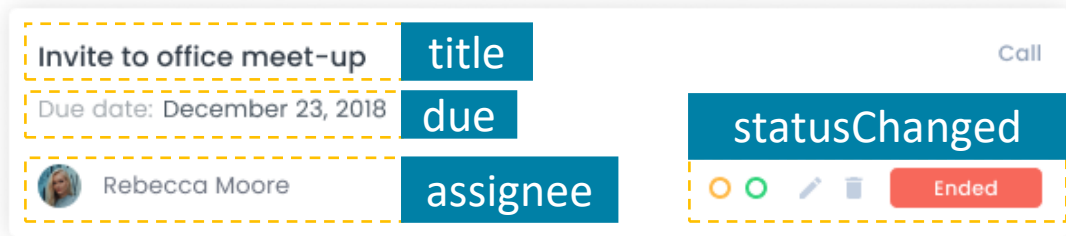
Ended

- Try to list `@Input()` properties
- Try to list `@Output()` events



Component Communication

1 Design components



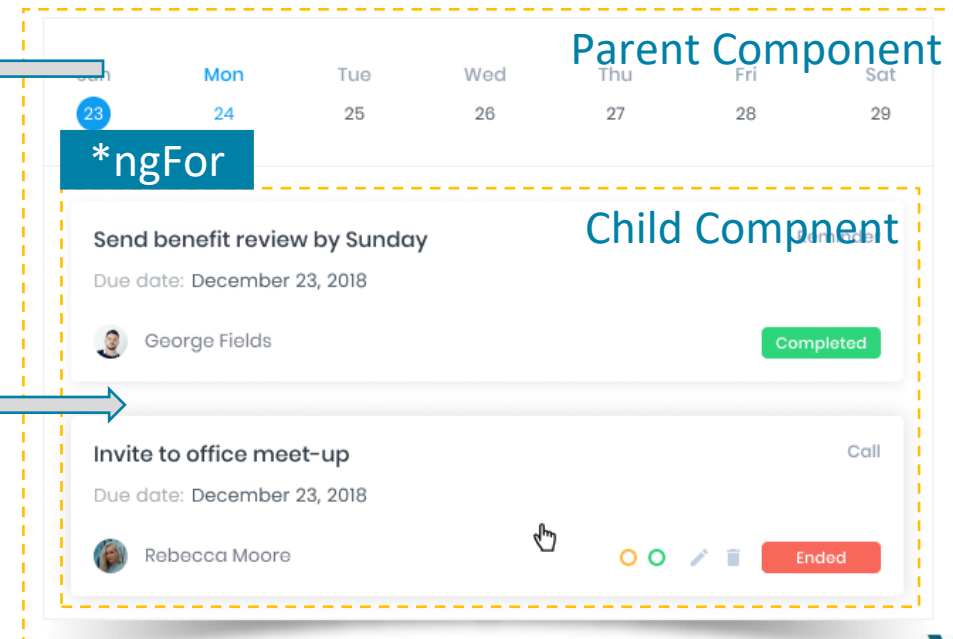
- @Input(): title, due, status, assignee, avatar
- @Output(): statusChanged

2 Collect Data

```
1 [
2   {
3     "title": "title1",
4     "due": 1594861891000,
5     "status": "completed",
6     "assignee": "assignee1",
7     "avatar": "avatar1"
8   },
9   ...
10 ]
```

若資料需要額外處理
請參考service

3 Render into View



Component Communication



Advantage

1. 提高元件的重用性
2. 避免元件會因需要多附屬在某個父元件下，而在衍生更多的方法
3. 邏輯單純，讓每個元件都是單一職責(SRP)



Disadvantage

1. 流程會變得繁瑣
2. 顆粒度為分子以上的元件，需重新實作子元件的屬性或方法

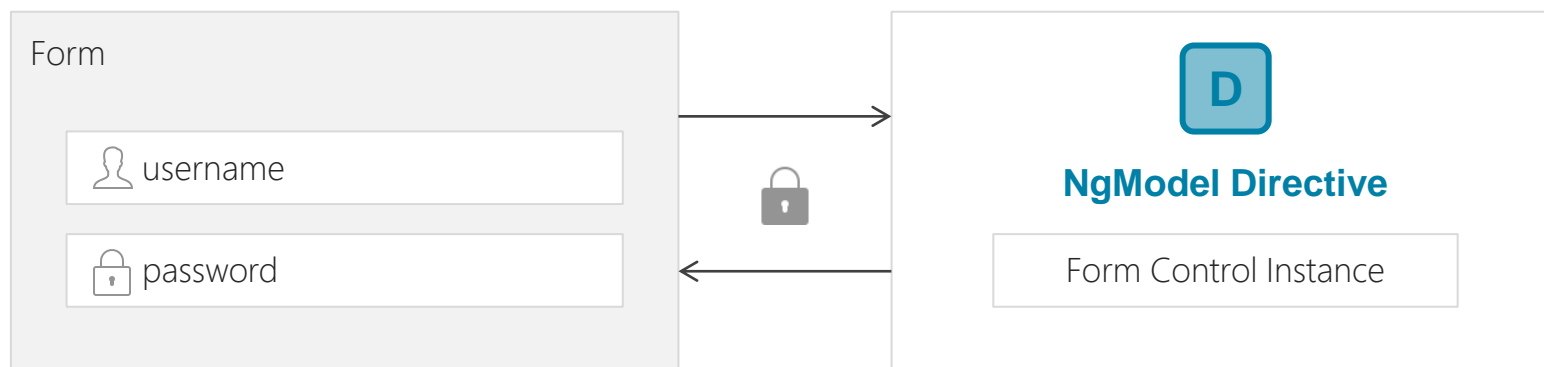
整體而言 “Z > B”

Form

Template Driven Form

- 適合時做簡易的表單
- 表單所需的元件都宣告在HTML中
- Import FormsModule

僅透過NgModel這個Directive來存取FormControl的實例

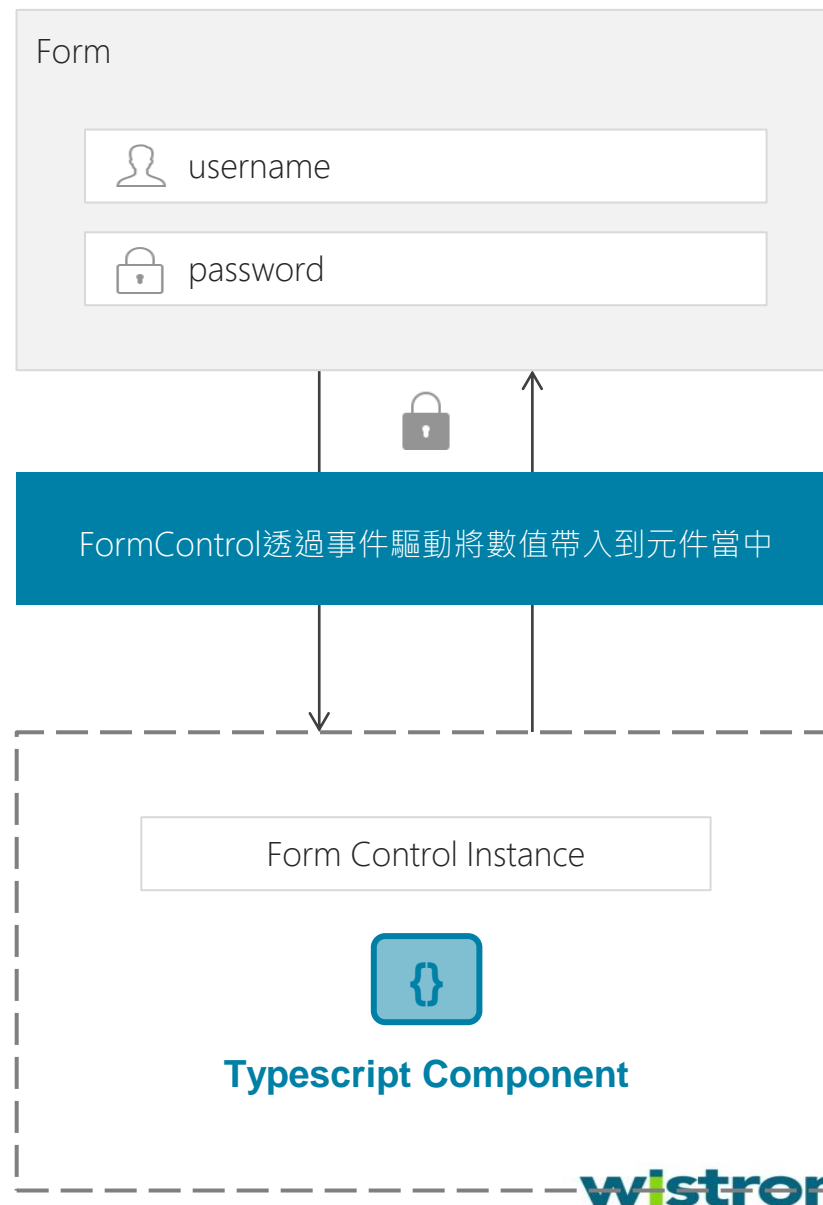


```
1 <form #myForm="ngForm">
2   <label>帳號</label>
3   <span *ngIf="account.touched && account.errors">
4     <code *ngIf="account.errors.required">必填</code>
5   </span>
6
7   <input type="text" #account="ngModel" ngModel name="account" required>
8   <input type="submit" value="Submit" [disabled]="!myForm.valid">
9 </form>
```

Form Reactive Form

- 適合時做複雜的表單
- 使用屬性綁定對表單進行宣告，表單的宣告是建立formGroup屬性，而針對控制項，則是建formControlName
- Import ReactiveFormsModule

```
1  import { FormGroup, FormControl, Validators } from '@angular/forms';
2
3  ...
4  export class SignUpComponent {
5    public signUpForm = new FormGroup({
6      name: new FormControl('', Validators.required),
7      age: new FormControl(18, Validators.required),
8      contact: new FormGroup({
9        email: new FormControl('', Validators.required),
10       mobile: new FormControl('', Validators.pattern(/^[0-9]{8}$/)),
11     })
12   });
13 }
```



Form

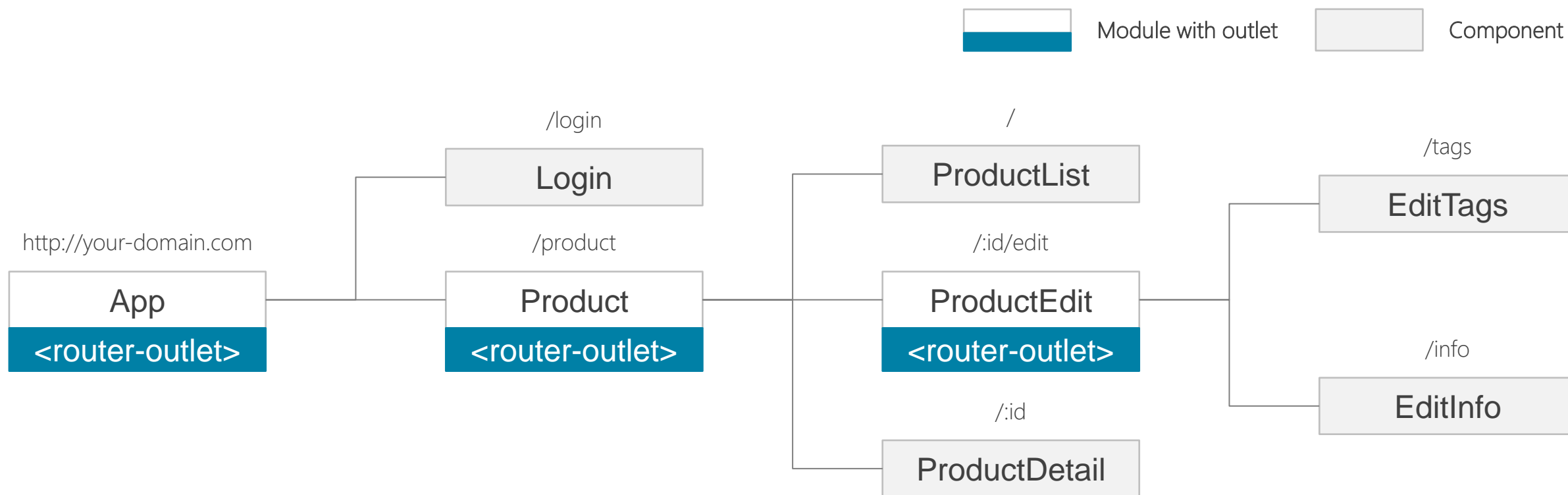
Validators List

1. `min` : 限制最小值
2. `max` : 限制最大值
3. `required` : 必填
4. `requiredTrue` : 必須為True
5. `email` : 符合email格式
6. `minLength` : 最短長度
7. `maxLength` : 最常長度
8. `pattern` : 使用RegExp自訂驗證格式
9. `nullValidator` : 不為NULL
10. `compose` : 將多個驗證器合成一個

```
1  const Control = new FormControl(16, Validators.max(15));
2  const control = new FormControl(14, Validators.min(15));
3  const control = new FormControl('', Validators.required);
4  const control = new FormControl(false, Validators.requiredTrue);
5  const control = new FormControl('bad@', Validators.email);
6  const control = new FormControl('ng', Validators.minLength(3));
7  const control = new FormControl('Angular', Validators.maxLength(5));
8  const control = new FormControl('abc', Validators.pattern('[a-zA-Z ]*'));
9  const control = new FormControl(null, Validators.nullValidator);
10 const control = new FormControl(0, Validators.compose([
11     Validators.required,
12     Validators.min(15)
13 ]));
```

Router and Navigate

在單頁面應用中，你可以透過顯示或隱藏特定元件的顯示部分來改變使用者能看到的內容，而不用去伺服器獲取新頁面。當用戶執行應用任務時，他們要在你預定義的不同檢視之間移動。要想在應用的單個頁面中實現這種導航，你可以使用 Router。



Router and Navigate

路由Parameters介紹

- **path**: 路由的路徑，path為""時，表示路徑為空時匹配該路由；值為** 時所有路徑都匹配不到時匹配該路由，**路徑有順序性，符合就導向該路由**。
- **redirectTo**: 重新導向哪個路徑
- **component**: 路由中會導入哪個元件
- **pathMatch**: 設置路由的匹配規則，**full** 表示完全匹配；**prefix** 表示只匹配前綴，當path值為home時，/home、/home/123, 都能匹配到該路由。
- **children**: 子路由的設定
- **canActivate**: 驗證路由是否可存取，待Guard做詳細介紹

```
1  import { Routes } from '@angular/router';
2  const routes: Routes = [
3    { path: '', redirectTo: '/login', pathMatch: 'full' },
4    { path: 'login', component: Login, pathMatch: 'full' },
5    {
6      path: 'dashboard',
7      component: Dashboard,
8      children: [
9        { path: 'profile/:id', component: Profile, pathMatch: 'full' },
10       { path: 'setting', component: Setting, pathMatch: 'full' },
11      ]
12    },
13    { path: '**', component: NotFound, pathMatch: 'full' }
14  ]
15  ...
```

Router and Navigate

路由Navigate範例

- 使用HTML

```
1  <ul>
2    <li>
3      <a [[routerLink]="['/login']" routerLinkActive="router-link-active" ]></a>
4    </li>
5  </ul>
```

- 使用Typescript

```
1  import { Router } from '@angular/router';
2
3  ...
4  export class AppComponent {
5    constructor(private readonly router: Router) {}
6    public async navigate() {
7      await this.router.navigate(['/dashboard']);
8      // TODO
9    }
10 }
```


Router and Navigate

Lazy Loading 惰性載入

Advanced進階用法

Lazy loading 是指從一個資料物件通過方法獲得裡面的一個屬性物件時，這個對應物件實際並沒有隨其父資料物件建立時一起儲存在執行空間中，而是在其讀取方法第一次被呼叫時才從其他資料來源中載入到執行空間中，這樣可以避免過早地匯入過大的資料物件但並沒有使用的空間占用浪費。

app-routing.module.ts

```
1  const routes: Routes = [  
2    { path: '', pathMatch: 'full', redirectTo: '/main' },  
3    {  
4      path: 'main',  
5      loadChildren: () => import('./pages/main/main.module')  
6        .then(m => m.MainModule)  
7    }  
8  ];  
9  
10 @NgModule({  
11   imports: [RouterModule.forRoot(routes)],  
12   exports: [RouterModule]  
13 })  
14 export class AppRoutingModule { }
```

Router and Navigate

Nested Lazy Loading 巢狀惰性載入

Advanced進階用法

使用巢狀惰性載入在import路由Module時，須使用 `forChild()` 方法代入路由的路徑參數。

main-routing.module.ts

```
1  const routes: Routes = [
2    {
3      path: '',
4      component: MainComponent,
5      children: [
6        { path: '', pathMatch: 'full', redirectTo: '/welcome' },
7        {
8          path: 'welcome',
9          loadChildren: () => import('../welcome/welcome.module')
10           .then(m => m.WelcomeModule)
11        }
12      ]
13    },
14  ];
15
16 @NgModule({
17   imports: [RouterModule.forChild(routes)],
18   exports: [RouterModule]
19 })
20 export class MainRoutingModule { }
```

Router and Navigate

路由Data及Resolve

Advanced進階用法

在進入路由前，可以直接設定data或是resolve的屬性，讓資料可以先行預載，而這兩個屬性都可以透過ActivatedRoute的data取得。

login-routing.module.ts

```
1  const routes: Routes = [  
2    {  
3      path: '',  
4      component: LoginComponent,  
5      data: { test: 123 },  
6      resolve: {  
7        configs: LoginTestService  
8      }  
9    },  
10  ];
```

login-routing.module.ts

```
1  ...  
2  export class LoginComponent implements OnInit {  
3    constructor(private readonly activatedRoute: ActivatedRoute) { }  
4    public ngOnInit(): void {  
5      this.activatedRoute.data  
6        // Output: {"test":123,"configs": ...}  
7        .subscribe(data => this.logger.data(data));  
8    }  
9  }
```

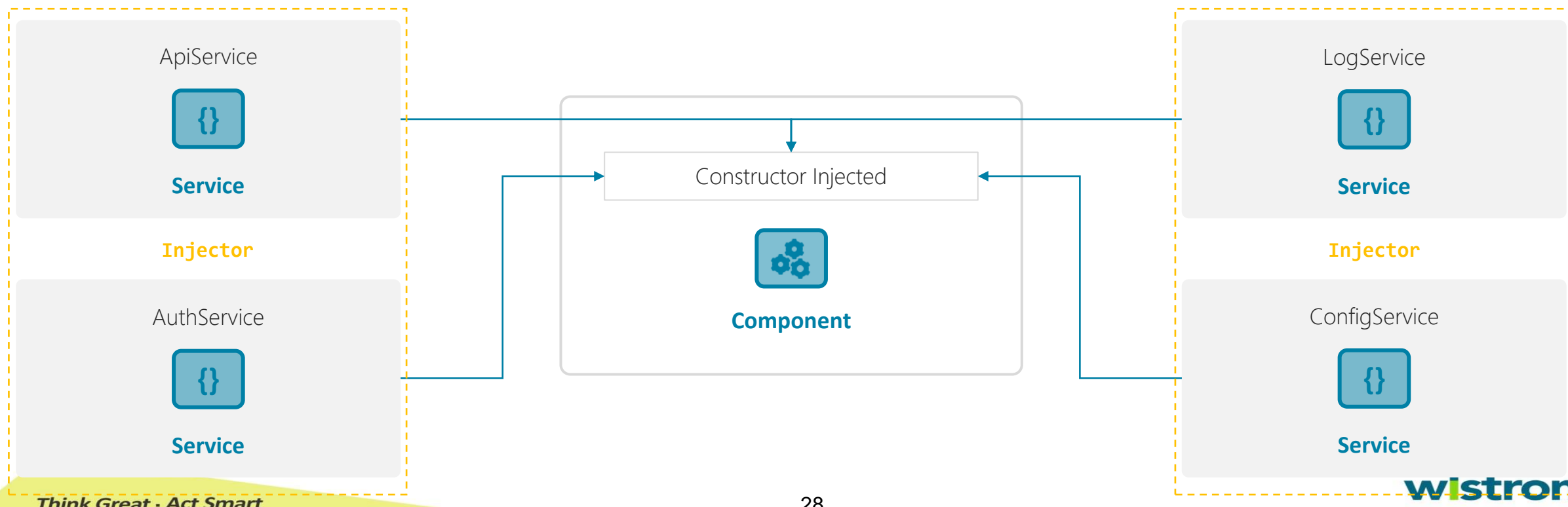
login-test.service.ts

```
1  import { HttpClient } from '@angular/common/http';  
2  import { Injectable } from '@angular/core';  
3  import { Resolve } from '@angular/router';  
4  import { Conifg } from 'src/app/shared/models';  
5  
6  @Injectable()  
7  export class LoginTestService implements Resolve<Conifg> {  
8    constructor(private readonly http: HttpClient) { }  
9    public resolve(): Promise<any> {  
10      return this.http  
11        .get(' ../../../../configs/config.json')  
12        .toPromise();  
13    }  
14  }
```

Other Schematic

Service

服務是一個廣義的概念，它包括應用所需的任何值、函式或特性。狹義的服務是一個明確定義了用途的類別。它應該做一些具體的事，並做好，如**伺服器獲取資料(API, HttpClient)**、**驗證使用者輸入或直接往控制檯中寫日誌等工作委託給各種服務等商業邏輯**。把元件和服務區分開，以提高模組性和複用性。



