ChatGPT

Saf PHP ile OOP, MVC ve RESTful Web Servisleri

Bu belgedeki örnekler ve açıklamalar, saf (frameworksüz) PHP kullanılarak orta düzey uygulamalar geliştirmek için gereklidir. Aşağıdaki bölümler, PHP'de nesne yönelimli programlama, MVC mimarisi, yönlendirme (routing), RESTful web servisler, uygun klasör yapısı ve tüm bu kavramları içeren örnek bir proje yapısını detaylı olarak ele alır.

PHP'de Nesne Yönelimli Programlama (OOP)

Nesne yönelimli programlama (OOP), kodu gerçek dünyadaki nesnelere benzer şekilde sınıflar ve nesneler halinde düzenlemeyi sağlar [1]. PHP'de `class` anahtar kelimesi ile tanımlanan sınıflar, **özellikler** (properties) ve **metotlar** (methods) içerebilir. Her sınıf, aynı zamanda bir **nesne** (instance) oluşturmak için şablon görevi görür. Örneğin bir `User` sınıfı, kullanıcıyla ilgili verileri ve bu verilere ilişkin işlemleri bir arada tutabilir.

Sınıf Yapıları ve Görünürlük (Visibility)

Sınıf içinde tanımlanan özellikler ve metotlar, erişim düzeyine göre `public`, `protected` veya `private` olarak işaretlenebilir. **Public** üyeler sınıf dışından ve sınıf içinden serbestçe erişilebilirken, **protected** üyeler yalnızca tanımlandığı sınıf içinde ve bu sınıfı miras alan (child) sınıflarda erişilebilir. **Private** üyeler ise yalnızca tanımlandığı sınıf içinde kullanılabilir [2]. Örneğin:

```
class Person {
    public $name;            // Her yerden erişilebilir
    protected $age;          // Sınıf içinde ve alt sınıflarda erişilebilir
    private $password;       // Yalnızca bu sınıf içinde erişilebilir

    public function __construct($name, $age, $password) {
        $this->name = $name;
        $this->age = $age;
        $this->password = $password;
    }
}
```

Yukarıda `$name` genel (public), `$age` korumalı (protected), `$password` ise özel (private) olarak tanımlanmıştır. `__construct` yöntemi ise sınıf örneği oluşturulurken (nesne yaratılırken) çalışarak özellikleri başlatır. Bir nesnenin özelliklerine ve metotlarına erişim, okuma-yazma için `->` operatörüyle yapılır:

```
$person = new Person("Ahmet", 30, "secret");
echo $person->name;    // "Ahmet" (public olduğu için dışarıdan erişilebilir)
// echo $person->age;   // Hata: protected olduğu için erişilemez
```

Kalıtım (Inheritance)

PHP'de bir sınıf başka bir sınıfı genişletebilir (`extends`). Bu şekilde **miras (inheritance)** yoluyla alt sınıf, üst sınıfın (parent) tüm public ve protected özelliklerini/metotlarını devralır [3]. Örneğin, `Animal` sınıfını genişleten `Dog` sınıfı:

```
class Animal {
    protected $type;
    public function __construct($type) {
        $this->type = $type;
    }
    public function getType() {
        return $this->type;
    }
}

class Dog extends Animal {
    public function bark() {
        echo "Hav! Ben bir " . $this->type . " köpeğiyim.";
    }
}

$dog = new Dog("memeli");
$dog->bark(); // Çıktı: Hav! Ben bir memeli köpeğiyim.
```

`Dog` sınıfı, `Animal` sınıfını kalıtım yoluyla genişlettiği için `getType()` metoduna ve `$type` özelliğine erişebilir. PHP dökümantasyonuna göre, alt sınıf üst sınıfın public ve protected tüm üye öğelerini miras alır [3].

