معماری MV

مهدی عباس تبار - یاسمین کدخدایی

معرفی

معماری *MV یک الگوی طراحی معماری نرم افزار متداول است که توسط چارچوب های وب مانند Ruby on Rails و AngularJS با ظهور برنامه های تک صفحه ای محبوب شده است. *MV با اصل SoC برنامه را به سه جزء اصلی تقسیم می‌کند و نحوه تعامل اجزا را تعریف می‌کند. این امر اتصال بین اجزا را کاهش می دهد و اجازه می دهد تا هر جزء به طور مستقل توسعه، آزمایش، اصلاح، استفاده مجدد و مقیاس بندی شود. *MV یک اصطلاح کلی است که انواع این الگوی معماری را در بر می‌گیرد و از نظر نوع و سطح جفت شدن بین اجزاء متفاوت است که در ادامه به آنها می‌پردازیم.

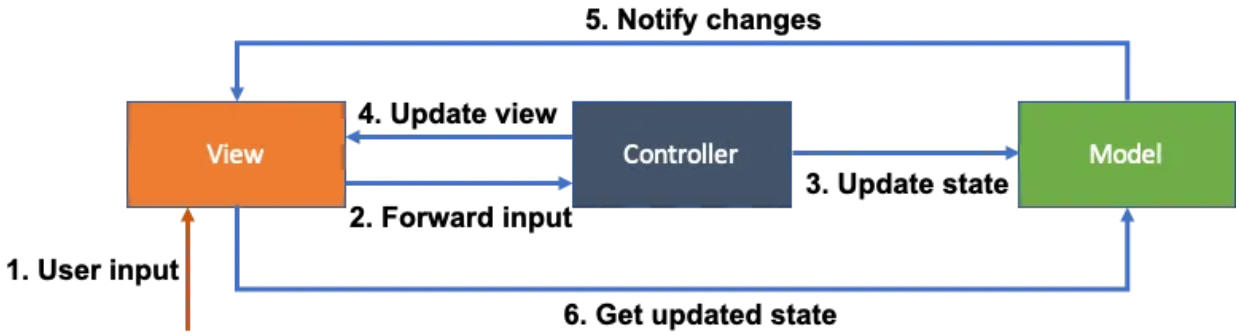
دیاگرامی کلی از معماری MV

این معماری از چه بخش‌هایی تشکیل شده است؟

ابتدا اجزای ثابت این معماری که M و V هستند را معرفی می‌کنیم:
لایه‌ی Model: این لایه مربوط به هندل کردن business logic برنامه و لایه‌ی دیتا است که عمدتا محاسبات و ارتباط با پایگاه داده را بر عهده دارد.
لایه‌ی View: این لایه عمدتا مربوط به نمایش رابط کاربری که شامل ساختار و طرح‌بندی است می‌شود.
در اصل، Model مشخص می‌کند که داده‌ها چیستند و View مشخص می‌کند که چگونه داده‌ها به کاربر ارائه شود. چه اتفاقی می‌افتد وقتی یک کاربر با رابط تعامل دارد، داده‌ها باید تغییر کنند؟ نحوه

تعامل Model و View با یکدیگر توسط مؤلفه سوم تعیین می شود که به نوع *MV مورد نظر بستگی دارد.
سه مدل معماری را در ادامه با هم بررسی میکنیم.

معماری MVC یا Model View Controller



دیاگرامی از معماری MVC

MVC دارای یک کنترلر است که منطق کنترل ورودی کاربر را کنترل می کند. ورودی هایی که به لایه View منتقل شده اند را را پردازش می کند و به روز رسانی ها و آپدیت ها را به View و Model ارسال می کند. تفاوت‌های جزئی در نحوه تعریف عملیات‌ها در MVC وجود دارد، اما به طور کلی آنها یک ویژگی مشترک دارند: View مستقیماً با مدل تعامل دارد تا وضعیت و داده‌های به‌روز شده را دریافت کند، بنابراین View باید اطلاعاتی درباره آن بداند. همچنین، View و Controller کاملاً با هم جفت شده‌اند، زیرا کنترلر باید اطلاعاتی درباره View بداند تا به View نحوه نمایش اطلاعات را بگوید.