Other Schematic Service

providedIn使用 **root** 指定 Angular 應該在根注入器中提供該服務，此 UserService 就會是單例的，並在整個應用中都可用

user.service.ts

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root',
5 })
6 export class UserService {
7 }
```

user-list.component.ts

```
1 ...
2 export class UserListComponent {
3   constructor(private readonly userService: UserService) {}
4   public async findUser(id: string): Promise<User> {
5     return await this.userService.findOne(id);
6   }
7 }
```

若想要讓服務可以做用在特定Scope，也可以使用以下方式來達成，讓服務分開產生實例

```
1 @NgModule({
2   providers: [
3     UserService,
4   ],
5   ...
6 })
```

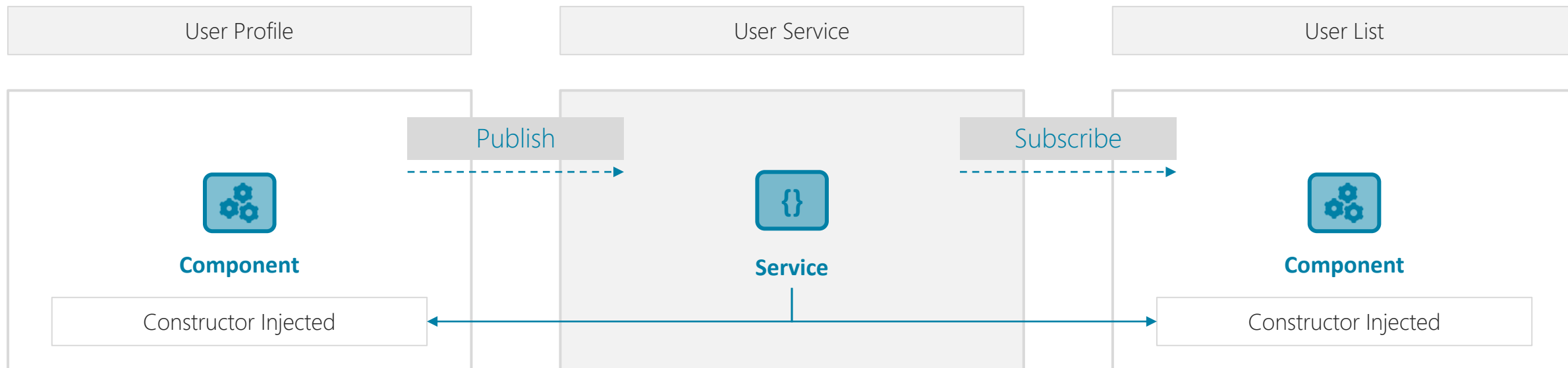
在特定Module產生實例

```
1 @Component({
2   selector: 'app-user-list',
3   templateUrl: './user-list.component.html',
4   providers: [UserService]
5 })
6 ...
```

在特定元件產生實例

Other Schematic

由於服務可以有單例的特性，因此也可以透過此方式讓元件間建立一個水平溝通的管道，並且加上RxJS中的Observable及Subject後，能夠讓多個元件間進行數據的訂閱以及推播。



Other Schematic Service

實際應用範例

user-profile.component.ts

```
1  ...
2  export class UserProfileComponent {
3      constructor(private readonly userService: UserService) {}
4      public editUserName(name: string): void {
5          this.userService.user$.next(name);
6      }
7  }
```

.next() Publish Data

user-list.component.ts

```
1  ...
2  export class UserListComponent implements OnInit {
3      constructor(private readonly userService: UserService) {}
4      public ngOnInit(): void {
5          this.userService.user$
6              .asObservable()
7              .subscribe(name => console.log(name));
8      }
9  }
```

.subscribe() Subscribe Data

user.service.ts

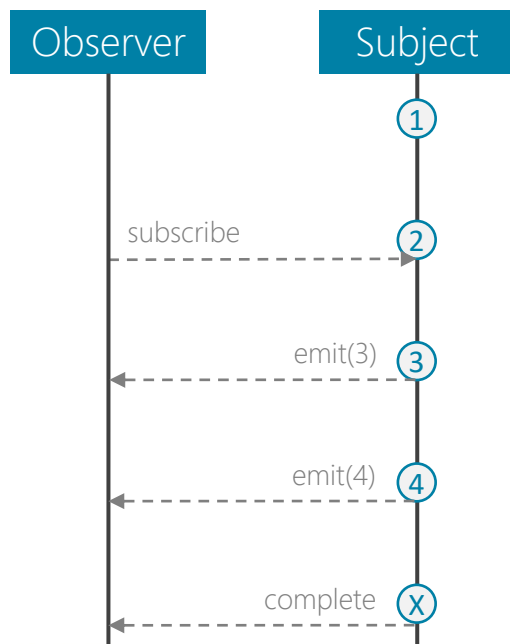
```
1  @Injectable({
2      providedIn: 'root'
3  })
4  export class UserService {
5      public user$ = new Subject<string>();
6  }
```

Other Schematic Service

Subject列表

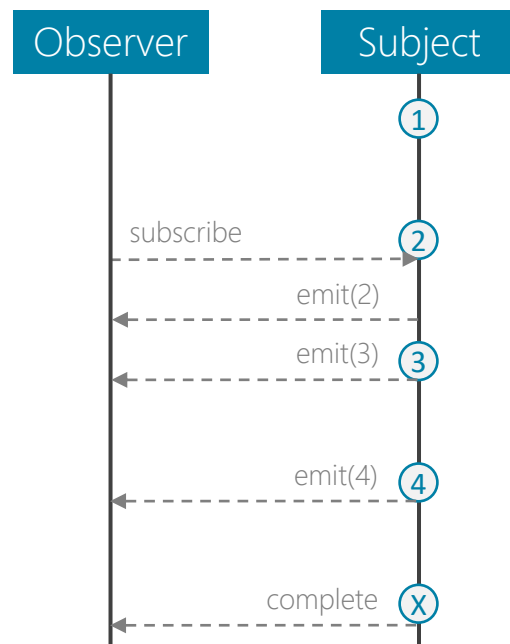
1 Subject

內部有一份 observer 的清單，並在接收到值時遍歷這份清單並送出值



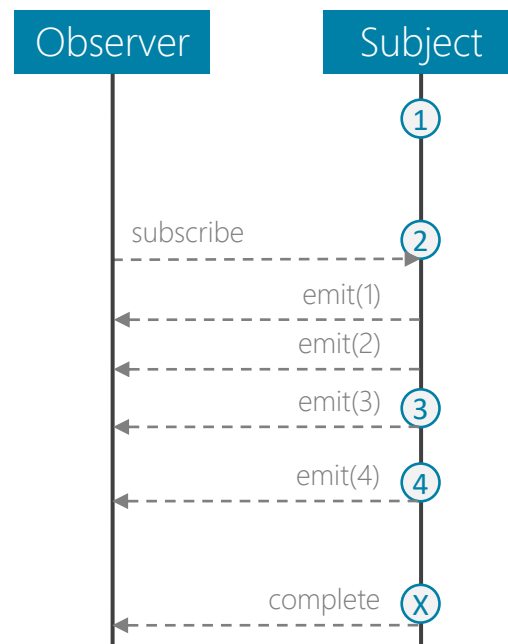
2 BehaviorSubject

新的訂閱產生時，希望 Subject 能立即給出最新的值，而不是沒有回應



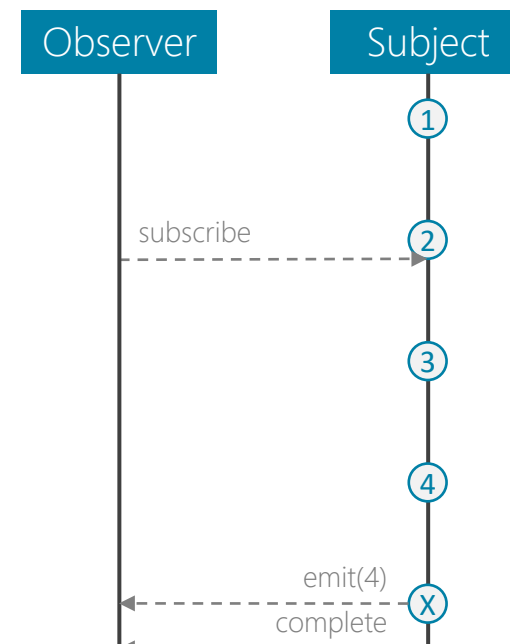
3 ReplaySubject

新的訂閱產生時，希望Subject能重新發送前幾個元素



4 AsyncSubject

Subject complete時，發送最後一元素給訂閱者



Other Schematic

Pipe

管道用來對字串、貨幣金額、日期和其他顯示資料進行轉換和格式化。管道是一些簡單的函式，可以在範本表示式中用來接受輸入值並返回一個轉換後的值。例如，你可以使用一個把epochtime日期顯示為 1988 年 4 月 15 日。

Angular內建Pipe可參考 <https://angular.io/api?type=pipe>

1	<code><!-- Output: 002.71828 --></code>
2	<code><p>e (3.1-5): {{ e number:'3.1-5' }}</p></code>
3	
4	<code><!-- Output: \$0.26 --></code>
5	<code><p>price: {{ price currency }}</p></code>
6	
7	<code><!-- Output: (6/15/15, 9:03AM) --></code>
8	<code><p>The time is {{ now date:'short' }}</p></code>

Other Schematic

Pipe

Custom Pipe

在自訂管道類別中，使用@Pipe()裝飾器，並實現 PipeTransform 介面來執行轉換。Angular 呼叫 transform 方法，該方法使用繫結的值作為第一個引數，把其它任何引數都以列表的形式作為第二個引數，並返回轉換後的值。

my-date.pipe.ts

```
1  @Pipe({
2    name: 'myDate'
3  })
4  export class MyDatePipe implements PipeTransform {
5    transform(value: number, format?: string): string {
6      return new MyDate(value).toDate(format);
7    }
8  }
```

app.component.ts

```
1  <!-- Output: 2020/01/01 00:00:00 -->
2  <p>timestamp: {{ timestamp: myDate:'yyyy/MM/dd hh:mm:ss' }}</p>
```

Other Schematic

Pipe

實際應用範例

利用自訂的Pipe，顯示代辦事項的完成狀態

```
1  @Pipe({
2    name: 'todo'
3  })
4  export class TodoPipe implements PipeTransform {
5    transform(done: boolean, date?: Date): string {
6      if (done) {
7        return `(done) timestamp: ${date.toLocaleDateString()}`;
8      } else {
9        return '(incomplete)';
10     }
11   }
12 }
```

- Task A | Delete (done) timestamp: 2020/01/01 08:00:00
- Task B | Delete (done) timestamp: 2020/01/01 09:00:00
- ☐ Task B | Delete (incomplete)
- ☐ Task B | Delete (incomplete)
- ☐ Task B | Delete (incomplete)

使用Pipe除了可以將資料轉換作為重用的方式外，以往針對在HTML中因應狀態顯示特定文字的邏輯，也可以抽離出來，降低HTML的複雜度

Other Schematic Directive

透過directive我們可以擴充原有的屬性(attribute)，來達到增加原有的DOM element、Component甚至其他的directive沒有的**功能**或**樣式**，並將這些功能或樣式抽離出來作為共享。使用@Directive ()裝飾器。

highlight.directive.ts

```
1  import { Directive, OnInit, Input, ElementRef } from '@angular/core';
2
3  @Directive({
4    selector: '[appHighlight]'
5  })
6  export class HighlightDirective implements OnInit {
7    @Input() public color = 'yellow';
8    constructor(private readonly elementRef: ElementRef) {}
9    public ngOnInit(): void {
10      this.elementRef.nativeElement.style.backgroundColor = this.color;
11    }
12  }
```

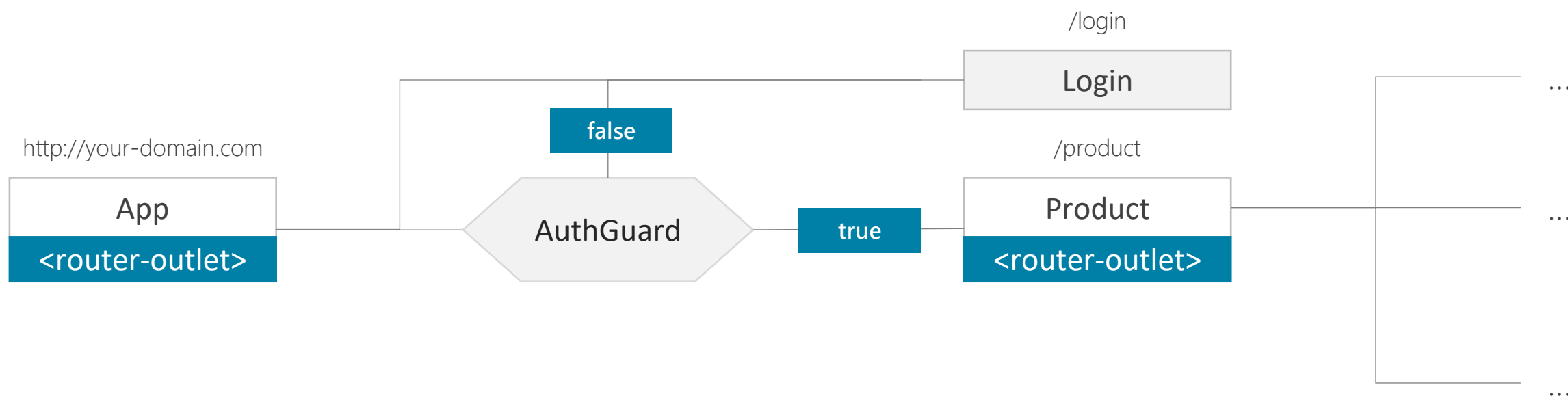
app.component.ts

```
1  <p appHighlight color="red">Highlight Element</p>
```

Other Schematic

Guard

通常我們會希望限制某些網址只有某特定規則者才可以進入或離開，Angular的Router提供了一系列的個方法來幫助我們而Guard則為訂製這些規則的Schematic，可以讓我們來決定是否允許使用者進入或是離開頁面。使用@Injectable ()裝飾器。



Other Schematic



```
1 import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
2 import { Injectable } from '@angular/core';
3
4 @Injectable({ providedIn: 'root' })
5 export class AuthGuard implements CanActivate {
6   constructor(private readonly router: Router, private readonly authApi: AuthApi) {}
7   public async canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<boolean> {
8     const isAuthenticated = await this.authApi.valid();
9     if (isAuthenticated) {
10       return true;
11     } else {
12       this.router.navigate(['/login']);
13       return false;
14     }
15   }
16 }
```

```
1 import { Routes } from '@angular/router';
2 const routes: Routes = [
3   ...,
4   { path: 'dashboard', component: Dashboard, canActivate: [AuthGuard] },
5 ]
6 ...
```

Other Schematic Module

一個 Module (模組) 是把彼此互相關聯的 components、directives、pipes 和 services 整合的機制。然後這個模組可以再和其他模組結合。使用@NgModule()裝飾器。

```
1  @NgModule({  
2    imports: [  
3      ...  
4    ],  
5    declarations: [  
6      ...  
7    ],  
8    exports: [  
9      ...  
10   ],  
11   providers: [  
12     ...  
13   ]  
14 })  
15 export class SharedModule { }
```

- **imports:** 在這個module下，需要匯入的module，提供其中的components、directives和pipes使用。
- **declarations:** 屬於這module的components、directives和pipes，讓module內部可相互使用這些資源。
- **exports:** 公開給外部使用的module中的類別。
- **providers:** 提供services，讓service在該module下產生該直屬於該modul的instance。

Other Schematic Module

Advanced進階用法

依賴提供者會使用 DI 令牌來配置注入器，注入器會用它來提供這個依賴值的具體的、執行時版本。注入器依靠 "提供者配置" 來建立依賴的實例，並把該實例注入到元件、指令、管道和其它服務中。

值提供者

提供一個現成的物件或常數，請使用 `useValue` 選項來配置該注入器。

```
1 @NgModule({
2   providers: [
3     {
4       provide: NZ_I18N,
5       useValue: zh_TW
6     }
7   ],
8 })
```

替代類別提供者

不同的類別都可用於提供相同的服務，最常用的是 `useClass` 方法，代表的是使用某個類別。

```
1 @NgModule({
2   providers: [
3     {
4       provide: LogService,
5       useClass: NewLogService
6     }
7   ],
8 })
```

工廠提供者

有時候你需要動態建立依賴值，建立時需要的資訊你要等執行期間才能拿到，可以使用 `useFactory` 搭配 `deps` 將相依的服務代入

```
1 @NgModule({
2   providers: [
3     {
4       provide: APP_CONF,
5       deps: [HttpClient],
6       useFactory: (http: HttpClient) => http
7         .get('../.../configs/config.json')
8         .toPromise()
9     }
10  ],
11 })
```


Other Schematic Module

Advanced進階用法

在分割模組的時候有一點要注意，SharedModule 經常會匯入到各個子模組中，應該盡量避免在 SharedModule 中建立 App 層級的服務，因為這樣會讓每個子模組都產生自己的單一實例服務(Singleton Service)，造成 App 中有多個一樣的單一實例服務，而可能引發問題（若是刻意要這樣做，則不在此考慮範圍內）。

shared.module.ts

```
1  import { NgModule, ModuleWithProviders } from '@angular/core';
2  ...
3
4  @NgModule({
5    imports: [],
6    declarations: [MySharedComponent],
7    exports: [MySharedComponent]
8  })
9  export class SharedModule {
10    public static forRoot(): ModuleWithProviders {
11      return {
12        ngModule: SharedModule,
13        providers: [MyService]
14      };
15    }
16  }
```

app.module.ts

```
1  @NgModule({
2    declarations: [
3      AppComponent
4    ],
5    imports: [
6      BrowserModule,
7      SharedModule.forRoot()
8    ],
9    bootstrap: [
10      AppComponent
11    ]
12  })
13  export class AppModule { }
```

Library Usage

Configuration — @ngx-config

設定檔載入Module

```
1 | npm install @ngx-config/core -save
2 | npm install @ngx-config/http-loader --save
```

使用方式說明

<https://www.npmjs.com/package/@ngx-config/core>

使用方式說明

<https://www.npmjs.com/package/@ngx-config/http-loader>

i18n — @ngx-translate

Angular多國語系套件

```
1 | npm install @ngx-translate/core -save
2 | npm install @ngx-translate/http-loader --save
```

使用方式說明

<https://www.npmjs.com/package/@ngx-translate/core>

使用方式說明

<https://www.npmjs.com/package/@ngx-translate/http-loader>

Library Usage

echarts — ngx-echarts

ECharts，一個使用JavaScript實現的開源可視化庫，可以流暢的運行在PC和移動設備上，兼容內部絕大部分瀏覽器，依賴於矢量圖形庫ZRender，提供直觀，交互豐富，可高度個性化定制的數據可視化圖表。

1	<code>npm install echarts --save</code>
2	<code>npm install ngx-echarts --save</code>

詳細使用方式	https://www.npmjs.com/package/ngx-echarts
官方配置文件	https://echarts.apache.org/zh/option.html#title
官方範例Demo	https://echarts.apache.org/examples/zh/index.html
主題建構工具	https://www.echartsjs.com/theme-builder/

Library Usage

MQTT — paho-mqtt

Paho JavaScript Client是基於MQTT瀏覽器的客戶端庫Javascript編寫而成，該庫使用WebSockets連接到MQTT Broker。

```
1 | npm install paho-mqtt --save
2 | npm install @types/paho-mqtt --save
```

詳細使用方式

<https://www.npmjs.com/package/@types/paho-mqtt>

jQuery

jQuery是一個快速，小型且功能豐富的庫。借助易於使用的API，使HTML文檔的遍歷和操作，事件處理等事情變得更加簡單。

```
1 | npm install jquery --save
2 | npm install @types/jquery --save
```

詳細使用方式

<https://api.jquery.com/>

Library Usage

ElasticSearch — elastic-builder

提供Elasticsearch 查詢語句的構建器語法，使用上可以更直觀。

```
1 | npm install elastic-builder --save
```

使用方式說明	https://www.npmjs.com/package/elastic-builder
官方配置文件	https://elastic-builder.js.org/docs/#elastic-builder
查詢語法轉換工具	https://elastic-builder.js.org/

版本相容性注意事項

elastic-builder是為5.x查詢DSL構建的。但是該Library也可用於2.x的版本。對於較舊版本的DSL，可以嘗試使用[elastic.js](#)或[bodybuilder](#)

Library Usage

TS Builder — builder-pattern

使用ES6代理為Typescript創建構建器模式。

```
1 | npm install builder-pattern --save
```

詳細使用方式

<https://www.npmjs.com/package/builder-pattern>

Matomo — ngx-matomo

Matomo使用者監控套件

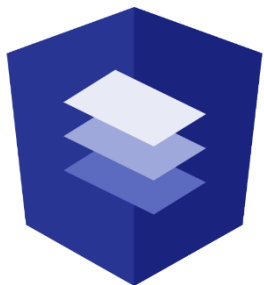
```
1 | npm install --save ngx-matomo
```

使用方式說明

<https://www.npmjs.com/package/ngx-matomo>

Library Usage

1 Angular Material



Angular Material的目標是使用Angular及TypeScript打造出**高品質**的UI元件，同時這些元件必須遵守Material Design的設計標準。

使用方法：