Arayüz (Interface)

Interface (arayüz) kavramı, bir sınıfın hangi metotları mutlaka uygulaması gerektiğini tanımlar. `interface` anahtar kelimesi ile tanımlanan bir arayüz, içinde sadece public metot imzaları bulundurur. Bir sınıf bu arayüzü `implements` anahtar kelimesi ile benimsediğinde, arayüzde tanımlı tüm metotları kendisi gerçekleştirmek (implement etmek) zorundadır [4]. Örneğin:

```
interface LoggerInterface {
    public function log($message);
}

class FileLogger implements LoggerInterface {
    public function log($message) {
        file_put_contents("app.log", $message.PHP_EOL, FILE_APPEND);
    }
}
```

Bu örnekte `LoggerInterface` bir `log` metodu sözü söyler. `FileLogger` sınıfı ise `implements LoggerInterface` diyerek bu metodu uygulamakta ve kendi `log` işlevini gerçekleştirmektedir.

Trait

Trait kavramı, PHP'de tekil kalıtımın sınırlamalarını aşmak için kullanılan bir kod paylaşım mekanizmasıdır. Trait içi fonksiyonlar, bir sınıf `extends` veya `implements` olmaksızın `use` anahtar kelimesi ile sınıfa eklenebilir. Örneğin:

```
trait TimestampTrait {
    public function getTimestamp() {
        return date("Y-m-d H:i:s");
    }
}

class Record {
    use TimestampTrait;
}

$rec = new Record();
echo $rec->getTimestamp();  // Trait içindeki metodu sınıfa ekler
```

Yukarıda, `TimestampTrait` içinde tanımlanan `getTimestamp` metodu, `Record` sınıfına `use` edilerek eklenmiştir [5] . Bu sayede aynı trait farklı sınıflarda ortak davranış olarak kullanılabilir.

Örnek

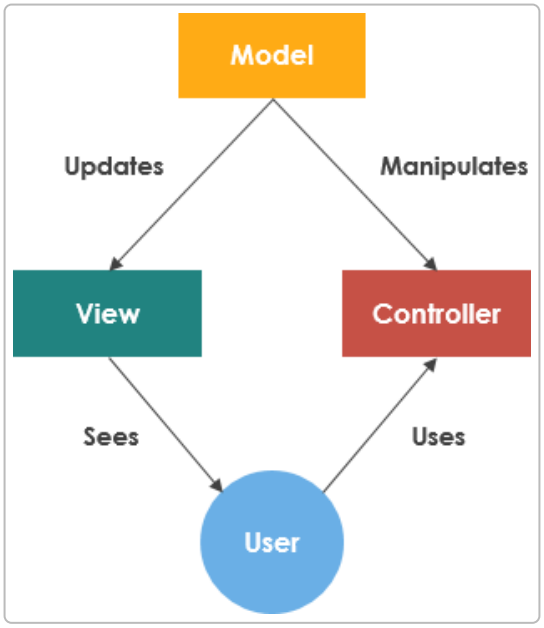
Aşağıdaki örnek, kalıtım, interface ve trait kullanımını bir arada göstermektedir. `Animal` sınıfını `Dog` genişletiyor, `PetInterface` arayüzünü `Cat` sınıfı uyguluyor, ayrıca `Friendly` trait'i sınıflara ek fonksiyonellik katıyor:

```
<?php
class Animal {
    public $name;
    protected $age;
    public function __construct($name, $age) {
        $this->name = $name;
        $this->age = $age;
    }
    public function getInfo() {
        return "{$this->name}, {$this->age} yaşında";
    }
}

interface PetInterface {
    public function play();
}

trait Friendly {
    public function greet() {
        echo "Merhaba! Ben arkadaş canlısıyım.";
    }
```

```php
}

class Dog extends Animal {
    public function bark() {
        echo "Hav hav!";
    }
}

class Cat extends Animal implements PetInterface {
    use Friendly;
    public function play() {
        echo "{$this->name} oyun oynuyor.";
    }
}

// Kullanım:
$dog = new Dog("Karabas", 5);
echo $dog->getInfo();   // Kalıtımla alınan metot
$dog->bark();           // Kendi metodu

$cat = new Cat("Minnak", 2);
echo $cat->getInfo();
$cat->greet();          // Trait ile eklenen metot
$cat->play();           // Interface gereği implement edilen metot
?>
```

Yukarıdaki kodda, `Dog` sınıfı `Animal` sınıfından miras alırken, `Cat` sınıfı bir arayüzü uygular ve trait'i kullanır. Bu sayede **kapsülleme** (özelliklerin görünürlüğü), **miras** (inheritance), **arayüz** ve **trait** gibi OOP kavramları birlikte gösterilmiştir.