مثالی از برنامه ای که با معماری MVC بنا نهاده شده است

با مثال کلیک بر روی یک دکمه که موجب آپدیت شدن یک دیتا می‌شود سعی می‌کنیم که بهتر مفهوم رو متوجه شویم. مرحله به مرحله را بررسی می‌کنیم:
ابتدا کاربر بر روی دکمه کلیک می کند.
سپس از طریق view، یک event صدا زده می‌شود.
پس از این controller به model اطلاع می‌دهد که دیتا را به روز رسانی کند و در صورت موفقیت آمیز بودن عملیات، alert مربوطه را نمایش می‌دهد.
در قدم بعدی model که دیتا را به روز کرده است به view اطلاع میدهد که دیتا آپدیت شده است.

قبل از این که view اطلاعات را به کاربر نمایش دهد، باید کوئری بزند و لیست به روز شده را از آن دریافت کند. یک مثال از کدی که با این معماری زده شده را در ادامه مشاهده می‌کنیم:

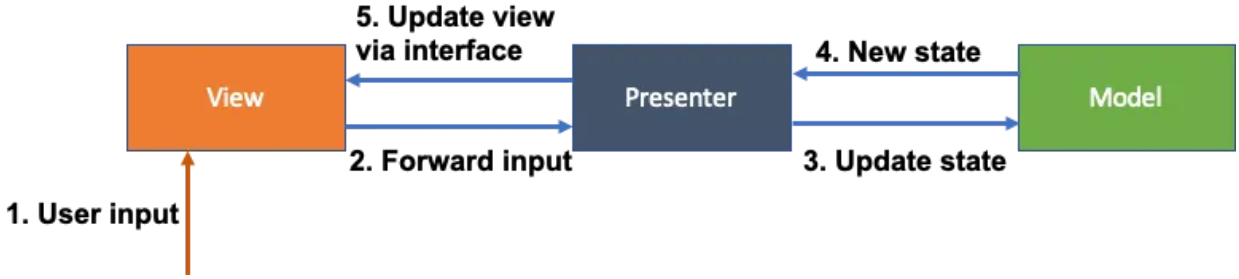
```
[ ]  class Model {
         constructor() {
             this.data = [];
         }

         getData() {
             return this.data;
         }

         addData(item) {
             this.data.push(item);
             return this.data;
         }
     }

     module.exports = Model;
```

```
 ]  class View {
        constructor() {
            this.button = document.getElementById('button');
            this.output = document.getElementById('output');
        }

        bindAddData(handler) {
            this.button.addEventListener('click', () => {
                const item = document.getElementById('input').value;
                handler(item);
            });
        }

        displayData(data) {
            this.output.innerHTML = data.join(', ');
        }
    }

    module.exports = View;
```

```
class Controller {
    constructor(model, view) {
        this.model = model;
        this.view = view;

        this.view.bindAddData(this.handleAddData.bind(this));
    }

    handleAddData(item) {
        const data = this.model.addData(item);
        this.view.displayData(data);
    }
}

const Model = require('./model');
const View = require('./view');
const Controller = require('./controller');

const app = new Controller(new Model(), new View());
```

معماری MVP یا Model View Presenter

این معماری تلاش دارد که جفت شدن یا coupling را کاهش دهد. Presenter که در این معماری داریم، واسطه بین view و model محسوب می‌شود. این واسطه بودن باعث می‌شود که الزامی نداشت باشیم که view اطلاعاتی در مورد model داشته باشد. ورودی از view به model پاس داده میشود و response از model به view.



دیاگرامی از معماری MVP (نمای passive)

دو تفاوت اصلی در تعامل view presenter در مقایسه با تعامل view controller وجود دارد. اولاً، view در MVP سبک تر است زیرا منطق presentation توسط presenter مدیریت می شود. (توجه داشته باشید که درجه منطق ارائه در view ممکن است بر اساس کاربرد و پیاده سازی متفاوت باشد.) ثانیاً presenter اکنون به جای این که مستقیم ارتباط داشته باشد، از طریق یک رابط با view تعامل دارد، بنابراین دیگر نیازی به دانستن اطلاعات در مورد view ندارد. در عوض، presenter به view می‌گوید که چه چیزی را نمایش دهد، و view که رابط یا interface را پیاده‌سازی می‌کند، آن را نمایش می‌دهد.در حالت passive معماری MVP مدل همیشه استیت آپدیت شده را به view از طریق presenter پاس می دهد. همچنین یک نوع نظارت کنترلر دیگر وجود دارد که در آن presenter مدل را به View برای اتصال داده ها ارسال می کند، به طوری که به روز رسانی های حالت ساده را می توان بدون presenter انجام داد. با این حال هم به روز رسانی های پیچیده تر باید از طریق presenter ارسال شوند.