```
1 | cd <project_directory>
2 | ng add @angular/material
```

2 Ng Zorro



這是 Ant Design 的 Angular 實現，結合Material CDK 優秀的元件，開發和服務於企業級後臺產品。

使用方法：

```
1 | cd <project_directory>
2 | ng add ng-zorro-antd
```

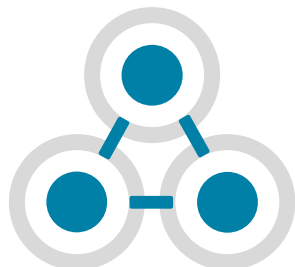
Atomic Design



Atoms

原子

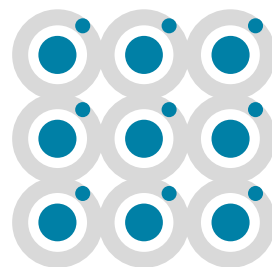
網頁構成基本元素，如輸入、按鈕，也可為抽象的概念，如字體、色調等



Molecules

分子

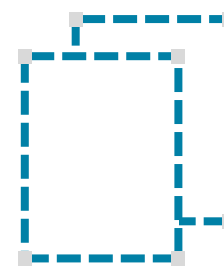
由元素或原子構成的簡單UI物件，構成形式為
原子 + 原子



Organisms

組織

對分子而言，較為複雜的構成物，構成形式為
原子 + 分子



Templates

模板

以頁面為基礎的架構，將以上元素進行排版
原子 + 分子 + 組織



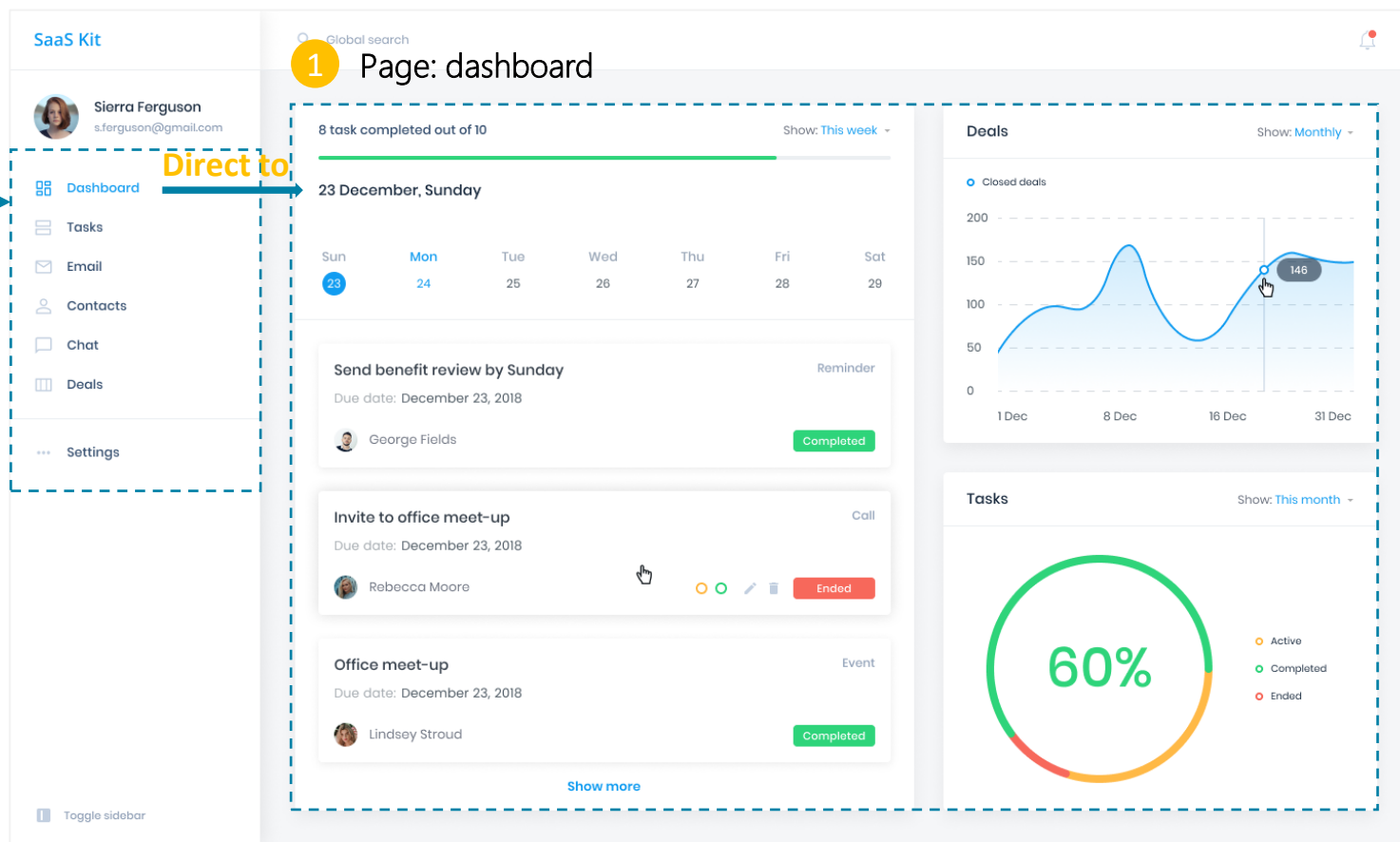
Pages

頁面

將實際內容（圖片、文章、圖表等）嵌入在特定模板當中

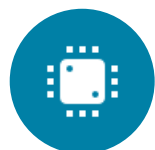
Atomic Design

檔案總管	settings.json
已開啟的編輯器	
settings.json	128
ATOMIC-DESIGN-01	129
core	130
pages	131
chat	132
contacts	133
dashboard	134
deals	135
email	136
main	137
settings	138
tasks	139
shared	140
	141



pages (by feature)

依照頁面(或功能)劃分



core

共用模組，僅單一實例元件或服務



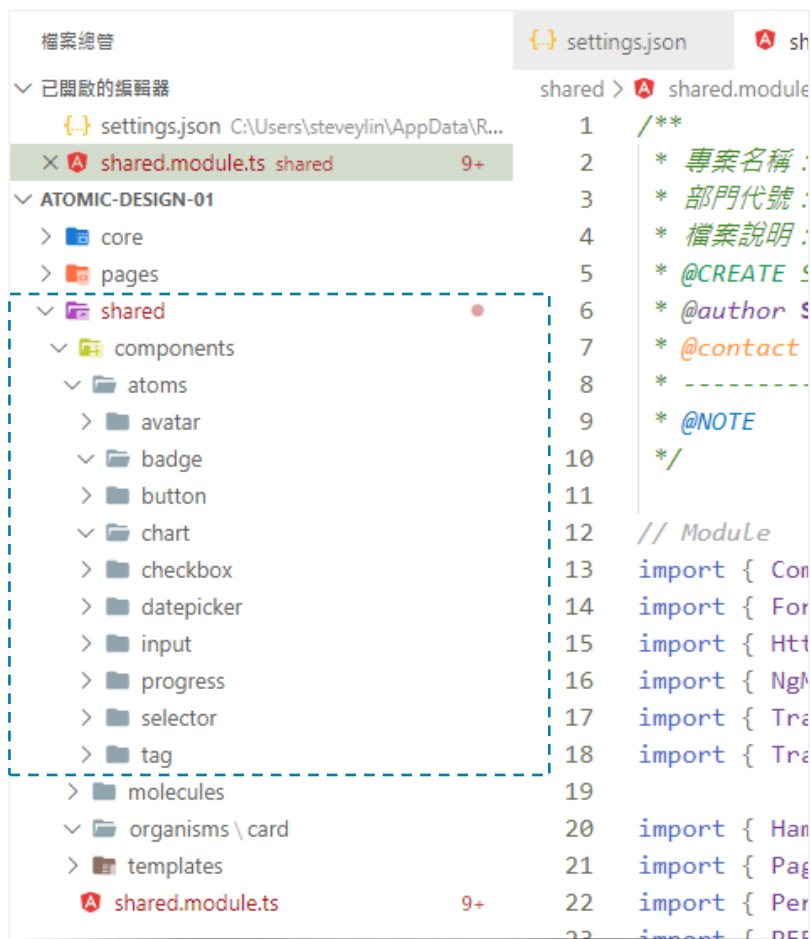
shared

共用模組，多個實例元件

Atomic Design

Shared Atoms

Path shared/components/atoms/{by feature}



selector

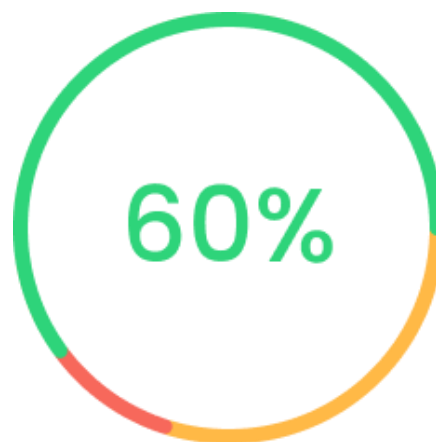
Selector: [item](#)

datepicker

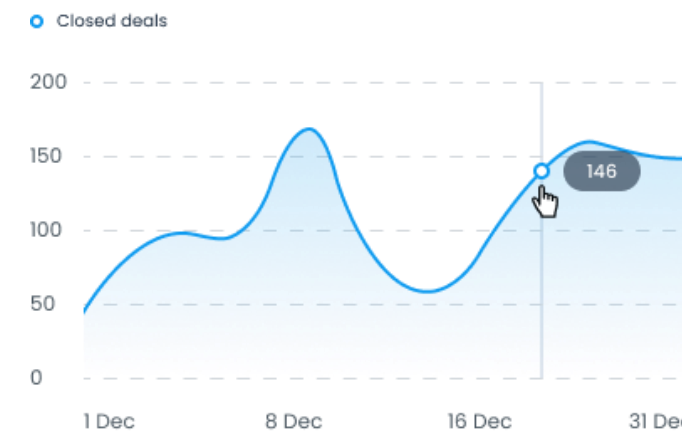
23 December, Sunday

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	24	25	26	27	28	29

chart



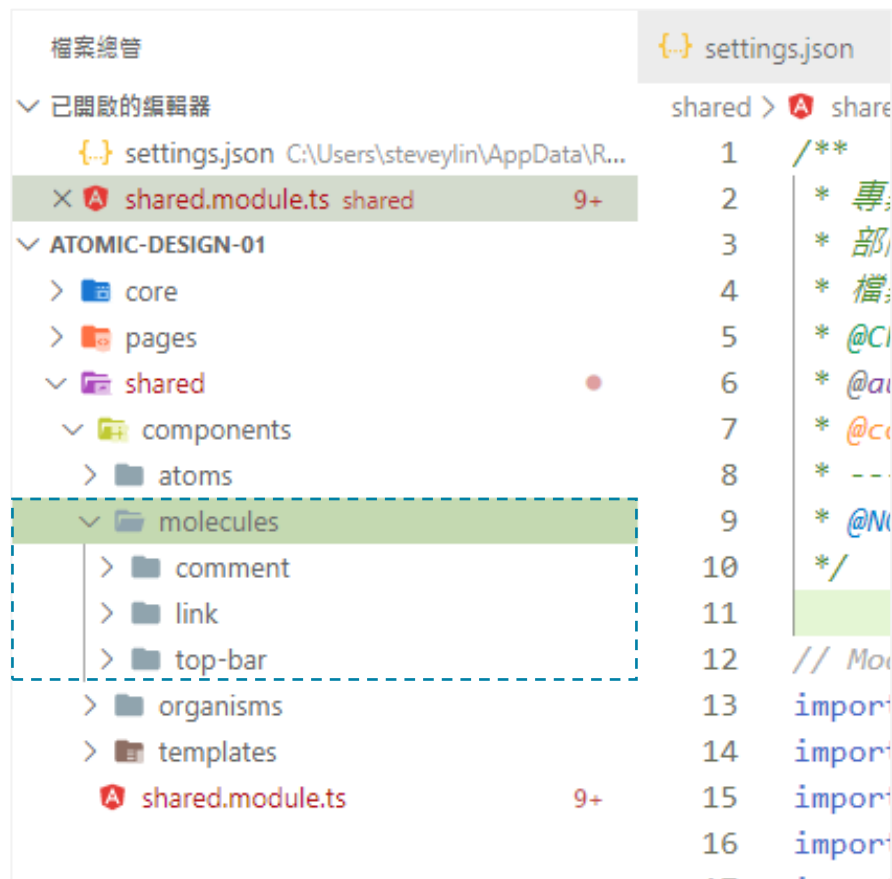
※ 利用echarts繪製canvas，故可視為最小單位



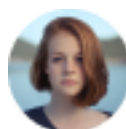
Atomic Design

Shared Molecules

Path shared/components/molecules/{by feature}



comment



Sierra Ferguson
s.ferguson@gmail.com



Rebecca Moore

Composition:

avatar

typography

link



Dashboard ... Settings

Composition:

icon

button

top-bar



Global search



Composition:

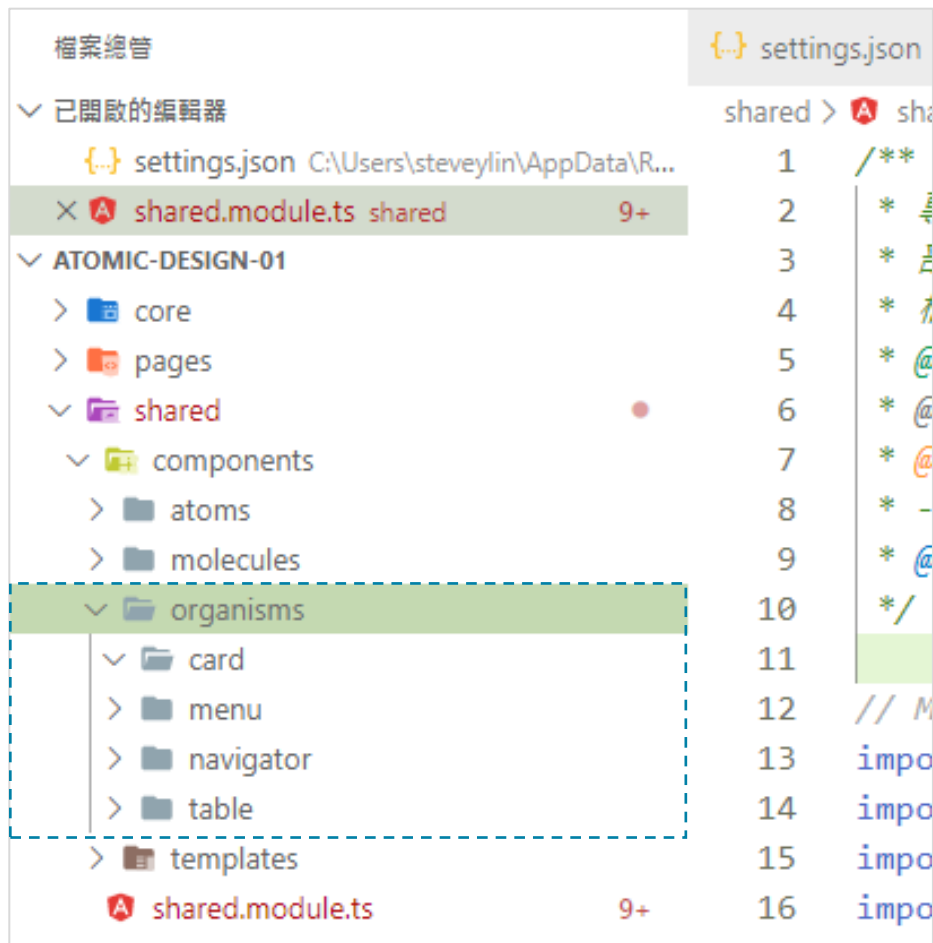
input

badge

Atomic Design

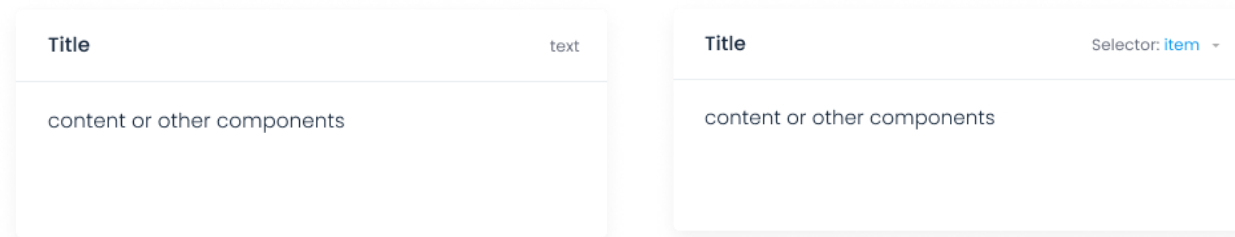
Shared Organisms

Path shared/components/organisms/{by feature}



card

✖ card可以嵌入其他元件



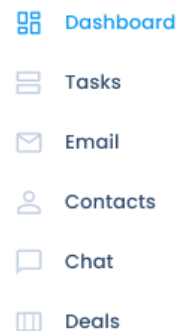
Composition:

selector

typography

transclude

menu



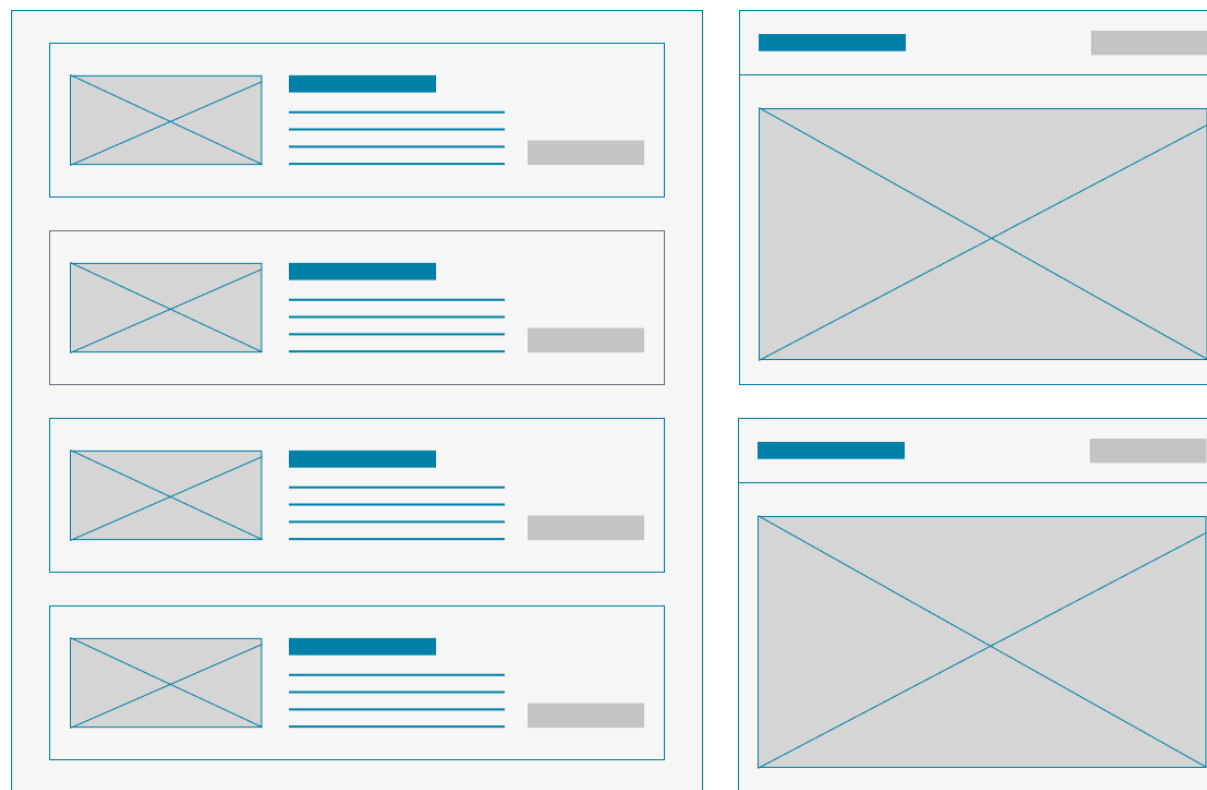
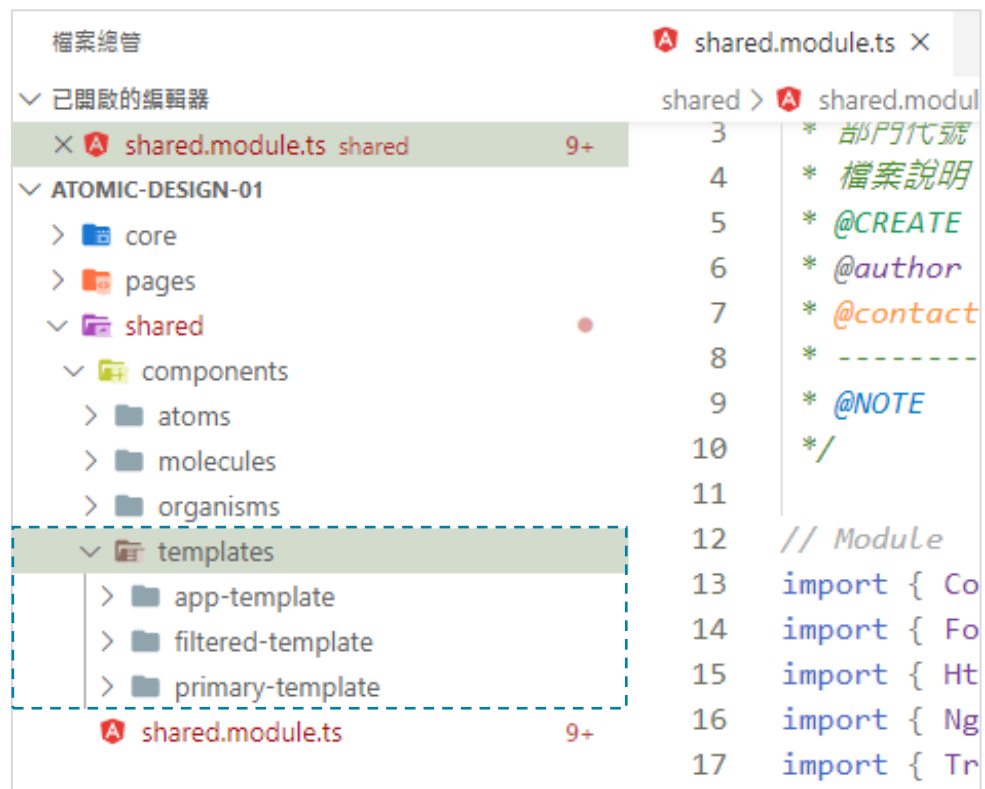
Composition:

link

Atomic Design

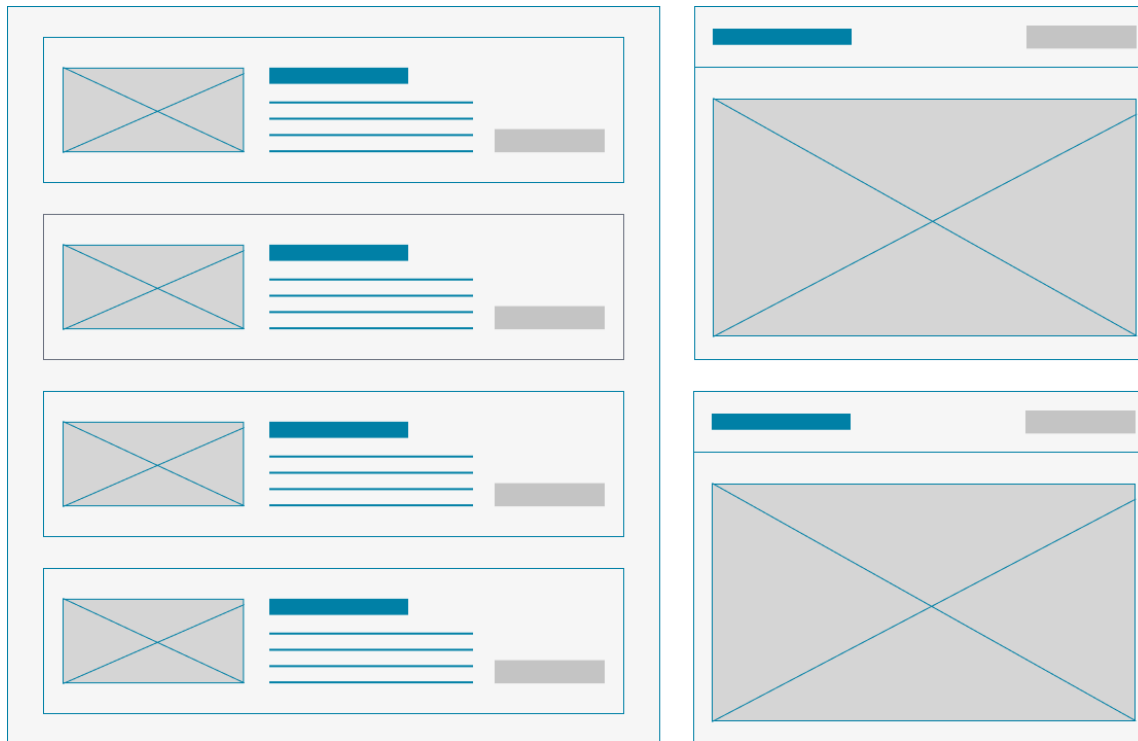
primary-template

Path `shared/components/templates/{by feature}`

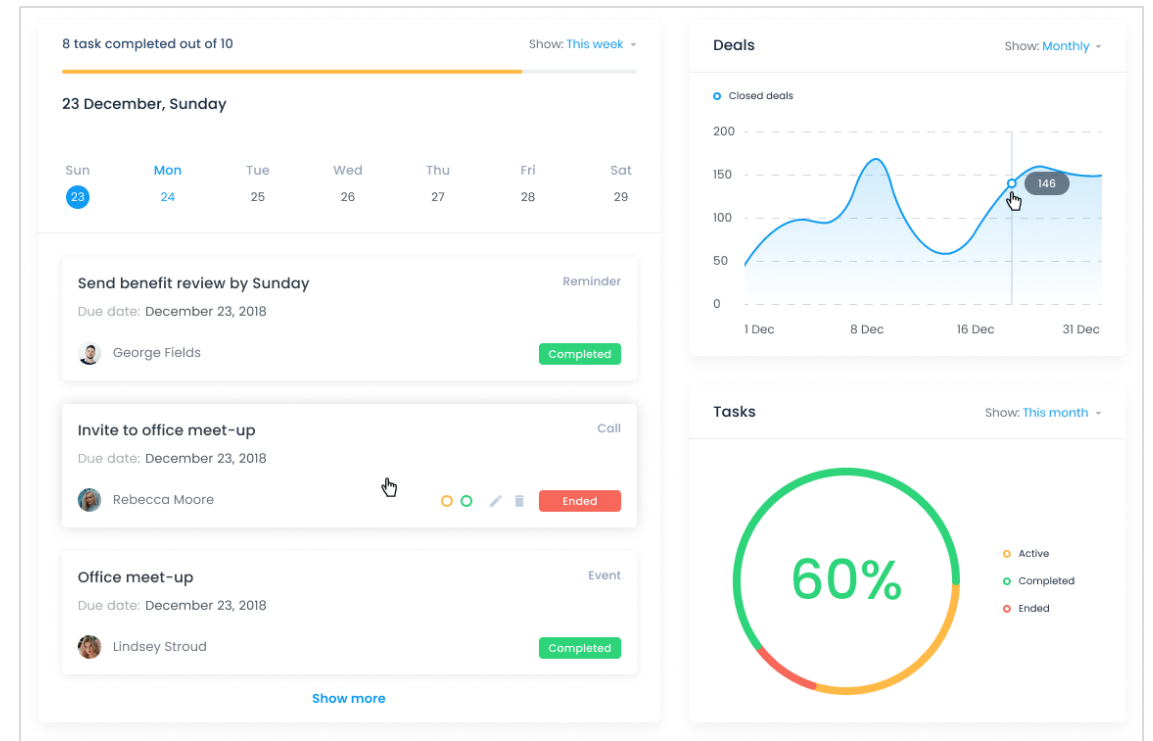


Atomic Design

primary-template



Page Dashboard



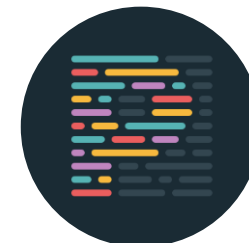
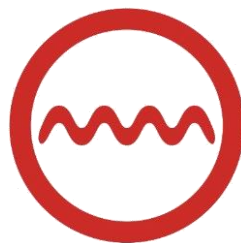
Guide Line

使用格式化工具



TSLint Code格式分析

靜態分析工具，用於檢查TypeScript代碼的可讀性，可維護性和功能性錯誤。



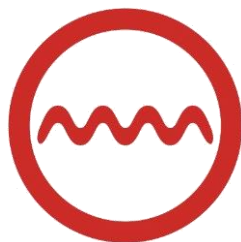
Step 1.

至Vscode的擴展工具中搜尋TSLint並安裝，並新增一個Typescript文檔，輸入右方的Sample Code，會發現不符合規範的Statement底下會有**波浪底線**標記