MVC (Model-View-Controller) Mimarisi



MVC (Model-View-Controller) mimarisi, uygulamanın farklı sorumlulukları ayrıştırarak daha kolay yönetilebilir hâle getiren bir tasarım desenidir [6]. MVC'de **Model** veri ve iş katmanını yönetir, **View** verinin kullanıcıya sunumunu sağlar, **Controller** ise kullanıcıdan gelen istekleri alır ve Model-View arasında köprü görevi görür [7]. Aşağıdaki görselde bu bileşenler ve aralarındaki etkileşim şeması görülmektedir:

- **Model:** Veritabanı işlemleri ve iş mantığı burada yer alır. Kullanıcı girdileri Controller üzerinden Model'e iletilir ve Model, gerekli veriyi sağlar [7].
- **View (Görünüm):** Model'den gelen veriyi HTML/CSS/JS formatında kullanıcıya gösterir. Tekrar kullanılabilir şablonlardan oluşabilir.
- **Controller (Denetleyici):** İstemciden gelen istekleri (`GET`, `POST` gibi) alır, doğrular ve hangi Model'in kullanılacağını belirler. Model'den gelen veriyi uygun View'a göndererek kullanıcıya sonuç döner [7].

Bu ayrım sayesinde geliştiriciler, örneğin kullanıcı arayüzünü (view) veya veri katmanını birbirinden bağımsız olarak geliştirebilir. Visual-Paradigm gibi kaynaklar, MVC'nin iş mantığını (business logic), kullanıcı arayüzünü ve giriş mantığını birbirinden ayırarak düşük bağlılık (low coupling) ve yüksek yeniden kullanım (reusability) sağladığını vurgulamaktadır [6] [7].

PHP ile MVC Kurulum Adımları

Saf PHP ile basit bir MVC uygulaması kurmak için izlenebilecek genel adımlar şunlardır:

1. **Proje Klasör Yapısını Oluşturma:** `public/` dizinini web kök dizini yapın. `src/` veya `app/` gibi bir dizin içinde `Controllers/`, `Models/`, `Views/` ve gerekirse `Routes/` gibi klasörler oluşturun. Yapılandırma dosyalarını `config/` dizinine koyun. Örneğin Mahesh Samudra'nın önerdiği yapıda `public/`, `src/Controllers`, `src/Models`, `src/Routes`, `src/Views` gibi klasörler bulunur [8].
2. **Front Controller (index.php):** `public/index.php` dosyası tüm gelen isteklerin yönlendirileceği tek giriş noktasıdır. Apache kullanıyorsanız `.htaccess` dosyasıyla bütün istekleri bu dosyaya yönlendirirsiniz. Örneğin:

   ```
   RewriteEngine On
   RewriteCond %{REQUEST_FILENAME} !-f
   RewriteRule ^(.*)$ index.php [L]
   ```

   Bu kurallarda, mevcut bir dosya/dizin değilse tüm URL'ler `index.php` dosyasına yönlendirilir [9].
3. **Routing (Yönlendirme):** `index.php` içinde `$_SERVER['REQUEST_URI']` veya benzeri bir yöntemle gelen URL elde edilir. Daha sonra bu yolu uygun controller'a yönlendiren bir **router** mekanizması oluşturulur veya basitçe bir `switch` / `if` dizisi kullanılır. Örneğin:

   ```php
   <?php
   $request = $_SERVER['REQUEST_URI'];
   switch ($request) {
       case '/':
           (new HomeController())->index();
           break;
       case '/users':
           (new UserController())->list();
   ```

```
        break;
    default:
        http_response_code(404);
}
```

Bu örnekte `/` isteği `HomeController`'ın `index` metoduna, `/users` isteği `UserController`'ın `list` metoduna yönlendirilir. Yazılan blog örneklerinde de benzer şekilde istek URI alınarak doğru dosya çağrılır veya 404 sayfası gösterilir [10].