مثالی از برنامه ای که با معماری MVP بنا نهاده شده است

با مثال کلیک بر روی یک دکمه که موجب آپدیت شدن یک دیتا می‌شود سعی می‌کنیم که بهتر مفهوم رو متوجه شویم. مرحله به مرحله را بررسی می‌کنیم:

کاربر روی دکمه کلیک می کند.

ابتدا view به presenter ایونت را ارسال میکند.

لایه presenter به model اطلاع میدهد که دیتا را آپدیت کند.
پس از این که model دیتا را آپدیت کرد، دیتای آپدیت شده را به presenter پاس میدهد.
در مرحله آخر presenter به view اطلاع میدهد که دیتا آپدیت شده و این را با یک alert مناسب به کاربر اطلاع میدهد. یک مثال از کدی که با این معماری زده شده را در ادامه مشاهده می‌کنیم:

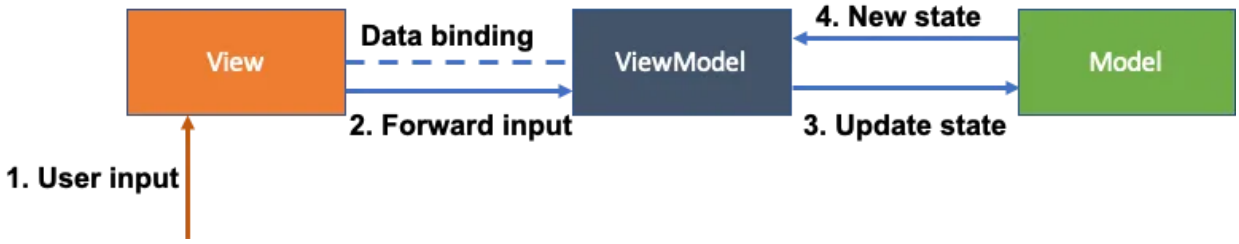
```
[ ]  class Model {
         constructor() {
             this.data = [];
         }

         getData() {
             return this.data;
         }

         addData(item) {
             this.data.push(item);
             return this.data;
         }
     }

     module.exports = Model;
```

```
 ]  class View {
        constructor() {
            this.button = document.getElementById('button');
            this.output = document.getElementById('output');
        }

        bindAddData(handler) {
            this.button.addEventListener('click', () => {
                const item = document.getElementById('input').value;
                handler(item);
            });
        }

        displayData(data) {
            this.output.innerHTML = data.join(', ');
        }
    }

    module.exports = View;
```

```
]  class Presenter {
       constructor(model, view) {
           this.model = model;
           this.view = view;

           this.view.bindAddData(this.handleAddData.bind(this));
       }

       handleAddData(item) {
           const data = this.model.addData(item);
           this.view.displayData(data);
       }
   }

   const Model = require('./model');
   const View = require('./view');
   const Presenter = require('./presenter');

   const app = new Presenter(new Model(), new View());
```

معماری MVVM یا Model View ViewModel

واریته supervising controller از MVP، آخرین نوع معماری ای که به آن میخواهیم بپردازیم را توضیح میدهد. بین این معماری و MVP شباهت‌هایی وجود دارد از جمله در هر دو این معماری ها view خیلی سبک است و این view است که منطق presentation را مدیریت می کند. البته View Model وضعیت view را نیز محصور می کند و از طریق اتصال داده ها، دستورات و alert ها با view تعامل می کند. می‌توانیم View Model را به عنوان زیرمجموعه‌ای از Model در نظر بگیریم، و بنابراین فقط داده‌های مربوط به استیت در معرض view قرار می‌گیرند.



دیاگرامی از مدل معماری MVVM

مثالی از برنامه ای که با معماری MVC بنا نهاده شده است

با مثال کلیک بر روی یک دکمه که موجب آپدیت شدن یک دیتا می‌شود سعی می‌کنیم که بهتر مفهوم رو متوجه شویم. مرحله به مرحله را بررسی می‌کنیم:

کاربر روی دکمه کلیک می کند

از طرف view ایونت با یک کامند به سمت ViewModel فرستاده می‌شود.