```
1      export class test {  
2          public value = "123";  
3      }  
4      // class自首須為大寫  
5      // 字串需使用"
```

Guide Line

使用格式化工具



SonarLint 程式碼靜態掃描工具

SonarLint是一個IDE擴展，編寫代碼時檢測和修復質量問題。



Step 2.

至Vscode的擴展工具中搜尋SonarLint([擴展來源連結](#))並安裝，輸入右方的Sample Code，會發現不符合規範的Statement底下會有波浪底線標記

```
1      <div>
2          
3      </div>
4      <!-- 會提示以下訊息 -->
5      <!-- Add an "alt" attribute to this image: sonarlint -->
```


Guide Line

使用格式化工具



Prettier Code格式化工具

可設定為存檔時自動格式化，並且可以根據團隊規範，去調整設定規則。

Step 3.

在Project底下新在一個.vscode的folder並在其中加入settings.json，並加入右方設定，在每一次存檔時，就會根據設定將code格式化。

```
1 {  
2   "editor.formatOnSave": true,  
3   "editor.insertSpaces": true,  
4   "editor.codeActionsOnSave": {  
5     "source.organizeImports": true,  
6     "source.fixAll tslint": true  
7   },
```

```
8   "prettier.bracketSpacing": false,  
9   "prettier.semi": true,  
10  "prettier.singleQuote": true,  
11  "prettier.trailingComma": "es5",  
12  "prettier.printWidth": 80,  
13  "prettier.useTabs": false,  
14  "prettier.tabWidth": 2,  
15 }
```

Guide Line

型別定義

資料型態盡可能不要使用any，最好能夠定義成interface或class。可以使用[JSON to TS](#)工具幫忙進行轉換

- **優點:** 嚴謹且在VS code編輯時，能夠出現提示，避免因為人為錯誤，導致bug發生。
- **缺點:** Coding過程變得繁瑣、相應檔案增加。

```
1 {  
2   "evt_dt": 1593532799000,  
3   "freq": "D",  
4   "site": "WCQ",  
5   "company": "WT",  
6   "plant": "WCQ-P1-711",  
7   "plant_code": "F711",  
8   "system_id": "dfq_sems",  
9   "type_id": "quality_optimization",  
10  "benefit_type": "manage"  
11 }
```



```
1 interface Definition {  
2   evt_dt: number;  
3   freq: string;  
4   site: string;  
5   company: string;  
6   plant: string;  
7   plant_code: string;  
8   system_id: string;  
9   type_id: string;  
10  benefit_type: string;  
11 }
```

OR

```
1 class Definition {  
2   public evt_dt: number;  
3   public freq: string;  
4   public site: string;  
5   public company: string;  
6   public plant: string;  
7   public plant_code: string;  
8   public system_id: string;  
9   public type_id: string;  
10  public benefit_type: string;  
11 }
```



Guide Line

泛型

若一個函數或方法，無法明確定義資料的型別時，可以透過泛型來達成，以避免any的使用，例如取得設定檔，一個APP可能包含多個設定檔，而每個設定檔都有對應的型別，因此可以透過泛型來達成。

```
1  export class Configuration<T> {
2      public config: T;
3
4      constructor(config: T) {
5          this.config = config;
6      }
7  }
8
9  export class ConfigureService {
10     private readonly config = new Map<string, Configuration<any>>();
11
12     constructor() {}
13
14     public setConfig<T>(key: string, val: T): void {
15         this.config.set(key, new Configuration<T>(val));
16     }
17
18     public getConfig<T>(key: string): T {
19         return this.config.get(key)?.config;
20     }
21 }
```

Guide Line

善用enum及type

若出現大量的常數項且使用的機會非常頻繁時，可以透過enum，甚至其他人接手開發時，可以更容易維護

- **優點:** 提高程式碼的可讀性
- **缺點:** Coding過程變得繁瑣、相應檔案增加。

enum

```
1  export class SharedModule { }
2
3  enum Payment {
4      CREDIT_CARD,
5      CASH,
6      TRANSFER
7  }
8
9  const discount = [0.8, 1, 1];
```

type

```
1  type Payment = 'credit_card' | 'cash' | 'transfer';
2
3  function pay(payment: Payment): void {
4      if (payment === 'credit_card') {
5          // TODO
6      } else if (payment === 'cash') {
7          // TODO
8      } else if (payment === '') {
9          // TODO
10     }
11 }
```



transfer

輸入時會跳出提示

Guide Line

Utility Types

TypeScript提供了幾種實用程序類型以促進常見的類型轉換。這些實用程序在全域範圍內皆可使用。

1 **Partial<T>** 構造一個類型，將T的所有屬性設置為可選。

```
1  interface Todo {
2      title: string;
3      description: string;
4  }
5
6  function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
7      return { ...todo, ...fieldsToUpdate };
8  }
9
10 const todo1 = {
11     title: 'organize desk',
12     description: 'clear clutter',
13 };
14
15 const todo2 = updateTodo(todo1, {
16     description: 'throw out trash',
17 });
```

Guide Line

Utility Types

2 Pick<T, K> 通過從T選取屬性K來構造類型

```
1 interface Todo {  
2     title: string;  
3     description: string;  
4     completed: boolean;  
5 }  
6  
7 type TodoPreview = Pick<Todo, 'title' | 'completed'>;  
8  
9 const todo: TodoPreview = {  
10     title: 'Clean room',  
11     completed: false,  
12 };
```

更多Utility Types可以參閱官網，<https://www.typescriptlang.org/docs/handbook/utility-types.html>

Decorator【裝飾器】

不修改原程式碼的狀況下，在執行原函式的前後做一些特定的操作，同時也把可以重複使用的邏輯切分出去

類別裝飾器

```
1 function MyDecorator(target: any) {  
2     console.log('Class Decorator');  
3 }  
4  
5 @MyDecorator  
6 class MyClass {}  
7  
8  
9  
10  
11  
12  
13  
14
```

方法裝飾器

```
1 function MyDecorator(  
2     target: any,  
3     propertyKey: string,  
4     descriptor: PropertyDescriptor  
5 ) {  
6     console.log('Method Decorator');  
7 }  
8  
9 class MyClass {  
10     @MyDecorator  
11     public myMethod() {  
12  
13     }  
14 }
```

Decorator【裝飾器】

參數裝飾器

```
1 function MyDecorator(  
2     target: any,  
3     propertyKey: string,  
4     parameterIndex: number  
5 ) {  
6     console.log('Parameter Decorator');  
7 }  
8  
9 class MyClass {  
10     constructor(@MyDecorator public a: number) {  
11     }  
12     public myMethod(@MyDecorator a: number) {  
13         this.a = a;  
14     }  
15 }
```

欄位裝飾器

```
1 function MyDecorator(  
2     target: any,  
3     propertyKey: string  
4 ) {  
5     console.log('Field Decorator');  
6 }  
7  
8 class MyClass {  
9     @MyDecorator  
10     public n: number = 0;  
11 }  
12  
13  
14
```


Decorator【裝飾器】

裝飾器工廠

```
1 function log(tag: string) {
2   return (
3     target: any,
4     propertyKey: string,
5     descriptor: PropertyDescriptor
6   ) => {
7     let originalMethod = descriptor.value;
8
9     descriptor.value = function () {
10       console.log(tag);
11
12       originalMethod.call(this);
13     };
14   };
15 }
16
17 class MyClass {
18   @log('m1')
19   public myMethod() {}
20 }
```

裝飾器工廠實際應用範例

```
1 import { scheduleJob } from 'node-schedule';
2
3 export function Cron(rule: string) {
4   return (
5     target: any,
6     propertyKey: string,
7     descriptor: PropertyDescriptor
8   ) => {
9     const original = descriptor.value;
10    descriptor.value = function(...args: any[]) {
11      const instance = this;
12      scheduleJob(rule, () => {
13        original.apply(instance, args);
14      });
15    };
16  };
17  // @Cron('10/* * * * * *')
18  // public run(): void {
19  //   console.log('run schedule');
20  // }
```

S.O.L.I.D

Single responsibility principle (SRP)

單一職責原則的原文定義：

A class should have only one reason to change.

可讀性與可維護性提升

單一類別的複雜度降低，因為要實現的職責都很清晰明確的定義，這將大幅提升可讀性與可維護性。

強健性提升

如果有做好 SRP，那修改只會對同一個介面或類別有影響，這對擴展性和維護性都有很大的幫助。



Component

處理View的操作及流程
切勿涉及商業邏輯



Service

資料介接獲商業邏輯
處理



Directive

讓Component可以有
額外的行為或樣式



Pipe

字串、金額、日期和
等資料進行格式轉換



Guard

檢視是否有訪問該路
由的權限

S.O.L.I.D

Single responsibility principle (SRP)

```
1  class Employee {
2
3      private db = new MariaDB('localhost', 3306);
4
5      constructor() { }
6
7      public connectDb(): Promise<void> {
8          return new Promise<void>(resolve => {
9              this.db.connect(() => resolve());
10             });
11        }
12
13        public disconnectDb(): void {
14            this.db.disconnect();
15        }
16
17        public findOne(emp: string): IEmployee {
18            this.db.execute(`SELECT * FROM employee WHERE empId = '${emp}'`);
19        }
20
21    }
```

Employee Model 進行DB連線及斷線 ??

Avoid

S.O.L.I.D

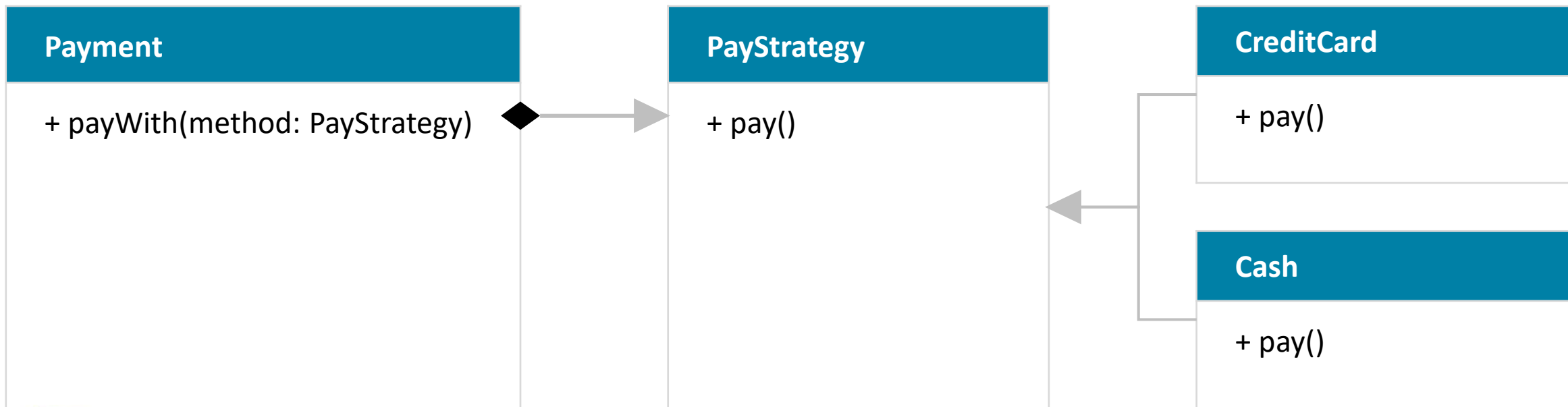
Open-Close principle (OCP)

開閉原則的原文定義：

Software entities (class, modules, functions, etc.) should be open for extension, but closed for modification.

優點

最大的好處正是降低修改風險。修改既有程式碼，因此有可能破壞原有功能；後面重構後的修改，只有新增程式碼，舊有程式因為沒修改，所以理論上問題會比較少。



S.O.L.I.D

O Open-Close principle (OCP)

```
1 class ChartDisplayManager {  
2     public display(name: string) {  
3         switch (name) {  
4             case 'pie':  
5                 new PieChart().display();  
6                 break;  
7             case 'bar':  
8                 new BarChart().display();  
9                 break;  
10            default:  
11                break;  
12        }  
13    }  
14 }
```

對擴展開放，但沒有對修改關閉

Avoid

S.O.L.I.D

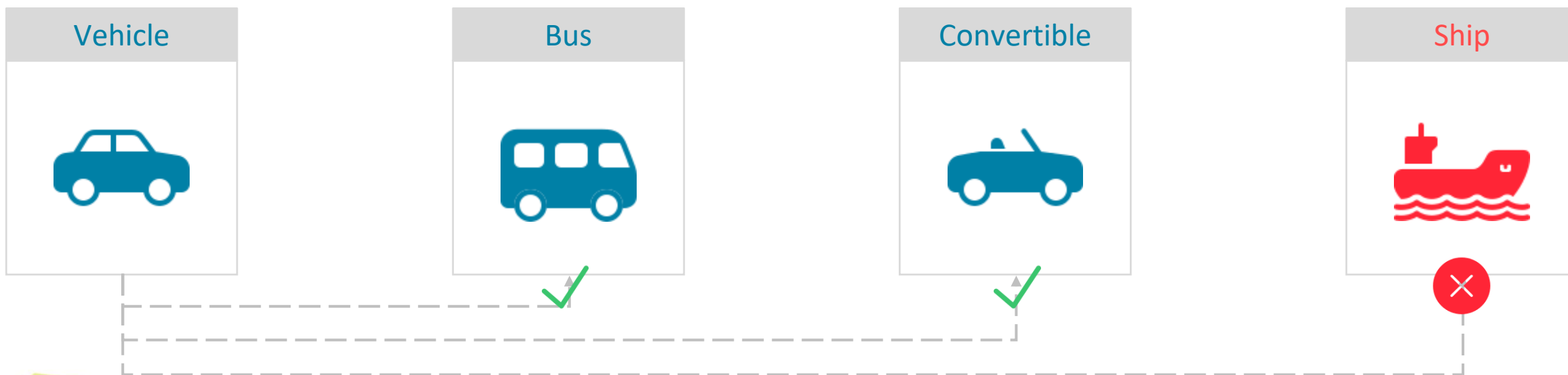
Liskov substitution principle (LSP)

里氏替換原則的原文定義：

Subtypes must be substitutable for their base types.

優點

里氏替換原則的重點是要增加程式的強健性，讓版本升級的時候也能有很好的兼容性。比方說：子類別增加或修改，並不影響其他子類別，這正是強健性的特質。



S.O.L.I.D

Liskov substitution principle (LSP)