4. **Controller ve Model Etkileşimi:** Controller içinde, gerekli veriyi almak için ilgili Model sınıfları çağrılır. Model sınıfının veritabanı sorgulama gibi işlemlerinin sonuçları Controller'a döner.
5. **View Render Etme:** Controller, aldığı verileri bir view dosyasına aktarır. View dosyası PHP/HTML karışık bir şablon olabilir. Örneğin, `$users` değişkenini listelemek için bir `list.php` şablonu yüklenebilir:

```
// Controller içinde
$users = User::getAll();
include __DIR__ . '/../Views/users/list.php';
```

Bu sayede, veriler HTML içinde döngülerle işlenip son kullanıcıya gösterilir.

Bu adımlar sayesinde saf PHP ile MVC tasarım desenini uygulayan bir yapı kurulabilir. Yönlendirme adımında `.htaccess` ile tüm isteklerin tek bir giriş noktası üzerinden kontrol edilmesi, MVC yapısını basitçe yönetilebilir kılar [9] [10].

Yönlendirme (Routing)

.htaccess ile URL Yönlendirme

PHP ile modern web uygulamaları geliştirirken, kullanıcı dostu ve temiz URL'ler kullanmak önemlidir. Apache kullanılıyorsa `.htaccess` dosyası aracılığıyla URL yeniden yazma (URL rewriting) yapılabilir. Aşağıdaki örnek `.htaccess` kuralları, gelen isteklerde fiziksel olarak var olmayan tüm yolları `index.php` dosyasına yönlendirir [9]:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php [L]
```

- `RewriteEngine On`: Apache mod_rewrite'ı etkinleştirir.
- `RewriteCond %{REQUEST_FILENAME} !-f`: Eğer talep edilen dosya fiziksel olarak mevcut değilse (örneğin `/about` gibi bir dosya yoksa),
- `RewriteRule ^(.*)$ index.php [L]`: bu isteği `index.php` dosyasına yönlendirir [9].

Bu sayede örneğin kullanıcı `https://site.com/users` adresine girdiğinde gerçek bir `users.php` dosyası olmasa bile tüm trafik `index.php` üzerinden işlenebilir. Benzer şekilde Nginx kullanılıyorsa, server konfigürasyonunda `try_files $uri $uri/ /index.php?$args;` gibi bir kural kullanmak gereklidir. Ancak saf PHP odaklı projelerde genellikle Apache ve `.htaccess` yeterli olacaktır.

PHP ile Routing Mekanizması

Yönlendirme düzeni (routing) genellikle index.php içinde tanımlanır. PHP tarafında `$_SERVER['REQUEST_URI']` kullanılarak kullanıcı isteği alınır ve bu URI'ye göre uygun kod bloğu çalıştırılır. Örneğin basit bir `switch` yapısı ile farklı URL yollarını yakalayıp ilgili denetleyiciyi çağırmak mümkündür:

```
<?php
$request = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
switch ($request) {
    case '/':
        (new HomeController())->index();
        break;
    case '/tasks':
        (new TaskController())->list();
        break;
    case '/tasks/create':
        (new TaskController())->create();
        break;
    default:
        http_response_code(404);
        echo "Sayfa bulunamadı.";
}
```

Bu örnekte `/tasks` yoluna gelen istek `TaskController`'ın `list` metoduna, `/tasks/create` ise `create` metoduna yönlendirilmektedir. Abel Mbula'nın örneğinde de (örn. bir blog yazısında) `switch` kullanılarak gelen isteklerle farklı sayfa dosyalarının çağrıldığı görülmektedir [10]. Gelişmiş projelerde bu iş için bir `Router` sınıfı veya rota tanımlama dosyası (`routes/web.php` vb.) kullanılabilir. Ancak temel bir uygulamada basit kontrol yapıları veya bir dizi (array) haritasıyla da yönlendirme sağlanabilir. Örneğin:

```
$routes = [
    '/' => ['HomeController', 'index'],
    '/users' => ['UserController', 'list']
];

if (isset($routes[$request])) {
    [$controller, $method] = $routes[$request];
    (new $controller())->$method();
} else {
    http_response_code(404);
    echo "Sayfa bulunamadı.";
}
```

Yukarıdaki gibi bir yaklaşım da seçilebilir. Önemli olan, her istek URI'sını uygun sınıf ve metodla eşleştirip doğru kodu çalıştırmaktır. Bu sayede URL'ye bağlı olarak uygulamanın farklı kısımları devreye girer.

Web Servis Yapısı (RESTful Servisler)

REST (Representational State Transfer), web servisleri için hafif ve yaygın bir mimari stildir. RESTful servisler, istemci ve sunucu arasında **HTTP protokolü** üzerinden kaynakları paylaştıran bir yapıdır [11]. Yani SOAP veya RPC gibi karmaşık protokoller yerine, basitçe HTTP istekleri (GET, POST, vb.) kullanılır. Bu yüzden bir RESTful API, web tarayıcılarında çalışan ön yüz uygulamalarıyla arka uç arasında veri aktarımı için idealdir.

RESTful web servislerinde her kaynak (örneğin "kullanıcılar", "görevler" vb.) bir URL (uç nokta – endpoint) ile tanımlanır. Örneğin `/api/users` URL'si kullanıcı listesine, `/api/users/1` tek bir kullanıcıya erişimi temsil edebilir. İstemci bu kaynaklar üzerinde belirli HTTP yöntemlerini (metodlarını) kullanarak işlem yapar. En yaygın olarak kullanılan HTTP yöntemleri şunlardır [12]:

- **GET:** Sunucudan veri **okumak** için kullanılır. Örneğin `/api/tasks` tüm görevleri getirir.
- **POST:** Sunucuya **yeni veri eklemek (oluşturmak)** için kullanılır. Örneğin `/api/tasks` adresine yapılan POST, yeni bir görev oluşturabilir.
- **PUT/PATCH:** Mevcut bir kaynağı **güncellemek** için kullanılır (genellikle PUT tüm kaynak, PATCH kısmı güncelleme içindir). Örneğin `/api/tasks/5` adresine PUT isteği, 5 numaralı görevi güncelleyebilir.
- **DELETE:** Bir kaynağı **silmek** için kullanılır. Örneğin `/api/tasks/5` adresine DELETE isteği, 5 numaralı görevi siler.

Bu yöntemlerin RESTful mimarideki karşılıkları, geleneksel veritabanı işlemlerine (CRUD: Create, Read, Update, Delete) paraleldir [12]. Ayrıca veriler genellikle **JSON** formatında iletilir, çünkü JSON günümüzde en yaygın veri formatıdır [13]. PHP tarafında bir RESTful API endpoint'i şöyle işleyebilir:

```php
<?php
header("Content-Type: application/json; charset=UTF-8");
switch ($_SERVER['REQUEST_METHOD']) {
    case 'GET':
        // Örneğin veri tabanından liste çek veya sabit veri
        $data = ['id' => 1, 'name' => 'Örnek'];
        echo json_encode($data);
        break;
    case 'POST':
        // İstek gövdesinden gelen JSON veriyi oku
        $input = json_decode(file_get_contents('php://input'), true);
        // Yeni kaynak oluşturma işlemi...
        echo json_encode(['status' => 'created', 'data' => $input]);
        break;
    case 'PUT':
        // Belirli kaynağı güncelle...
        echo json_encode(['status' => 'updated']);
        break;
    case 'DELETE':
        // Belirli kaynağı sil...
        echo json_encode(['status' => 'deleted']);
        break;
    default:
        http_response_code(405);
```