از ViewModel اطلاع رسانی ای به Model فرستاده میشود که دیتای مورد نظر را آپدیت کنند.

پس از این که Model دیتای مورد نظر را آپدیت کرد، مقدار جدید دیتا را به ViewModel ارسال می‌کند که این مقدار جدید در view هم آپدیت می شود و نمایش داده میشود.

با توجه به این که view به صورت اتوماتیک sync است، ViewModel دیگر نیاز نیست که حتما به view بگوید که فلان دیتا را آپدیت کن. از این روی coupling هم کاهش پیدا میکند. یک مثال از کدی که با این معماری زده شده را در ادامه مشاهده می‌کنیم:

```
[ ]  class Model {
         constructor() {
             this.data = [];
         }

         getData() {
             return this.data;
         }

         addData(item) {
             this.data.push(item);
             return this.data;
         }
     }

     module.exports = Model;
```

```
[ ]  <!DOCTYPE html>
     <html>
     <head>
         <title>MVVM Example</title>
     </head>
     <body>
         <input type="text" id="input">
         <button id="button">Add</button>
         <div id-"output"></div>
         <script src="viewmodel.js"></script>
     </body>
     </html>
```

```
[ ]  class ViewModel {
         constructor(model) {
             this.model = model;
             this.button = document.getElementById('button');
             this.output = document.getElementById('output');
             this.input = document.getElementById('input');

             this.button.addEventListener('click', () => this.addData());
             this.render();
         }

         addData() {
             const item = this.input.value;
             this.model.addData(item);
             this.render();
         }

         render() {
             const data = this.model.getData();
             this.output.innerHTML = data.join(', ');
         }
     }

     const Model = require('./model');
     const app = new ViewModel(new Model());
```

نتیجه گیری

سه مدل معماری نرم‌افزار *MV یعنی MVP ،MVC و MVVM را بررسی کردیم. هر یک از این معماری‌ها دارای مزایا و معایب خاص خود هستند و در شرایط مختلفی مناسب استفاده می‌شوند. معماری MVC با جداسازی واضح وظایف، تسهیل در نگهداری و توسعه‌ی مستقل اجزا و استفاده وسیع از کتابخانه‌ها و فریمورک‌های مختلف شناخته می‌شود. با این حال، coupling زیاد بین View و Controller و پیچیدگی در پروژه‌های بزرگ از معایب آن است. این معماری برای برنامه‌های وب با تعاملات پیچیده کاربر و نیاز به نگهداری و توسعه‌ی مداوم مناسب است. از سوی دیگر، معماری MVP با کاهش coupling بین اجزا، سبک‌تر بودن View و مدیریت بهتر منطق ارائه، به عنوان یک گزینه مطلوب شناخته می‌شود. اما نیاز به پیاده‌سازی رابط‌ها و پیچیدگی بیشتر در کدنویسی نسبت به MVC، از چالش‌های آن است. این معماری برای برنامه‌هایی که نیاز به جداسازی بیشتر بین اجزا و تعاملات کاربری پیچیده دارند، مناسب است. در نهایت، معماری MVVM با جداسازی کامل بین View و Model، کاهش coupling و امکان استفاده از binding برای ارتباط داده‌ها، مزایای بسیاری دارد. اما نیاز به یادگیری و درک دقیق از ViewModel و پیچیدگی در پیاده‌سازی در پروژه‌های بزرگ از معایب آن به شمار می‌رود. این معماری برای برنامه‌های تک صفحه‌ای (SPA) و برنامه‌هایی که نیاز به همگام‌سازی داده‌ها و تغییرات در زمان واقعی دارند، مناسب است. با توجه به نیازهای خاص هر پروژه، انتخاب معماری مناسب می‌تواند تاثیر زیادی بر کارایی و نگهداری سیستم داشته باشد. معماری MVVM به دلیل جداسازی وظایف و کاهش جفت شدن بین اجزا، برای پروژه‌هایی که نیاز به تعاملات کاربری پیچیده و همگام‌سازی داده‌ها دارند، بهترین گزینه به شمار می‌رود. اما برای پروژه‌های سنتی‌تر و با تعاملات پیچیده‌تر کاربری، معماری‌های MVC و MVP می‌توانند انتخاب‌های مناسبی باشند.