```
1  abstract class Gun {
2      public abstract shoot(): void;
3  }
4
5  class Rifle extends Gun {
6      public shoot() {
7          console.log('rifle shoot');
8      }
9  }
10
11 class ToyGun extends Gun {
12     public shoot() {
13         console.log(`toy gun can't shoot`);
14     }
15 }
16
17 const soldier = new Soldier();
18 soldier.killEnemy(new Rifle());
19 soldier.killEnemy(new ToyGun());
```

Avoid

S.O.L.I.D

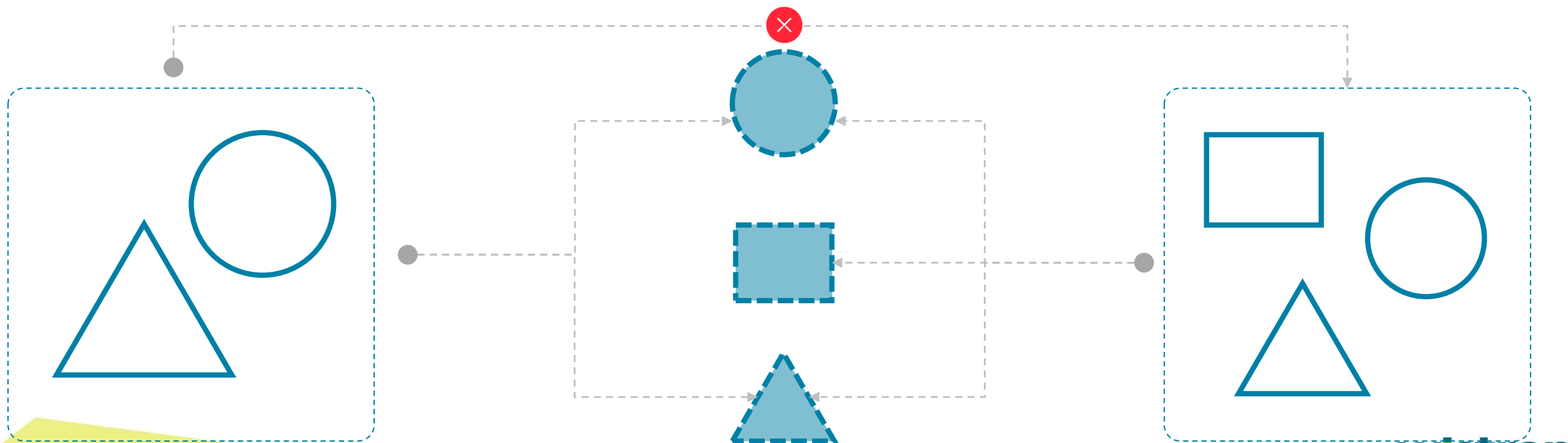
I Interface segregation principle (ISP)

介面隔離原則的原文定義：

Clients should not be forced to depend on methods that they do not use.

優點

遵守介面隔離原則最大的好處是，在需要多型時，會比較容易為類別實作對應方法



S.O.L.I.D

I Interface segregation principle (ISP)

Avoid

```
1 interface Bird {
2     eat(): void;
3     fly(): void;
4 }
5
6 class Cuckoo implements Bird {
7     public eat(): void {
8         console.log('Cuckoo is eating');
9     }
10    public fly(): void {
11        console.log('Cuckoo is flying');
12    }
13 }
```

若攝影師不僅拍攝鳥類，也拍所有動物？

```
15 class Photographer {
16     searchBird(bird: Bird): void {
17         bird.eat();
18         console.log('Take a photo');
19         bird.fly();
20         console.log('Take a photo');
21     }
22 }
```

S.O.L.I.D

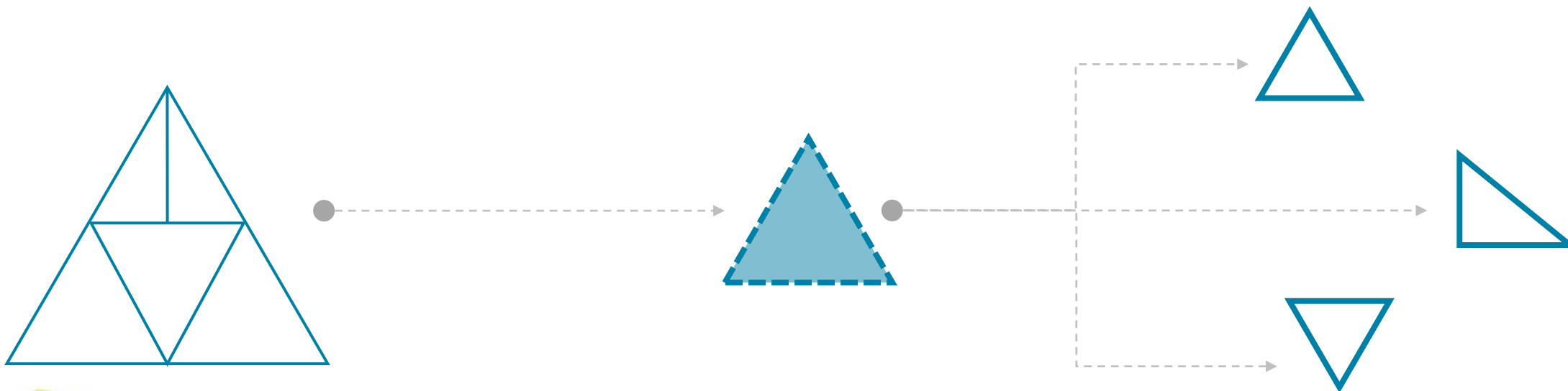
Dependency inversion principle (DIP)

依賴反轉原則的原文定義：

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

優點

從緊密結合變鬆散耦合關係，可依據需求隨時抽換實作，
符合DIP通常也意味著符合OCP與LSP原則，只需在考量
SRP與ISP原則



S.O.L.I.D

D Dependency inversion principle (DIP)

Avoid