```
        echo json_encode(['error' => 'Metod desteklenmiyor']);
}
?>
```

Bu örnekte `$_SERVER['REQUEST_METHOD']` kontrol edilerek gelen HTTP metoduna göre farklı işlemler yapılmaktadır. GET metodunda örnek bir JSON nesnesi döndürülüyor, POST metodunda ise istek gövdesindeki JSON verisi alınıp yanıt olarak dönülüyor. Bu şekilde, her uç nokta (endpoint) için farklı bir PHP kod bloğu tanımlanarak basit bir RESTful servis oluşturulabilir.

Uç Noktalar (Endpoints)

RESTful servislerde her işlev (örneğin kullanıcı ekleme, listeleme) ayrı bir endpoint ile sunulur. Örneğin:

- `GET /api/users` – Tüm kullanıcıları listeler.
- `GET /api/users/{id}` – Belirli bir kullanıcıyı getirir.
- `POST /api/users` – Yeni kullanıcı oluşturur (istek gövdesinde JSON verisi ile).
- `PUT /api/users/{id}` – Belirli kullanıcıyı günceller.
- `DELETE /api/users/{id}` – Belirli kullanıcıyı siler.

Bu şeklide her kaynak tipi için standart URI ve HTTP metodları kullanılır. Bir uç noktadan (endpoint'ten) alınan cevap genellikle `200 OK`, `201 Created`, `404 Not Found` gibi uygun HTTP durum kodlarıyla birlikte JSON formatında iletilir.

Önerilen Klasör Yapısı

Profesyonel projelerde kodu düzenlemek için belirli bir klasör yapısı izlemek faydalıdır. Aşağıdaki yapı yaygın bir örnektir. Mahesh Samudra'nın örnek yapısında `public/`, `src/Controllers/`, `src/Models/`, `src/Routes/`, `src/Views/` gibi dizinler yer almaktadır [8]. Biz bu yapıyı biraz daha açalım:

```
proje-kök-dizini/
├─ public/               # Web sunucusunun doküman dizini
│   ├─ index.php         # Tüm isteklerin yönlendirildiği front controller
│   ├─ .htaccess         # URL yeniden yazma kuralları
│   └─ assets/           # CSS, JS, görseller gibi statik dosyalar
│
├─ src/                  # Uygulama kaynak kodları
│   ├─ Controllers/      # Controller sınıfları (istekleri işler)
│   ├─ Models/           # Model sınıfları (veri ve iş mantığı)
│   ├─ Views/            # View dosyaları (HTML şablonlar)
│   └─ Routes/           # (Opsiyonel) Özel yönlendirme tanımları
│
├─ config/               # Konfigürasyon dosyaları (veritabanı ayarları,
sabitler vb.)
│
├─ routes/               # (Bazı projelerde) rota tanımlarının tutulduğu
dizin
```