```
1 interface Readable {
2     readBy(someone: string): void;
3 }
4
5 class Book implements Readable {
6     public readBy(someone: string): void {
7         console.log(`${someone} read a book`);
8     }
9 }
10
11 class Newspaper implements Readable {
12     public readBy(someone: string): void {
13         console.log(`${someone} read a newspaper`);
14     }
15 }
16
17 class Person {
18     public name = 'Mike';
19     public read(book: Book): void {
20         book.readBy(this.name);
21     }
22 }
```

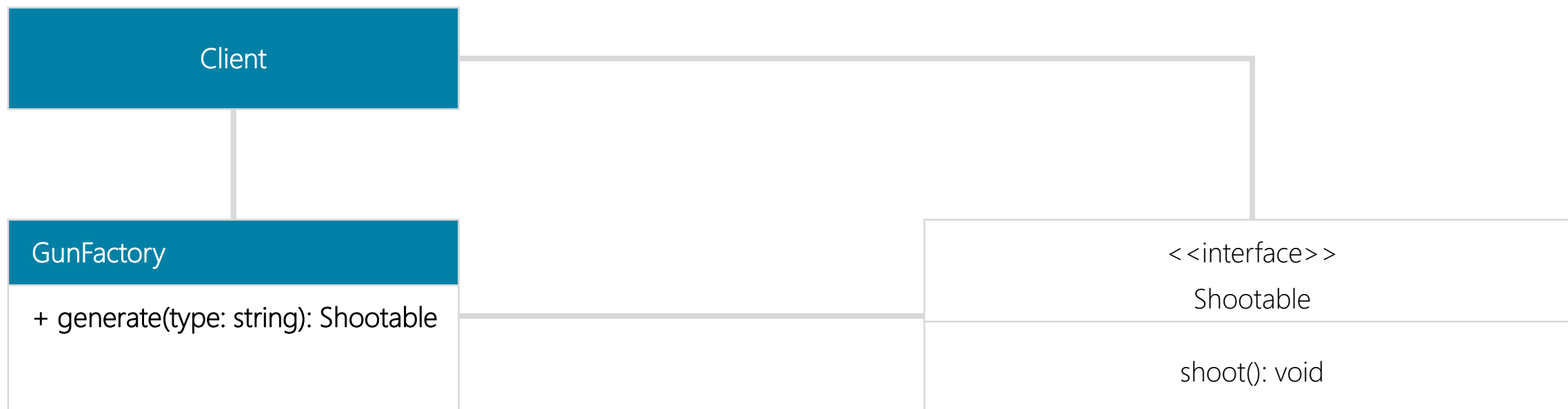
可以讀書籍，但不能讀報紙??

Design Pattern

Factory Pattern - Simple Factory

Creational 模式

Simple Factory模式又稱Static Factory模式。一個Simple Factory生產成品，而對客戶端隱藏產品產生的細節，物件如何生成，生成前是否與其它物件建立依賴關係，客戶端皆不用理會，用以將物件生成方式之變化 與客戶端程式碼隔離。

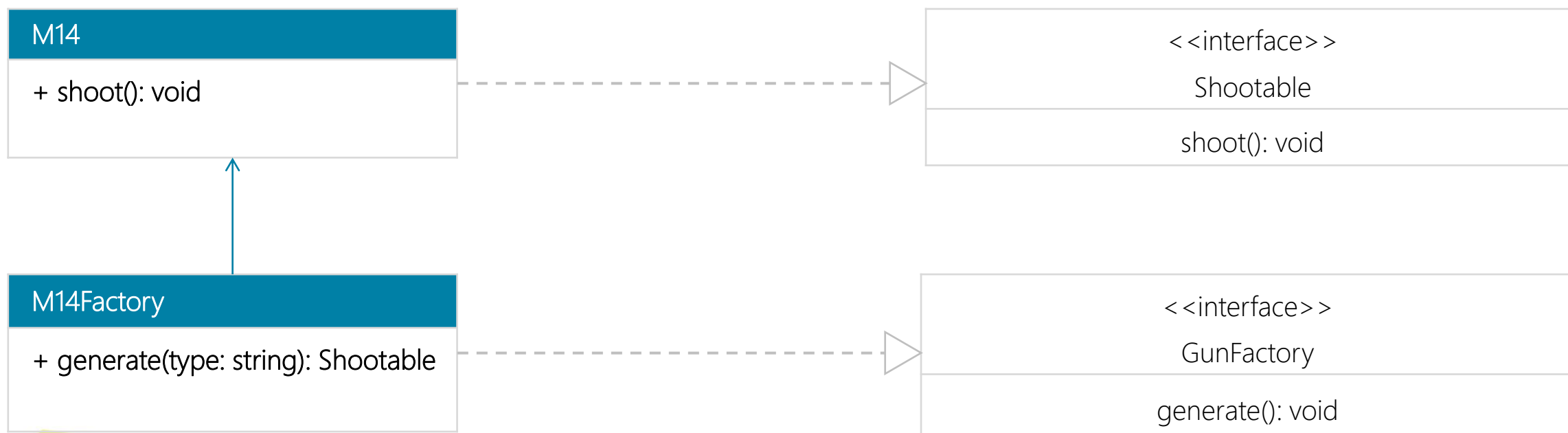


Design Pattern

Factory Pattern - Factory Method

Creational 模式

Factory Method模式在一個抽象類別中留下某個建立元件的抽象方法沒有實作，其它與元件操作相關聯的方法都先依賴於元件所定義的介面，而不是依賴於元件的實現，當您的成品中有一個或多個元件無法確定時，您先確定與這些元件的操作介面，然後用元件的抽象操作介面先完成其它的工作。

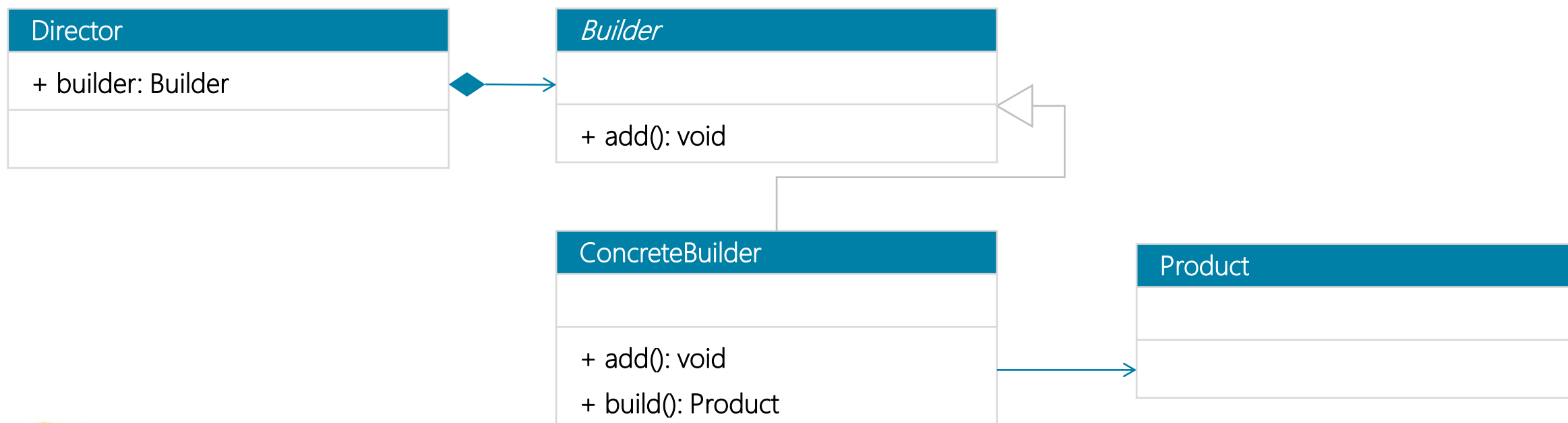


Design Pattern

Builder Pattern

Creational 模式

如果您有個物件必須建立，物件是由個別組件（Component）組合而成，單個組件建立非常複雜，希望將建立複雜組件與運用組件方式分離，則可以使用Builder模式。

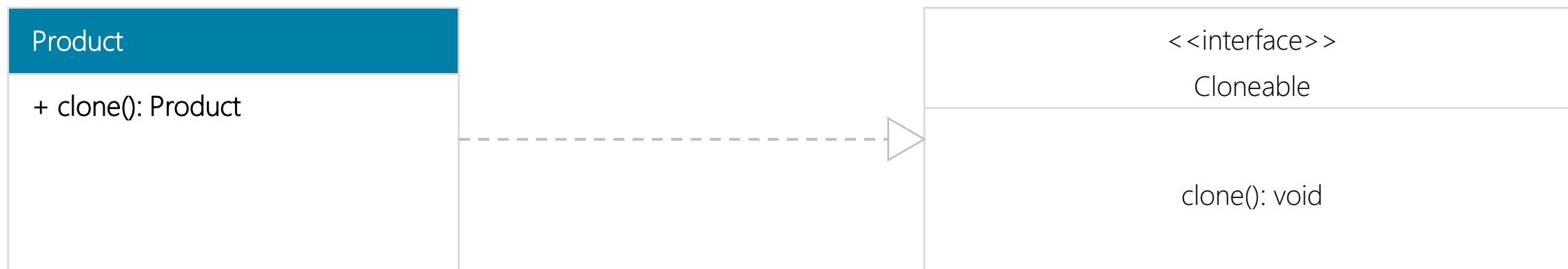


Design Pattern

Prototype Pattern

Creational 模式

有些物件若以標準的方式建立實例，或者是設定至某個狀態需要複雜的運算及昂貴的資源，則您可以考慮直接以某個物件作為原型，在需要個別該物件時，複製原型並傳回。



Design Pattern

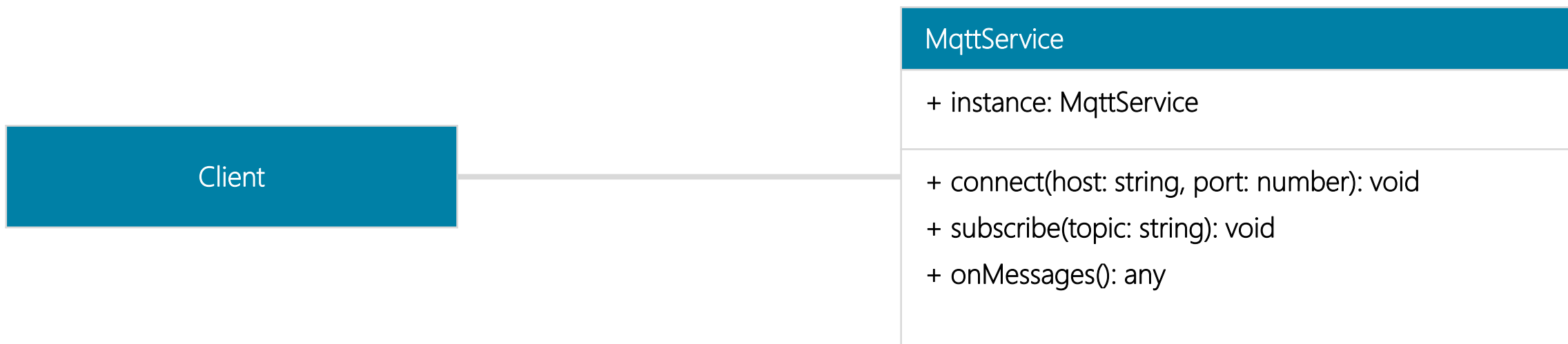
Singleton Pattern

Creational 模式

Singleton 的英文意義是獨身，也就是只有一個人，應用在物件導向語言上，通常翻譯作單例：單一個實例（Instance）。

Singleton 模式可以保證一個類別只有一個實例，並提供一個訪問（visit）這個實例的方法。

Angular的Service就是使用Singleton

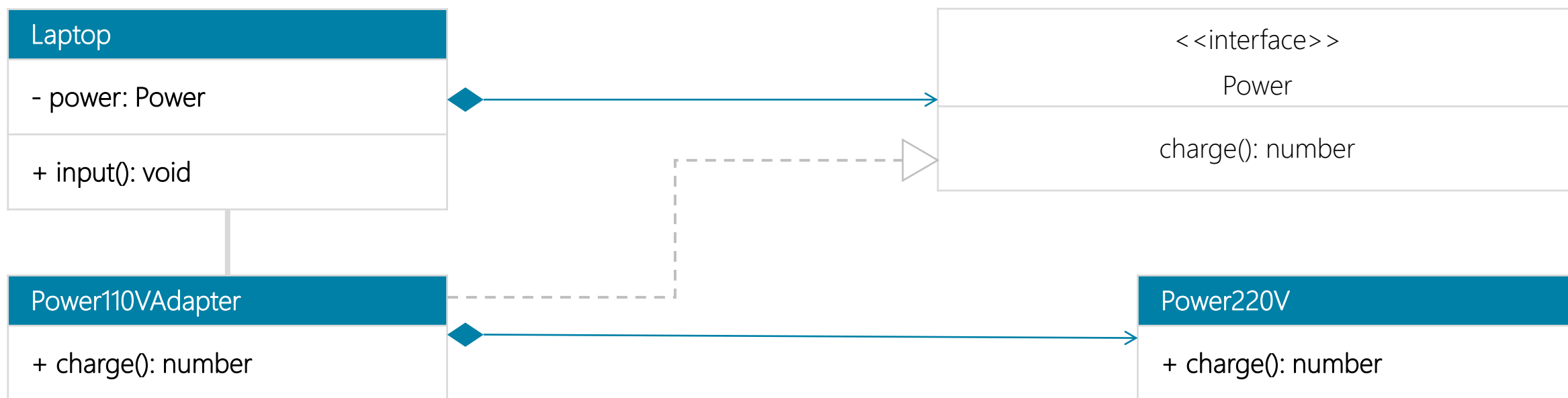


Design Pattern

Adapter Pattern

Structural 模式

物件之間溝通時介面的不匹配時有所見，也許您在開發應用程式的過程中已設計了一些方法供具某些公開協定的物件使用，您打算引入了先前開發過的類別，或者是第三方的類別，但很常發現到它們之間的公開介面並不一致。

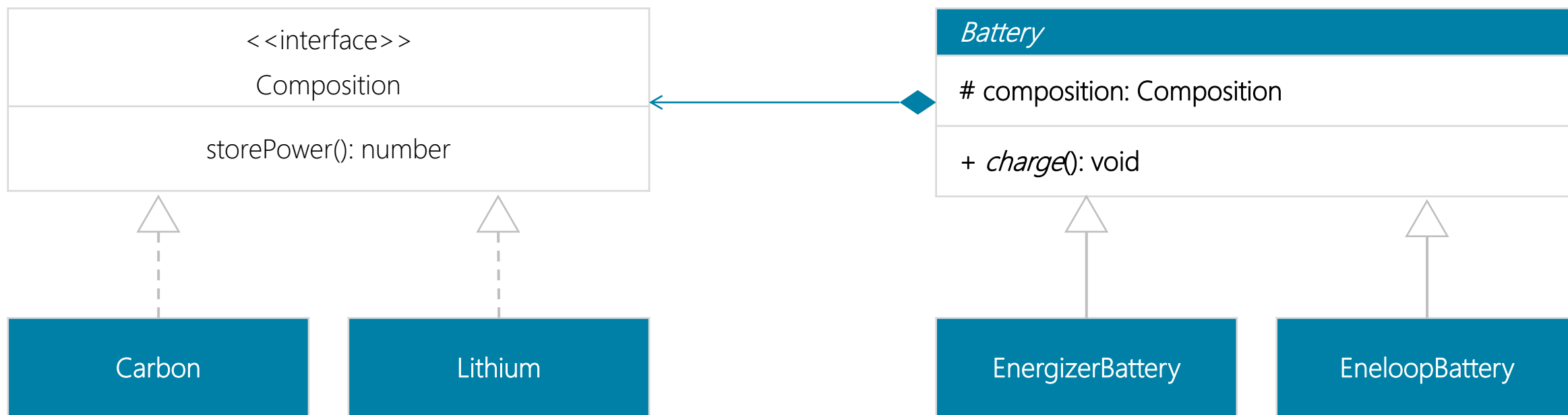


Design Pattern

Bridge Pattern

Structural 模式

最複雜的模式之一，它把事物對象和其具體行為、具體特徵分離開來，使它們可以各自獨立的變化。事物對象僅是一個抽象的概念。

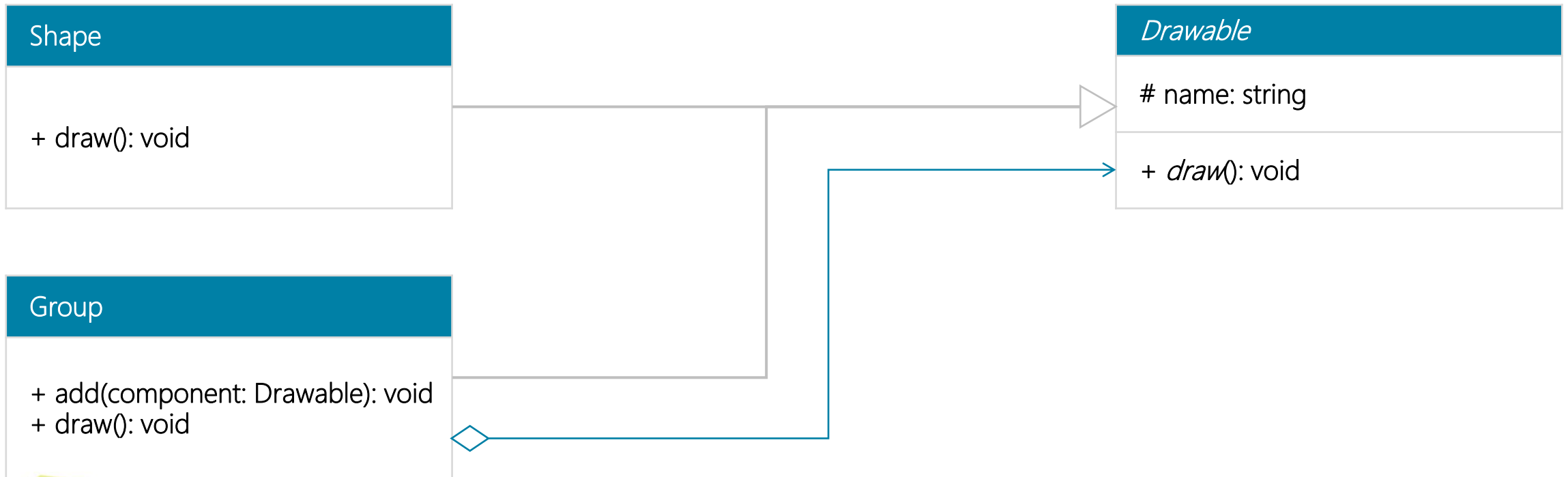


Design Pattern

Composite Pattern

Structural 模式

將物件以樹形結構組織起來,以達成「部分 - 整體」的層次結構，使得用戶端對單個物件和組合物件的使用具有一致性。

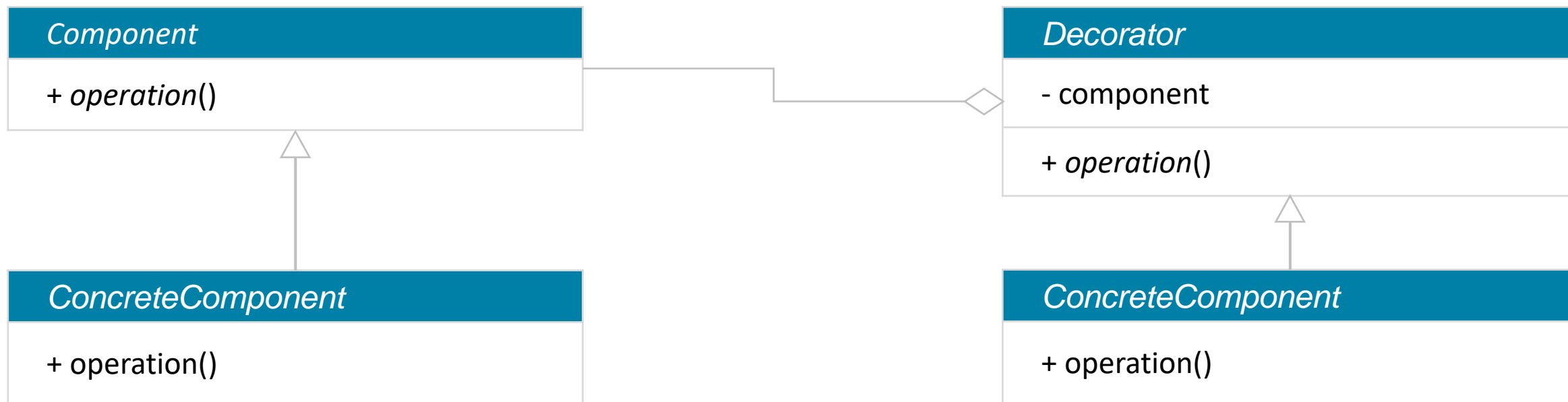


Design Pattern

Decorator Pattern

Structural 模式

一種動態地往一個類中添加新的行為的設計模式。就功能而言，修飾模式相比生成子類更為靈活，這樣可以給某個對象而不是整個類添加一些功能。

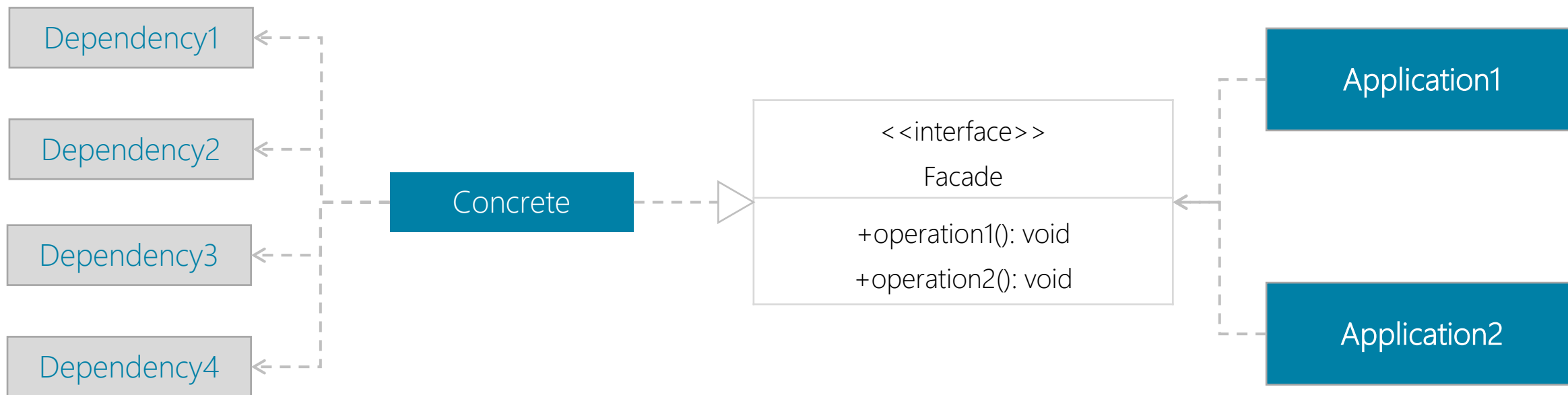


Design Pattern

Facade Pattern

Structural 模式

外觀模式，它為子系統中的一組接口提供一個統一的高層接口，使得子系統更容易使用。



Design Pattern

Flyweight Pattern

Structural 模式

它使用物件用來儘可能減少記憶體使用量；於相似物件中分享儘可能多的資訊。當大量物件近乎重複方式存在，因而使用大量記憶體時，此法適用。



Design Pattern

Proxy Pattern

Structural 模式

所謂的代理者是指一個類別可以作為其它東西的介面。代理者可以作任何東西的介面：網路連接、記憶體中的大物件、檔案或其它昂貴或無法複製的資源。

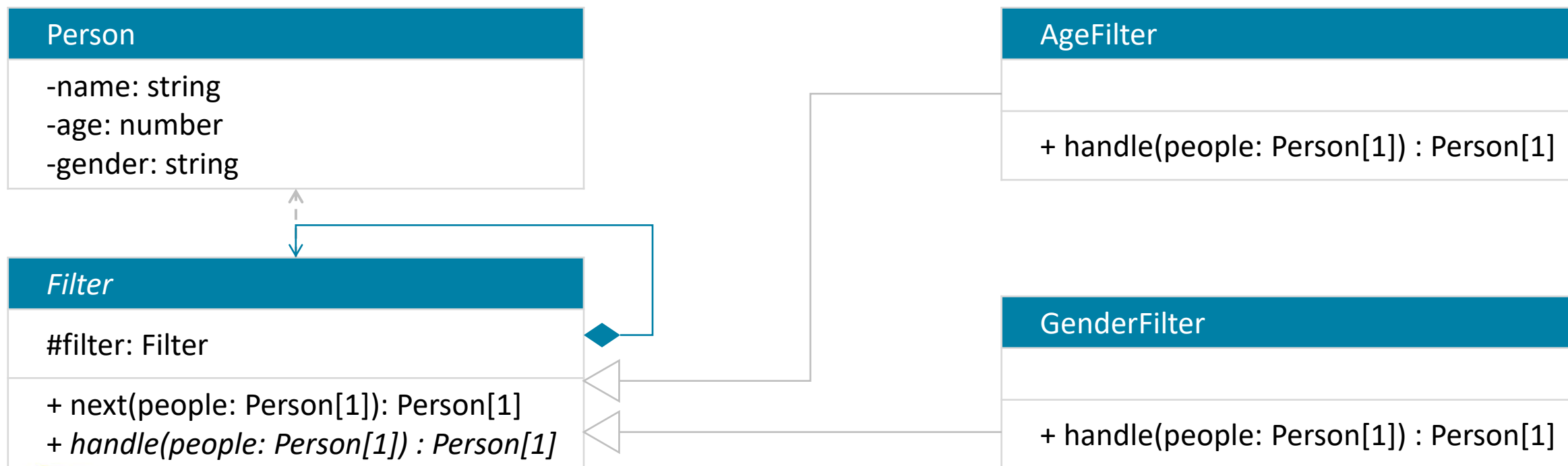


Design Pattern

Chain of Responsibility Pattern

Behavioral 模式

每一個處理對象決定它能處理哪些命令對象，它也知道如何將它不能處理的命令對象傳遞給該鏈中的下一個處理對象。該模式還描述了往該處理鏈的末尾添加新的處理對象的方法。

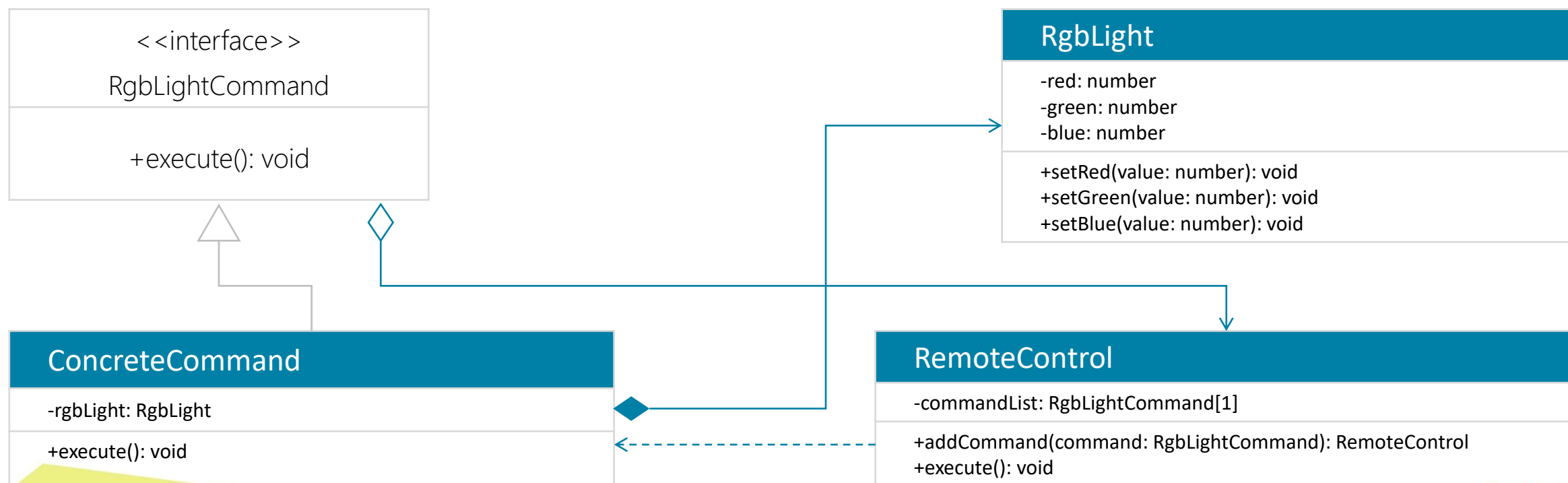


Design Pattern

Command Pattern

Behavioral 模式

嘗試以物件來代表實際行動。命令物件可以把行動(action) 及其參數封裝起來，命令模式來實作的功能例子有交易行為、進度列、精靈、使用者介面按鈕及功能表項目、巨集收錄。

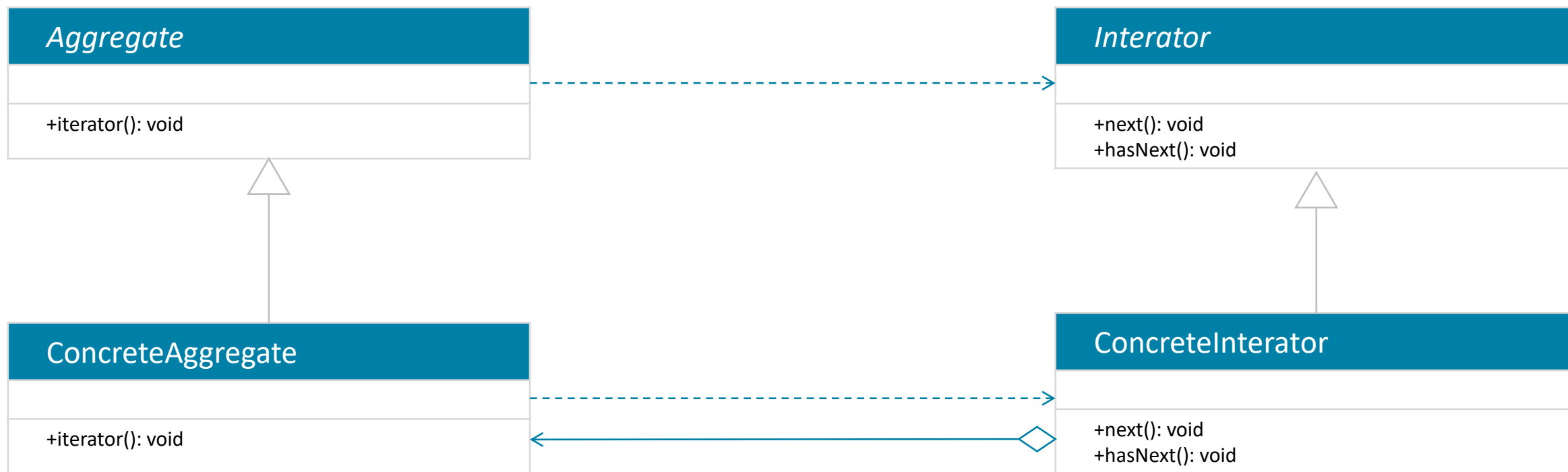


Design Pattern

Iterator Pattern

Behavioral 模式

一種最簡單也最常見的設計模式。它可以讓使用者透過特定的介面巡訪容器中的每一個元素而不用了解底層的實作。



Design Pattern

Template Method Pattern

Behavioral 模式

一個抽象類公開定義了執行它的方法的方式/模板。它的子類可以按需要重寫方法實現，但調用將以抽象類中定義的方式進行。這種類型的設計模式屬於行為型模式。

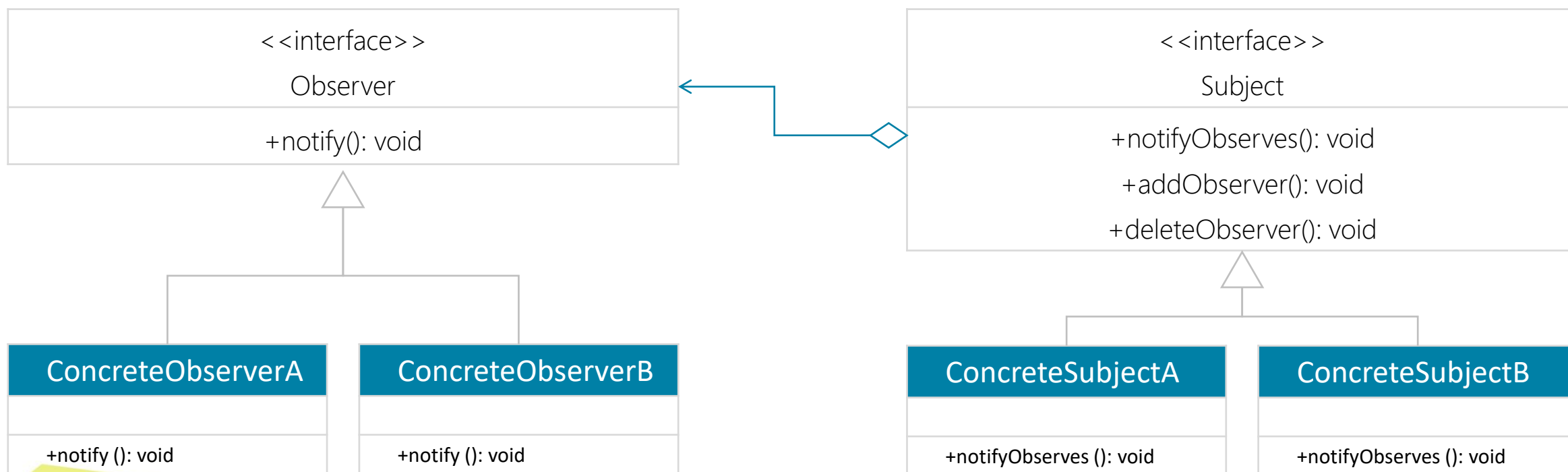


Design Pattern

Observer Pattern

Behavioral 模式

在此種模式中，一個目標物件管理所有相依於它的觀察者物件，並且在它本身的狀態改變時主動發出通知。這通常透過呼叫各觀察者所提供的方法來實現。

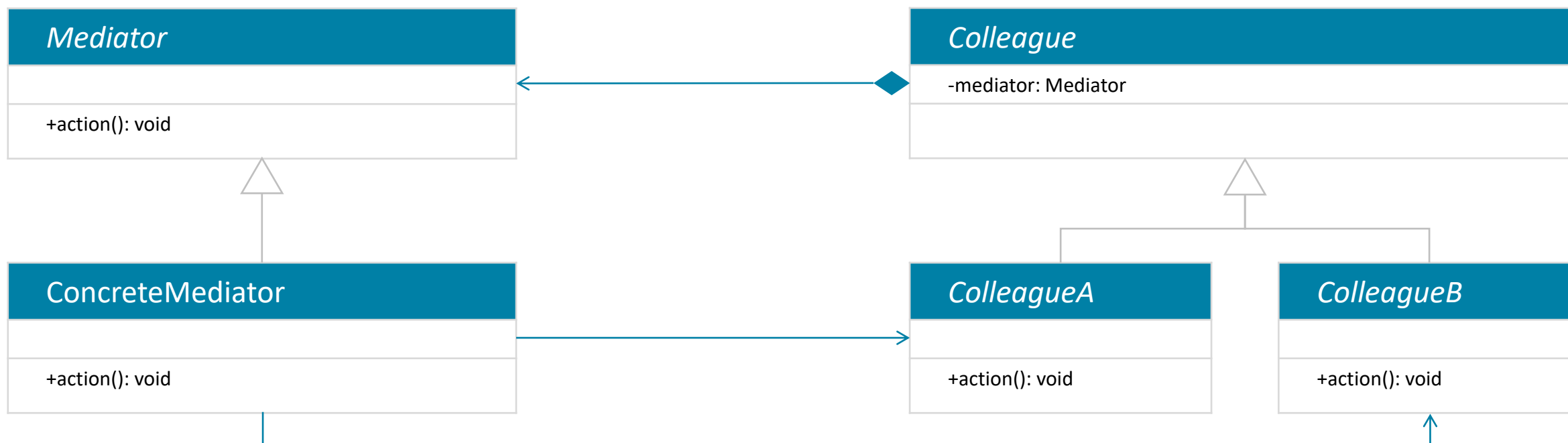


Design Pattern

Mediator Pattern

Behavioral 模式

Mediator的意思是中介者、調節者、傳遞物，顧名思義，這個模式在程式中必然負擔一個中介、調節、傳遞的工作。用一個中介的物件來封裝物件彼此之間的交互，物件之間並不用互相知道另一方，這可以降低物件之間的耦合性。

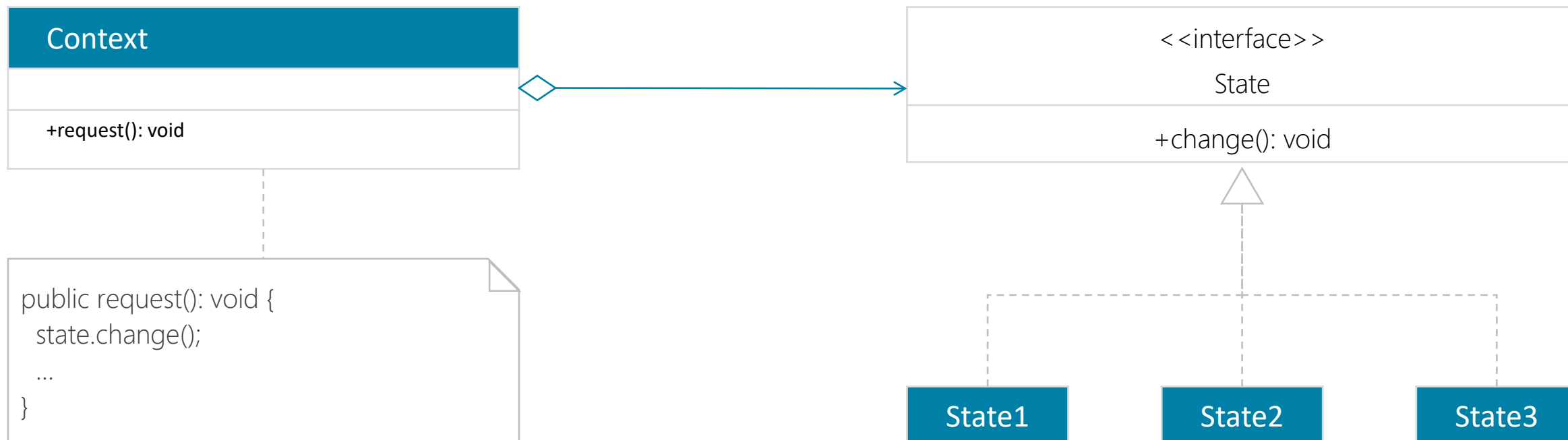


Design Pattern

State Pattern

Behavioral 模式

狀態模式用於解決系統中複雜物件的狀態轉換以及不同狀態下行為的封裝問題。當系統中某個物件存在多個狀態，這些狀態之間可以進行轉換，而且物件在不同狀態下行為不相同時可以使用狀態模式。

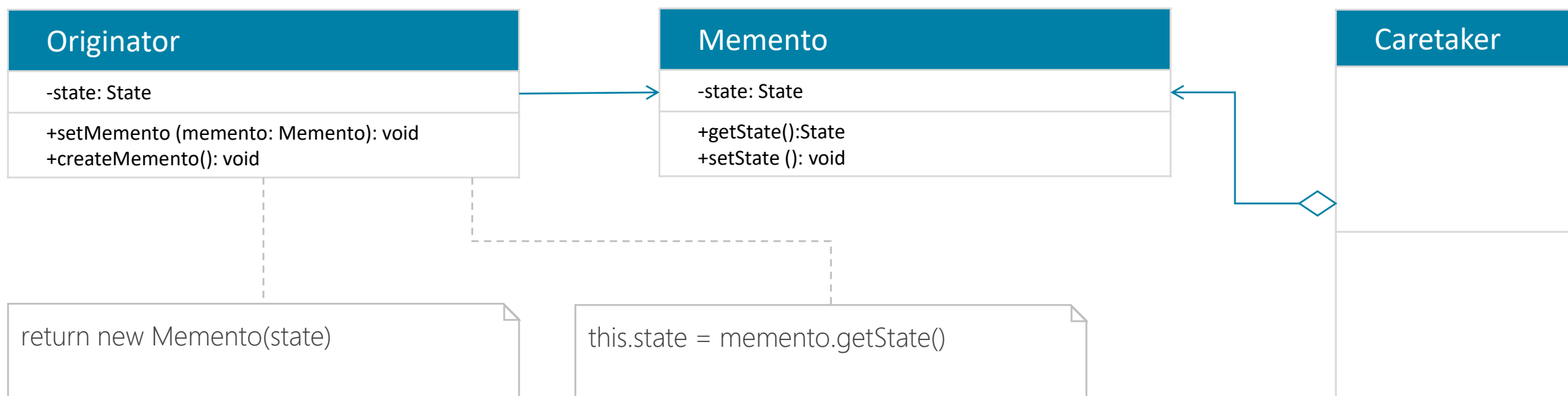


Design Pattern

Memento Pattern

Behavioral 模式

在不破壞封裝性的前提下，捕獲一個物件的內部狀態，並在該物件之外儲存這個狀態。

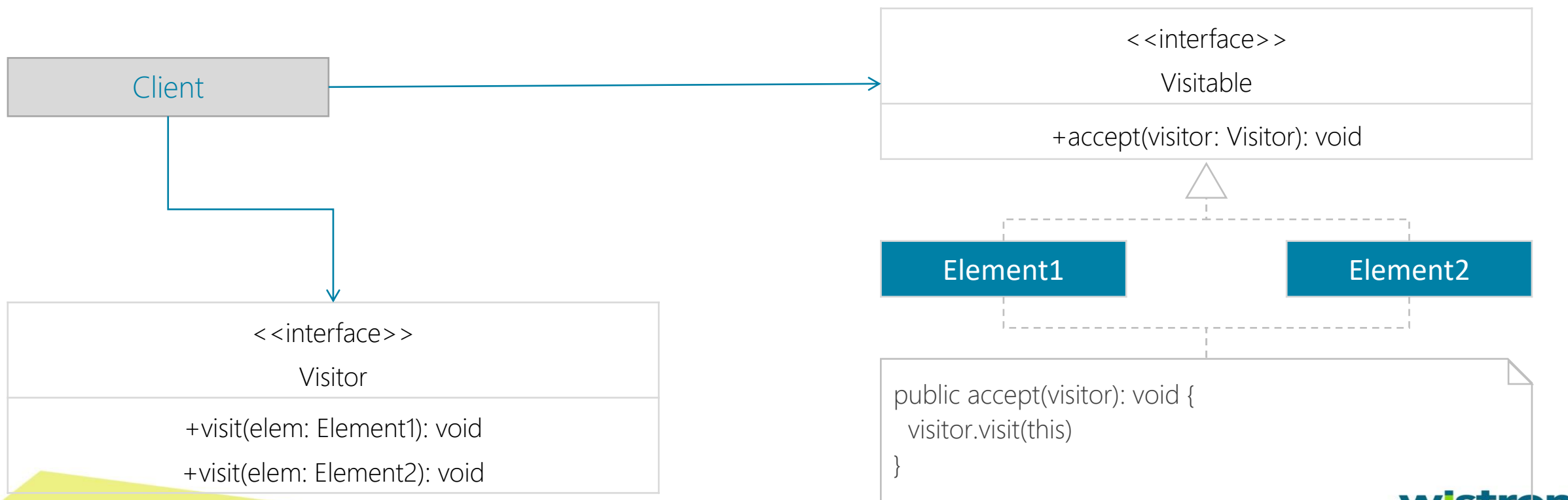


Design Pattern

Visitor Pattern

Behavioral 模式

訪問者是一個接口，它擁有一個visit方法，這個方法對訪問到的對象結構中不同類型的元素作出不同的反應。此Pattern在Typescript中較難還原，即使還原便利性也不及有支援Overload的語言

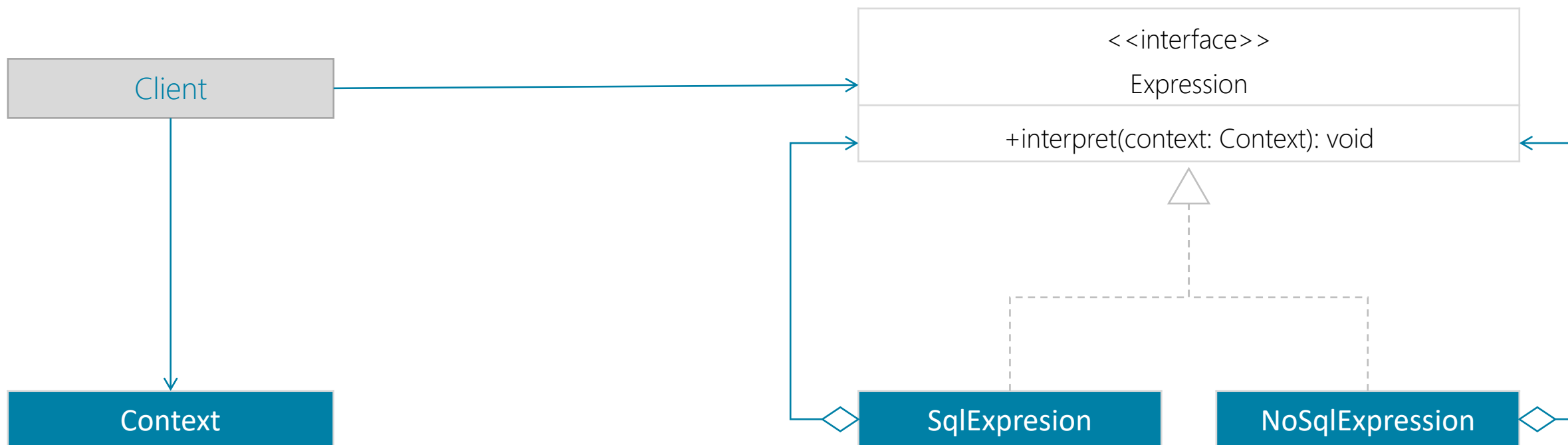


Design Pattern

Interpreter Pattern

Behavioral 模式

提供了評估語言的語法或表達式的方式，它屬於行為型模式。這種模式實現了一個表達式接口，該接口解釋一個特定的上下文。這種模式被用在 SQL 解析、符號處理引擎等。





Thank You !

Unit Test

單元測試(Unit Test)用來檢查一個獨立工作單位的行為。用來確認一個方法在接收特定的輸入後是否會得到預期的輸出。

常見問題

Q1

需要有完整的真實環境才能測試



A1

使用mock object來模擬外部回傳的資料

Q2

頁面出現問題，責任歸屬？



A2

把input值當做test case，跑Unit Test

Q3

交付過程，測過哪些功能



A3

交付的程式，要包含Unit Test

Q4

改了其中一支檔案，會不會影響整支程式



A4

改完程式就跑一次Unit Test

Q5

UI, Service, Data Access如何平行開發



A5

使用mock object，達到關注點分離

Unit Test

五個定義與五個準則

定義

最小測試單位

一個單元就是單個程式、函式、過程等

外部相依性為零

檔案讀取不到、網路斷線、套件錯誤都不應影響到測試的結果

不具有商業邏輯

不要有if else, for, try catch等，當裡面有判斷句，表示該測試進行多種使用情境

測試不依賴其他測試

不依賴流程（如測試新增->測試修改->測試刪除），這樣表示測試沒有獨立性

一個案例只測一件事

方便辨別哪個方法錯誤，就能找到那個使用情境的問題

準則

F.I.R.S.T

Fast

快速

Independent

獨立，不依賴外部資源

Repeatable

可重複，不會影響結果

Self-Validating

能驗證跑完「實際情況」跟「預期結果」是否吻合

Timely

及時。寫完Test即寫完Production Code

Unit Test

用3A原則寫單元測試

1

Arrange

初始化目標物件、參數，建立模擬物件，設定環境變數期望結果

2

Act

實際呼叫測試物件方法

3

Assert

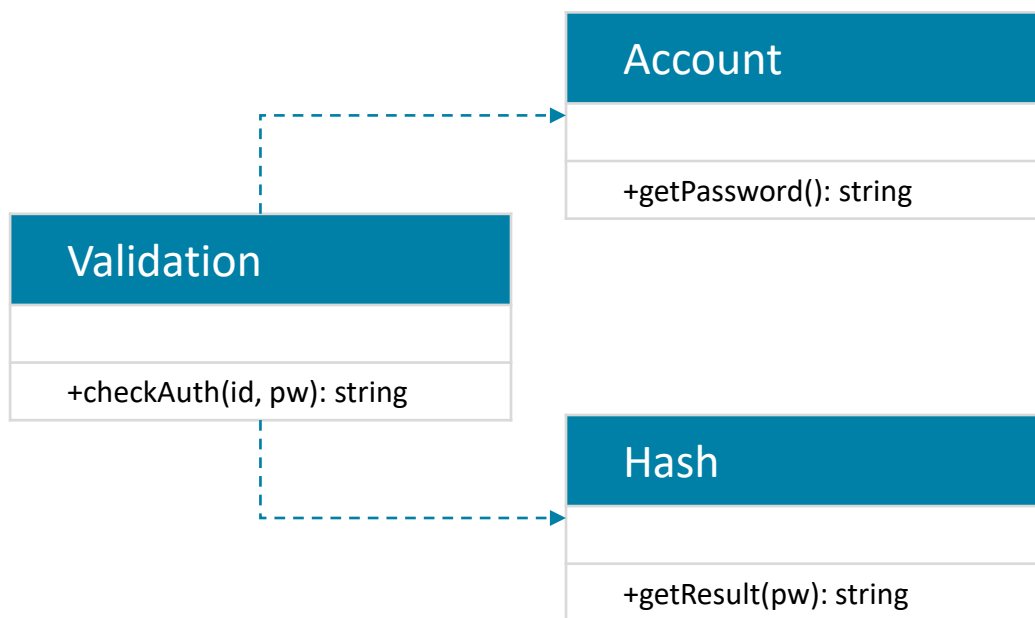
驗證目標物件是否符合預期

```
1  it('1 add 2 should equal 3', () => {
2    // Arrange
3    const calculator = new Calculator();
4    const firstNumber = 1;
5    const secondNumber = 2;
6    const expected = 3;
7
8    // Act
9    const actual = calculator.Add(firstNumber, secondNumber);
10
11   // Assert
12   expect(actual).toEqual(expected);
13 });
```

Unit Test

單元測試相依性

實際案例：有一個Validation類別，可以檢視登入帳號的正確性，其中有一個方法名為checkAuth，該方法可以取得密碼、取得hash結果、比對是否相同。嘗試利用3A原則建構單元測試，並觀察其中可能遇到的問題。



```
1  class Validation {
2      constructor() { }
3      public checkAuth(id: string, password: string): boolean {
4          const account = new GoogleAccount();
5          const matchedPassword = account.getPassword(id);
6
7          const hash = new Hash();
8          const encryPassword = hash.getResult(password);
9          return matchedPassword === encryPassword;
10     }
11 }
```

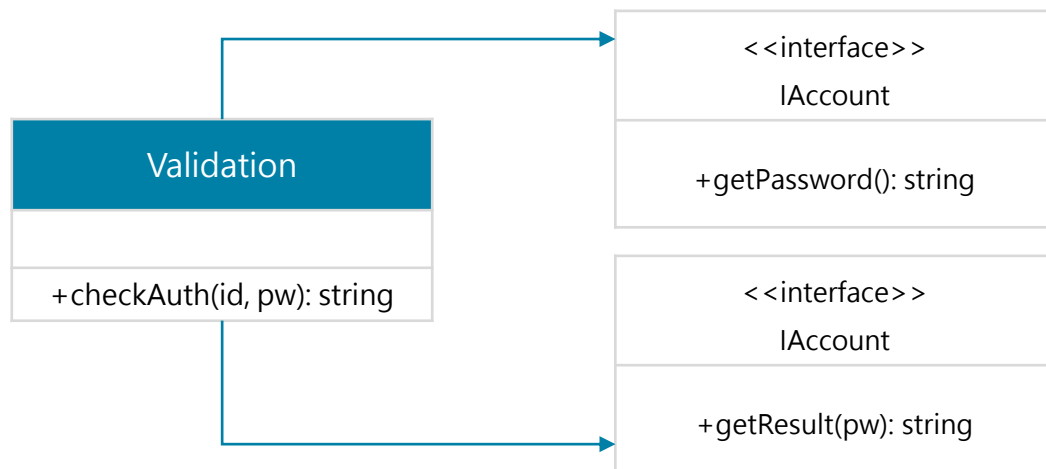
Unit Test

單元測試相依性

問題解析&程式重構

上述的例子，因為checkAuth依賴GoogleAccount及Hash，因此我們勢必要產生兩者的實例，但若其背後夾帶著網路請求或資料庫的連線，則我們要確保在環境都沒問題的情況下才有辦法測試。

上述狀況違反了不依賴外部相依性的定義，這個時候我們必須重構Validation，來解偶GoogleAccount及Hash對Validation的關係



```
1  class Validator {
2      private account: IAccount;
3      private hash: IHash;
4
5      constructor(account: IAccount, hash: IHash) {
6          this.account = account;
7          this.hash = hash;
8      }
9
10     public checkAuth(id: string, password: string): boolean {
11         const matchedPassword = this.account.getPassword(id);
12         const encryPassword = this.hash.getResult(password);
13         return matchedPassword === encryPassword;
14     }
15 }
```

Unit Test

單元測試相依性

1

Arrange

- 初始化MockAccount
- 初始化MockHash
- 初始化Validation

2

Act

呼叫checkAuth

3

Assert

期望透過MockAccount取得的密碼與透過MockHash加密過後的密碼相符

```
1  it(`account.getPassword` must equal `hash.getResult`, () => {
2    // Arrange
3    const account = new MockAccount();
4    const hash = new MockHash();
5    const validation = new Validation(account, hash);
6
7    // Act
8    const result = validation.checkAuth('id', 'password');
9
10   // Assert
11   expect(result).toBeTrue();
12 }
```

結論: 從本例子可以知道，要能夠順利進行單元測試，code本身就須遵守S.O.L.I.D，且要是低耦合、高內聚。程式若不具備可測試性，代表其物件設計不夠良好；程式具備可測試性，也並不表示該物件設計就一定良好，但起碼有一定的品質。

Unit Test

Angular單元測試使用工具

Jasmine 框架 + Karma 測試執行工具是 Angular 普遍使用的單元測試組合，也是官方推薦的方法。接下來會進行其介紹，並說明設定以及對應的指令。



Jasmine就是一種JavaScript單元測試框架，它不依賴任何其他JS框架，也不需要對DOM操作，具有靈巧而明確的語法可以編寫測試程式碼。它是一套Javascript行為驅動開發框架(BDD)，乾淨簡潔、表達力強且易於組織，不依賴於其他任何框架和DOM，可運行於Node.js，瀏覽器端或移動端



Karma是一個基於Node.js的JavaScript測試執行過程管理工具（Test Runner），這個測試工具的一個強大特性就是，它可以監控(Watch)變化，然後自行執行，通過console.log顯示測試結果。

Unit Test

Karma設定

由於ng test時會啟用Karma顯示幕前測試的結果，因此為了避免在Gitlab CICD的過程中因無法顯示瀏覽器而導致測試失敗，必須在設定檔中讓chrome使用headless的模式，並加入 `--no-sandbox` 參數，來讓測試的過程可以順利完成。

karma.conf.js

```
1  module.exports = function (config) {
2    config.set({
3      ...,
4      browsers: ['ChromeHeadlessCI'],
5      customLaunchers: {
6        ChromeHeadlessCI: {
7          base: 'ChromeHeadless',
8          flags: ['--no-sandbox']
9        }
10     },
11     singleRun: false,
12     restartOnFileChange: true
13   });
14 }
```

Unit Test

Jasmine使用說明

在Angular當中，測試的檔案會以 `.spec.ts` 作為結尾，並在其中包含測試的描述就會進行測試驗證，code會如右下所示。

- 在Command Line當中輸入以下指令即可啟動測試

```
> ng test
```

- 使用參數 `--browsers` 可以指定要觸發的瀏覽器

```
> ng test --browsers=Chrome
```

- 使用參數 `--code-coverage` 可以產生覆蓋率報表

```
> ng test --code-coverage
```

- 使用參數 `--source-map` 產生lcov.info，供sonarqube顯示覆蓋率

```
> ng test --source-map
```

app.component.spec.ts

```
1 describe('AppComponent', () => {
2   beforeEach(
3     async(() => {
4       TestBed.configureTestingModule({
5         imports: [RouterTestingModule],
6         declarations: [AppComponent],
7       }).compileComponents();
8     })
9   );
10
11   it('should create the app', () => {
12     const fixture = TestBed.createComponent(AppComponent);
13     const app = fixture.componentInstance;
14     expect(app).toBeTruthy();
15   });
16 });
```

Unit Test

Angular使用Jasmine單元測試實作

當前有一個情境，有一個Guard會協助驗證登入帳號是否具備管理員的權限，如果有允許存取該路由，沒有則跳轉回登入頁面。

authorization.guard.ts