```
│
└ vendor/                # (composer kullanılıyorsa) bağımlılıklar
```

- **public/**: Uygulamanın kök dizini olarak, web sunucusunun eriştiği yerdir. `index.php` front controller olarak burada yer alır. CSS, JS, resim gibi statik dosyalar genelde `assets/` veya `public/css`, `public/js` gibi alt klasörlerde tutulur.
- **src/Controllers/**: Her URL isteğini karşılayacak denetleyici sınıfları burada bulunur. Örneğin `UserController.php`, `TaskController.php`. Her sınıf genellikle bir sayfa ya da API endpoint'ini işler.
- **src/Models/**: Veritabanı ile etkileşen veya uygulama verisini temsil eden sınıflar burada yer alır. Örneğin `User.php`, `Task.php` gibi sınıflar, veritabanı sorgularını yapar veya iş kurallarını içerir.
- **src/Views/**: Kullanıcıya gösterilecek HTML şablonları (view dosyaları) bu klasörde tutulur. Örneğin `users/list.php`, `tasks/form.php` gibi dosyalar PHP ile gömülü HTML içerebilir.
- **src/Routes/**: Bazı yapılarda tüm URL yönlendirmeleri bu dizinde tanımlı bir dosyada yapılır. Örneğin `routes/web.php` dosyasında `/tasks` yolunun hangi controller'a gideceği belirtilebilir.
- **config/**: Veritabanı bağlantı ayarları, uygulama sabitleri (örneğin `config/database.php`, `config/app.php`) gibi yapılandırma dosyaları burada bulunur.
- **vendor/**: Composer kullanılıyorsa, tüm kütüphane bağımlılıkları bu dizin altında tutulur. (Saf PHP'de composer kullanımı isteğe bağlıdır.)

Bu önerilen yapı, kodu temiz ve modüler tutar. Örneğin `Controllers` altındaki sınıflar sadece yönlendirme ve akış kontrolüyle ilgilenirken, veritabanı kodları `Models` içinde toplanır ve HTML içeriği `Views` altında izole edilir [8]. Böylece her geliştirici ihtiyacı olan katmanda çalışarak projeyi sürdürebilir.

Örnek Mini Proje Yapısı: Görev Yönetimi Uygulaması

Bu bölümde, yukarıdaki kavramları bir araya getiren örnek bir mini proje yapısı göstereceğiz. Örneğimizde basit bir **Görev Yönetim (ToDo)** uygulaması düşünelim. Kullanıcılar görev ekleyip listeleyebiliyor ve silebiliyor olacak. Aşağıda proje dosya yapısının genel bir görünümü bulunmaktadır:

```
todo-app/
├ public/
│   ├ index.php          # Tüm isteklerin yönlendirildiği ana dosya
│   ├ .htaccess          # Yönlendirme kuralları
│   └ assets/
│       ├ css/
│       └ js/
│
├ src/
│   ├ Controllers/
│   │   └ TaskController.php   # Görev işlemlerini yöneten controller
│   ├ Models/
│   │   └ Task.php             # Task modeli (veritabanı işlemleri)
│   └ Views/
│       └ tasks/
│           ├ list.php        # Görev listesi görünümü (HTML şablon)
```