```
1 | @Injectable({
2 |   providedIn: 'root',
3 | })
4 | export class AuthorizationGuard implements CanActivate {
5 |   constructor(
6 |     private readonly router: Router,
7 |     private readonly userApi: UserControllerService
8 |   ) {}
9 |
10 |   public async canActivate(): Promise<boolean> {
11 |     try {
12 |       await this.userApi.userControllerIsAdmin().toPromise();
13 |       return true;
14 |     } catch (error) {
15 |       this.router.navigate(['/login']);
16 |       return false;
17 |     }
18 |   }
19 | }
```

authorization.guard.spec.ts



Unit Test

Angular使用Jasmine單元測試實作

1 jasmine.createSpyObj

建立Spy物件以方便模擬所需的來源物件或參數

2 TestBed.configureTestingModule

注入當前測試元件或服務所相依的Mock Object

3 TestBed.inject

注入要測試的元件或服務

4 and.returnValue

Spy一個假數據，做為測試所需

```
1 describe('AuthorizationGuard', () => {
2   const router = jasmine.createSpyObj<Router>('Router', ['navigate']);
3   const apiConfiguration = jasmine.createSpyObj<
4     Configuration
5   >('Configuration', [null]);
6   ...
7
8   beforeEach(() => {
9     2 / 引入Module
10    TestBed.configureTestingModule({
11      imports: [HttpClientModule, RouterModule],
12      providers: [{ provide: Router, useValue: router, }, ...],
13      ...
14    });
15    3 guard = TestBed.inject(AuthorizationGuard);
16  });
17
18  it('should ...', async () => {
19    const http200 = new HttpResponse({ status: 200 });
20    apiConfiguration.accessToken 4 emulated.jwt.token';
21    userApi.userControllerIsAdmin.and.returnValue(http200);
22    const accessible = await guard.canActivate(null, null);
23    expect(accessible).toBeTrue();
24  });
25 });
```

Unit Test

Angular Unit Test Code Coverage

下方結果可透過`--code-coverage`參數產生，並寫可利用`--source-map`參數產生lcov.info，有了這個檔案就可以將測試的Code Coverage整合到Sonarqube中，具體的設定方式，會在Gitlab CI章節中說明。

All files									
7.38% Statements 75/1016 0.68% Branches 7/1034 14.43% Functions 14/97 6.28% Lines 62/987									
Press n or j to go to the next uncovered block, b, p or k for the previous block.									
File		Statements		Branches		Functions		Lines	
src	<div></div>	100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app	<div></div>	100%	11/11	100%	0/0	100%	1/1	100%	9/9
src/app/core/services/api	<div></div>	38.89%	14/36	12.5%	2/16	25%	3/12	39.39%	13/33
src/app/core/services/api/api	<div></div>	1.41%	13/922	0%	0/1010	0%	0/70	0.77%	7/904
src/app/core/services/auth/token	<div></div>	66.67%	2/3	100%	0/0	0%	0/1	50%	1/2
src/app/core/services/config	<div></div>	100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/app/core/services/storage	<div></div>	46.15%	6/13	0%	0/2	60%	3/5	54.55%	6/11
src/app/core/services/test	<div></div>	83.33%	5/6	100%	0/0	75%	3/4	80%	4/5
src/app/pages/login/services	<div></div>	100%	3/3	100%	0/0	100%	1/1	100%	2/2
src/app/shared/guards	<div></div>	94.12%	16/17	83.33%	5/6	100%	2/2	93.75%	15/16

E2E Test

Protractor

Protractor是Angular端到端測試框架，測試會放在目錄e2e底下，測試的檔案會以 `.e2e-spec.ts` 作為結尾，並在其中包含測試的描述就會進行測試驗證，code會如右下所示。

- 啟動e2e測試

```
> ng e2e
```

- 使用參數`--webdriverUpdate`，選擇driver是否做更新

```
> ng e2e --webdriverUpdate=false
```

- 使用參數`--protractor-config`，選擇設定檔來源

```
> ng e2e --protractor-config=e2e/protractor.conf.js
```

```
1 describe('workspace-project App', () => {
2   let page: AppPage;
3
4   beforeEach(() => {
5     page = new AppPage();
6   });
7
8   it('should redirect to login page', async () => {
9     await page.navigateTo();
10    const url = await page.getCurrentUrl();
11    expect(url).toEqual(`${browser.baseUrl}login`);
12  });
13
14  afterEach(async () => {
15    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
16    expect(logs).not.toContain(jasmine.objectContaining({
17      level: logging.Level.SEVERE,
18    } as logging.Entry));
19  });
20 });
```

E2E Test

Protractor設定

由於e2e測試啟用Browser模擬使用者的操作行為，因此為了避免在Gitlab CICD的過程中因無法顯示瀏覽器而導致測試失敗，必須在設定檔中讓chrome使用headless的模式，並加入以下設定，來讓測試的過程可以順利完成。

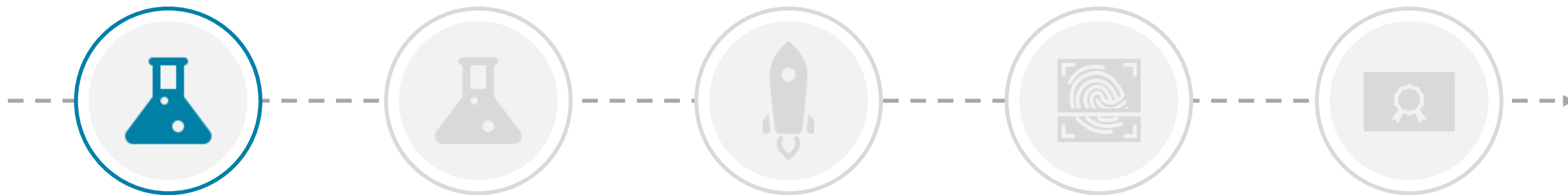
e2e/protractor.conf.js

```
1  exports.config = {
2    ...,
3    chromeDriver: '../driver/chromedriver',
4    capabilities: {
5      browserName: 'chrome',
6      chromeOptions: {
7        args: [
8          '--headless',
9          '--no-sandbox',
10         '--disable-setuid-sandbox',
11         '--disable-gpu',
12         '--disable-dev-shm-usage',
13         '--disable-extensions',
14         '--disable-infobars'
15       ]
16     }
17   },
18   ...
19 };
```




Thank You !

Gitlab CI



Stage: Test

Unit Test

1. 完成Unit Test Code
2. 調整Karma.conf.js
3. 增加.gitlab-ci.yml Stage
4. 須將code coverage加到artifacts

.gitlab-ci.yml

```
1  ng-unit-test:
2    stage: test
3    dependencies:
4      - install-package
5    except:
6      - tags
7    before_script:
8      - npm i -g @angular/cli
9    script:
10     - npm run test:prod
11    tags:
12     - prd-runner04
13    artifacts:
14     paths:
15       - ./coverage/
16     when: always
17     expire_in: 1 days
```

Gitlab CI



Stage: Test

E2E Test

1. 完成E2E Test Code
2. 調整protractor.conf.js
3. 增加.gitlab-ci.yml Stage

.gitlab-ci.yml

```
1  ng-e2e-test:
2    stage: test
3    dependencies:
4      - install-package
5    except:
6      - tags
7    before_script:
8      - npm i -g @angular/cli
9      - chmod +x ./driver/chromedriver
10   script:
11     - npm run e2e:prod
12   tags:
13     - prd-runner04
```

Gitlab CI



Stage: build_for_test

build_test_image

1. 完成Unit Test Code
2. 調整Karma.conf.js
3. 增加.gitlab-ci.yml Stage
4. 須將code coverage加到artifacts

.gitlab-ci.yml

```
1 build_test_image:
2   only:
3     - master
4     - production
5     - /^pre-production.*$/
6   stage: build_for_test
7   image: harbor.wistron.com/base_image/docker:stable
8   tags:
9     - prd-runner04
10  variables:
11
12  script:
13    - echo "this is docker image packing for TEST"
14    - docker build -t ${BUILD_IMAGE_NAME}:${DOCKER_TEST_IMAGE_TAG} --rm=true .
15    - docker tag ${BUILD_IMAGE_NAME}:${DOCKER_TEST_IMAGE_TAG} ${HARBOR_URL}/${HARBOR_PROJECT}/${BUILD_IMAGE_NAME}:${DOCKER_TEST_IMAGE_TAG}
16    - echo "${HARBOR_PASSWORD}" | docker login -u "${HARBOR_USER}" --password-stdin ${HARBOR_URL}
17    - docker push ${HARBOR_URL}/${HARBOR_PROJECT}/${BUILD_IMAGE_NAME}:${DOCKER_TEST_IMAGE_TAG}
```

Gitlab CI



Stage: deploy_for_test

deploy_for_test

1. 完成Unit Test Code
2. 調整Karma.conf.js
3. 增加.gitlab-ci.yml Stage
4. 須將code coverage加到artifacts

.gitlab-ci.yml

```
1 deploy_for_test:
2   only:
3     - master
4     - production
5     - /^pre-production.*$/
6   stage: deploy_for_test
7   image: harbor.wistron.com/base_image/rancher-deploy-tool:latest
8   tags:
9     - prd-runner04
10  variables:
11  script:
12    - ls -al
13    - deploy -n ${SERVICE_NAME} -i ${HARBOR_URL}/${HARBOR_PROJECT}/${BUILD_IMAGE_NAME}:${DOCKER_TEST_IMAGE_TAG} -m 4200
```

Gitlab CI



Stage: vulnerability_scan

vulnerability_scan

1. 完成Unit Test Code
2. 調整Karma.conf.js
3. 增加.gitlab-ci.yml Stage
4. 須將code coverage加到artifacts

.gitlab-ci.yml

```
1 vulnerability_scan:
2   only:
3     - master
4     - production
5     - /^pre-production.*$/
6   image: harbor.wistron.com/base_image/zap2docker-stable:latest
7   tags:
8     - prd-runner04
9   stage: vulnerability_scan
10  before_script:
11    - cp -r ./dist/* /usr/share/nginx/html
12    - cp ./nginx-custom.conf /etc/nginx/conf.d/default.conf
13    - /etc/init.d/nginx start
14    - . ci-funcs.sh
15  script:
16    - zap_scan
```

Gitlab CI



Stage: vulnerability_scan

vulnerability_scan

1. 完成Unit Test Code
2. 調整Karma.conf.js
3. 增加.gitlab-ci.yml Stage
4. 須將code coverage加到artifacts

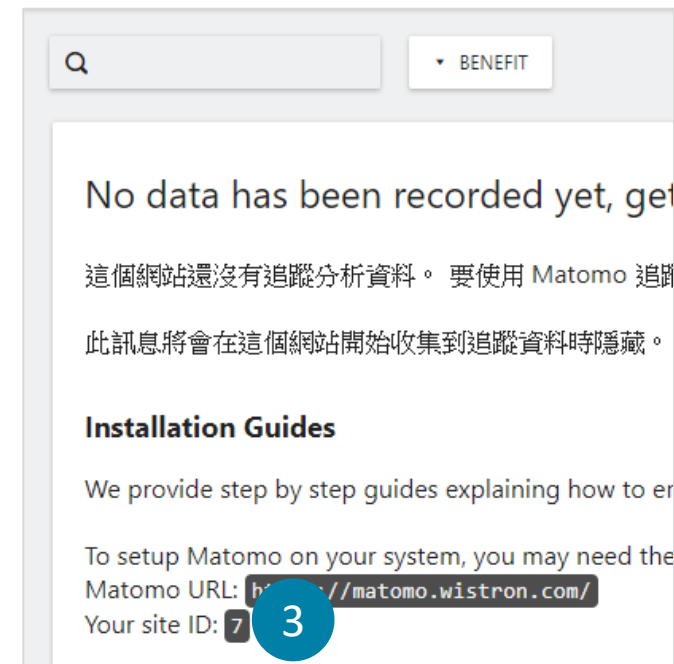
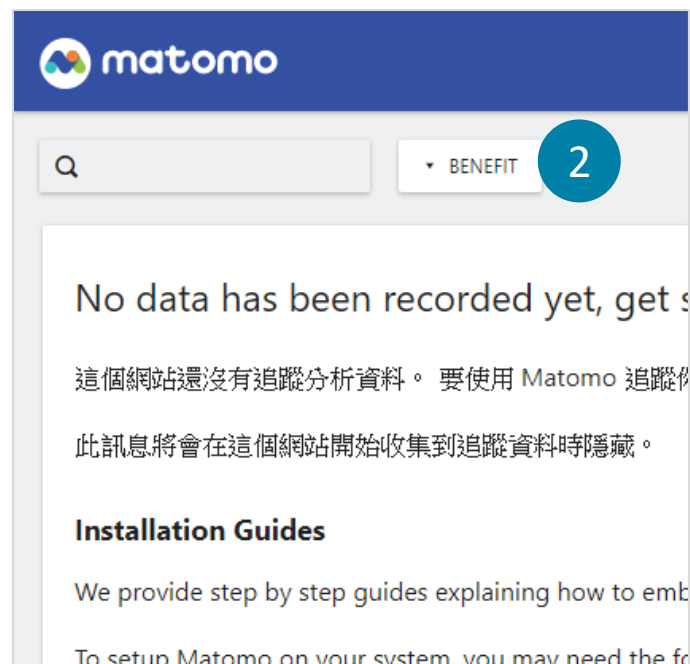
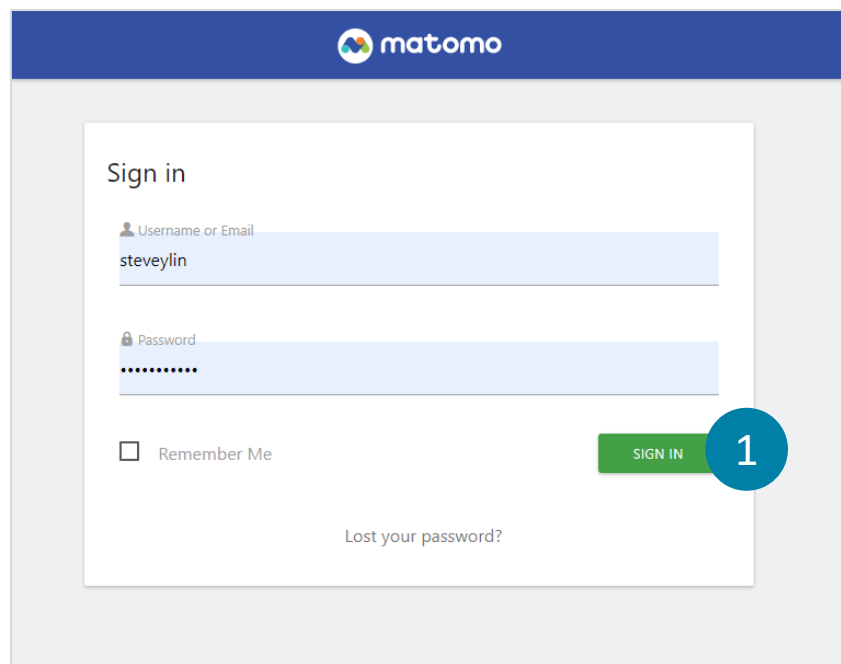
.gitlab-ci.yml

```
1  code_scan:
2    only:
3      - master
4      - production
5      - /^pre-production.*$/
6    image: harbor.wistron.com/base_image/sonar-scanner-cli:4
7    except:
8      - tags
9    stage: code_scan
10   tags:
11     - prd-runner04
12   script:
13     - sonar-scanner -Dsonar.projectName=$CI_PROJECT_NAME -Dsonar.projectKey=$CI_PROJECT_NAME -
      Dsonar.sources=. -Dsonar.host.url=${SONAR_URL} -Dsonar.login=${SONAR_TOKEN} -
      Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info -
      Dsonar.exclusions=src/app/core/services/api/** -Dsonar.zaproxxy.reportPath=./report.xml
```



Thank You !

初始化Project



Matomo

1. 進到Project目錄底下，安裝ngx-matomo

```
> npm install ngx-matomo --save
```

2. 加入以下代碼，左右兩種方法擇一即可

index.html(加到head當中)

```
1 <script type="text/javascript">
2   var _paq = window._paq || [];
4   _paq.push(['trackPageView']);
5   _paq.push(['enableLinkTracking']);
6   (function() {
7     var u="https://matomo.wistron.com/";
8     _paq.push(['setTrackerUrl', u+'matomo.php']);
9     _paq.push(['setSiteId', '7']);
10    var d=document,
11        g=d.createElement('script'),
12        s=d.getElementsByTagName('script')[0];
13    g.type='text/javascript';
14    g.async=true;
15    g.defer=true;
16    g.src=u+'matomo.js';
17    s.parentNode.insertBefore(g,s);
18  })();
19 </script>
```

app.module.ts

```
1 import { MatomoModule } from 'ngx-matomo';
2
4 @NgModule({
5   imports: [MatomoModule],
6   ...
7 })
```

app.component.ts

```
1 import { MatomoInjector } from 'ngx-matomo';
2
4 export class AppComponent {
5   constructor(private matomo: MatomoInjector) {
6     this.matomo.init('https://matomo.wistron.com/', 8);
7   }
8 }
```

完成以上步驟後即可在Matotmo上的Dashboard看到當前的訪客數據





Tracing Page View

加入以下代碼，讓網頁標題可以被追蹤，下列兩種方法擇一即可。

index.html(加到head當中)

```
1 <script type="text/javascript">
2   var _paq = window._paq || [];
4   _paq.push(['setDocumentTitle', `${document.domain}/${document.title}`]);
5   _paq.push(['trackPageView']);
6   _paq.push(['enableLinkTracking']);
7   ...
8 </script>
```

加到要追蹤的Component當中

```
1 import { MatomoInjector, MatomoTracker } from 'ngx-matomo';
2
4 @Component({ ... })
5 export class AppComponent implements OnInit {
6   constructor(
7     private readonly matomoInjector: MatomoInjector,
8     private readonly matomoTracker: MatomoTracker
9   ) {
10     this.matomoInjector.init('https://matomo.wistron.com/', 8);
11   }
12   public ngOnInit(): void {
13     this.matomoTracker.trackPageView('your_view_name');
14   }
15 }
```

Tracing Page View

上述的範例可以使用Decorator Pattern，讓追蹤的邏輯與當前Function內所做的邏輯做切割，使其符合單一職責原則。

page-view.decorator.ts

```
1 import { MatomoTracker } from 'ngx-matomo';
2
3 export function PageView(viewName: string = 'app') {
4     const matomoTracker = new MatomoTracker();
5
6     return (
7         target: any,
8         propertyKey: string,
9         descriptor: PropertyDescriptor
10    ) => {
11        const original = descriptor.value;
12        descriptor.value = function(args: any) {
13            matomoTracker.trackPageView(viewName);
14            original.apply(this, args);
15        };
16    };
17 }
```

加到要追蹤的Component當中

```
1 @Component({ ... })
2 export class AppComponent implements OnInit {
3     constructor(
4         private readonly matomoInjector: MatomoInjector,
5         private readonly matomoTracker: MatomoTracker
6     ) {
7         this.matomoInjector.init('matamo.com', 8);
8     }
9     @PageView('view.name')
10    public ngOnInit(): void {
11        // TODO
12    }
13 }
14
15
16
17
```

Tracing Event

加入以下代碼，讓事件的觸發可以被追蹤，**下列兩種方法擇一即可**。

index.html(加到head當中)

```
1 <script>
2   var _paq = window._paq || [];
4   _paq.push(['setDocumentTitle', `${document.domain}/${document.title}`]);
5   _paq.push(['trackEvent', 'Query Data', 'inquiry', 'module', 5]);
6 </script>
```

加到要追蹤的Component當中

```
1 @Component({ ... })
2 export class MeterKanbanComponent OnInit {
4   constructor(private readonly matomoTracker: MatomoTracker) { }
5
6   public queryData(): void {
7     this.matomoTracker.trackEvent('Query Data', 'inquiry', 'module', 5);
8     // TODO Query Data
9   }
10 }
```

1. 事件類別(必填)
2. 事件動作(必填)
3. 事件名稱(選填)
4. 事件評分(選填)

Tracing Event

使用Decorator Pattern，讓追蹤的邏輯與當前Function內所做的邏輯做切割，使其符合單一職責原則。

track-event.decorator.ts

```
1 import { MatomoTracker } from 'ngx-matomo';
2
3 export function TrackEvent(params: TrackEventParamsModel) {
4   const matomoTracker = new MatomoTracker();
5   return (
6     target: any,
7     propertyKey: string,
8     descriptor: PropertyDescriptor
9   ) => {
10     const original = descriptor.value;
11     descriptor.value = function(args: any) {
12       matomoTracker.trackEvent(
13         params.category,
14         params.action,
15         params.name,
16         params.value
17       );
18       original.apply(this, args);
19     };
20   };
21 }
```

加到要追蹤的Component當中

```
1 @Component({ ... })
2 export class MeterKanbanComponent implements OnInit {
3   constructor() {}
4
5   @TrackEvent({
6     category: 'Query Data',
7     action: 'inquiry',
8     name: 'data-analysis.module',
9     value: 3
10  })
11   public queryData(): void {
12     // TODO Query Data
13   }
14 }
15
16
17
```



Thank You !