```
│           └─ form.php        # Görev ekleme formu (HTML şablon)
│
├─ routes/
│   └─ web.php            # Uygulama rotalarının tanımlandığı dosya
│
└─ config/
    └─ database.php       # Veritabanı bağlantı ayarları
```

Yapı Açıklamaları:
- `public/index.php`: Front controller olarak tüm istekler önce bu dosyaya gelir. Örneğin `.htaccess` ile `/tasks` isteği `index.php`'ye yönlendirildiyse, burada gerekli route (yönlendirme) kodu çalışır.
- `routes/web.php`: (İsteğe bağlı) Bütün URL rotalarını tanımlamak için basit bir dizin. Örneğin `'/tasks' => [TaskController::class, 'list']` gibi eşlemeler yapılabilir.
- `src/Controllers/TaskController.php`: Görevlerle ilgili istekleri işleyen sınıftır. İçinde örneğin `list()`, `create()`, `delete()` gibi metodlar bulunur. Bu metodlarda Model çağrıları yapılır ve sonuçlar View'a aktarılır.
- `src/Models/Task.php`: Görev nesnesini temsil eden model sınıfı. Bu sınıfta görev verilerini veritabanından çekmek (`getAll`), yeni görev eklemek (`create`), silmek (`delete`) gibi statik metotlar olabilir. Örneğin PDO ile veritabanı sorguları burada yazılır.
- `src/Views/tasks/list.php`: Görev listesini HTML olarak gösteren şablon. `$tasks` dizisi döngü ile işlenip ekrana yazdırılır.
- `src/Views/tasks/form.php`: Yeni görev ekleme formunu içeren HTML şablonu. Form gönderildiğinde `TaskController::create` metodu tetiklenir.

Örnek Kod Akışı:
1. **Yönlendirme:** Kullanıcı tarayıcısında `/tasks` yoluna girdiğinde, Apache `.htaccess` kuralları sayesinde istek `public/index.php`'ye gelir. Bu dosyada, `parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH)` ile `/tasks` elde edilir.
2. **Controller Çağrısı:** Routing mekanizmasına göre `/tasks` isteği `TaskController`'ın `list` metoduna yönlendirilir. Yani `new TaskController()->list()` çalışır.
3. **Model Etkileşimi:** `TaskController::list()` içinde `Task::getAll()` gibi bir model metodu çağrılarak tüm görevler veritabanından çekilir.
4. **View Yükleme:** Controller, aldığı `$tasks` verisini `include __DIR__ . '/../Views/tasks/list.php';` gibi bir kodla ilgili View dosyasına aktarır. List şablonu çalışırken `$tasks` dizisini HTML içine basar.
5. **Form İşlemi (POST):** Eğer kullanıcı görev ekleme formunu doldurup gönderirse, yine `index.php` gelen POST isteği yakalar ve `/tasks/create` rotası `TaskController::create()` metodunu çalıştırır. Bu metod `Task::create($_POST)` diyerek modeli kullanır, veritabanına yeni kayıt ekler ve tekrar liste sayfasına yönlendirir.

Bu örnekle, tüm katmanların birlikte nasıl çalıştığı gösterilmiş olur. MVC mimarisinde Controller, Model ve View bileşenleri kendi görevlerinde uzmanlaşırken, routing ise kullanıcı isteklerini doğru Controller'a aktarır. RESTful bir API olarak yapılsaydı, benzer Controller metodları JSON dönecek şekilde düzenlenebilirdi. Buradaki örnek, bir web arayüzü üzerinden çalışan geleneksel MVC akışını göstermektedir.

Sonuç: Yukarıdaki yapı ve kod akışı, saf PHP kullanarak orta ölçekli bir web uygulamasının nasıl düzenlenebileceğini göstermektedir. Sınıflar (`class`), kalıtım (`extends`), interface ve trait gibi OOP kavramlarıyla kod organize edilir. MVC deseni ile iş, görünüm ve veri katmanları ayrıştırılır. `.htaccess`

ve PHP tarafında routing mekanizmasıyla URL'ler yönetilir. RESTful prensiplerle de API uç noktaları oluşturulup HTTP metodlarıyla veri alışverişi yapılabilir. Bu kavramların entegrasyonu, orta seviye PHP geliştiricilere ölçeklenebilir ve bakımı kolay uygulamalar geliştirmede sağlam bir temel sunar.

Kaynakça: Açıklamalarda PHP'nin resmi belgeleri ve güncel blog yazılarından yararlanılmıştır [2] [4] [9] [10] [14] [6]. Bu kaynaklar PHP OOP, MVC ve REST mimarileri hakkında teknik detaylar içermektedir.

---

[1] [2] [4] [5] The definitive guide to object-oriented programming in PHP - Honeybadger Developer Blog
https://www.honeybadger.io/blog/in-depth-guide-to-object-oriented-programming-in-php/

[3] PHP: Object Inheritance - Manual
https://www.php.net/manual/en/language.oop5.inheritance.php

[6] [7] What is Model-View and Control?
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-model-view-control-mvc/

[8] Creating a Simple PHP MVC Framework from Scratch | by Mahesh Samudra | Medium
https://maheshsamudra.medium.com/creating-a-simple-php-mvc-framework-from-scratch-7158f12340a0

[9] [10] Building a Routing System in PHP from Scratch
https://www.abelmbula.com/blog/php-routing/

[11] REST ve RESTful Web Servis Kavramı | by Deniz İrgin | Deniz İrgin Blog
https://denizirgin.com/rest-ve-restful-web-servis-kavram%C4%B1-30bc4400b9e0

[12] [13] [14] REST API using PHP (Basic GET, POST, PUT & DELETE) | by Dzikri Robbi, S.Tp, M.Kom | Medium
https://dzkrrbb.medium.com/rest-api-with-php-get-post-put-delete-8365fe092618