

Pertemuan 11: Namespaces dan Autoloading

Tujuan Pembelajaran

Setelah mengikuti pertemuan ini, mahasiswa diharapkan dapat:

1. Memahami konsep Namespaces dalam PHP
2. Menggunakan keyword `namespace` dan `use` dengan benar
3. Mengatasi name collision dengan namespaces
4. Memahami Fully Qualified Names (FQN)
5. Mengimplementasikan autoloading dengan `spl_autoload_register()`
6. Menggunakan PSR-4 autoloading standard
7. Mengelola dependencies dengan Composer autoloader

Konsep Namespaces

Definisi Namespaces

Namespaces adalah cara untuk mengorganisir class, interface, function, dan constant dalam PHP. Namespace memungkinkan penggunaan nama yang sama di konteks yang berbeda tanpa konflik.

Masalah yang Dipecahkan Namespaces

1. **Name Collision** - Konflik nama class yang sama
2. **Code Organization** - Mengorganisir code dalam struktur hierarkis
3. **Third-party Integration** - Menghindari konflik dengan library eksternal
4. **Readability** - Memberikan konteks yang jelas untuk class names

Analogi Namespace

Seperti sistem file di komputer:

- **Directory** = Namespace
- **File** = Class/Interface/Function
- **Full Path** = Fully Qualified Name (FQN)

```
/App/Models/User.php      → App\Models\User  
/App/Controllers/User.php → App\Controllers\UserController  
/Vendor/Logger/File.php   → Vendor\Logger\FileLogger
```

Basic Namespace Usage

Mendeklarasikan Namespace

```
<?php  
// File: src/Models/User.php
```

```
namespace App\Models;

class User {
    public function getName() {
        return "User from App\Models";
    }
}
```

Menggunakan Class dari Namespace

```
<?php
// File: index.php

// Option 1: Fully Qualified Name
$user = new \App\Models\User();

// Option 2: Use statement
use App\Models\User;
$user = new User();

// Option 3: Use with alias
use App\Models\User as AppUser;
$user = new AppUser();
```

Multiple Classes dalam Satu File

```
<?php
namespace App\Models;

class User {
    // Implementation
}

class Post {
    // Implementation
}

interface Timestampable {
    // Interface definition
}
```

Namespace Hierarchy

Nested Namespaces

```
<?php
namespace App\Http\Controllers;

class UserController {
    // Controller implementation
}
```

```
<?php
namespace App\Http\Middleware;

class AuthMiddleware {
    // Middleware implementation
}
```

Global Namespace

```
<?php
// Global namespace (no namespace declaration)

class DateTime {
    // This conflicts with built-in DateTime
}

// To access built-in DateTime from namespaced code:
namespace App\Models;

class Event {
    public function getDate() {
        return new \DateTime(); // Leading backslash for global namespace
    }
}
```

Use Statements

Basic Use

```
<?php
namespace App\Controllers;

use App\Models\User;
use App\Models\Post;
use App\Services\EmailService;

class UserController {
    public function index() {
        $user = new User();           // App\Models\User
```

```
$post = new Post();           // App\Models\Post
$email = new EmailService(); // App\Services\EmailService
}
}
```

Use dengan Alias

```
<?php
namespace App\Controllers;

use App\Models\User as UserModel;
use App\Http\Controllers\Admin\User as AdminUserController;
use DateTime as PhpDateTime;

class UserController {
    public function create() {
        $user = new UserModel();           // App\Models\User
        $admin = new AdminUserController(); // App\Http\Controllers\Admin\User
        $date = new PhpDateTime();         // DateTime
    }
}
```

Group Use Statements (PHP 7+)

```
<?php
namespace App\Controllers;

// Instead of:
// use App\Models\User;
// use App\Models\Post;
// use App\Models\Comment;

// Use group import:
use App\Models\{
    User,
    Post,
    Comment
};

// With aliases:
use App\Services\{
    EmailService,
    LoggerService as Logger,
    CacheService
};
```

Autoloading

Manual Loading (Before Autoload)

```
<?php
// Manual include/require
require_once 'src/Models/User.php';
require_once 'src/Models/Post.php';
require_once 'src/Controllers/UserController.php';

use App\Models\User;
use App\Controllers\UserController;

$user = new User();
$controller = new UserController();
```

Basic Autoloading

```
<?php
// Basic autoloader
function autoload($className) {
    // Convert namespace to file path
    $file = str_replace('\\', '/', $className) . '.php';
    $fullPath = __DIR__ . '/src/' . $file;

    if (file_exists($fullPath)) {
        require_once $fullPath;
    }
}

// Register autoloader
spl_autoload_register('autoload');

// Now classes are loaded automatically
use App\Models\User;
$user = new User(); // Automatically loads src/App/Models/User.php
```

PSR-4 Autoloading

```
<?php
class Psr4Autoloader {
    private $prefixes = [];

    public function addNamespace($prefix, $baseDir) {
        // Normalize namespace prefix
        $prefix = trim($prefix, '\\') . '\\';

        // Normalize base directory
        $baseDir = rtrim($baseDir, DIRECTORY_SEPARATOR) . '/';
```

```
// Initialize prefix array if needed
if (!isset($this->prefixes[$prefix])) {
    $this->prefixes[$prefix] = [];
}

// Add base directory to prefix
$this->prefixes[$prefix][] = $baseDir;
}

public function loadClass($class) {
    // Current namespace prefix
    $prefix = $class;

    // Loop through prefixes to find a match
    while (false !== $pos = strpos($prefix, '\\\\')) {
        $prefix = substr($class, 0, $pos + 1);
        $relativeClass = substr($class, $pos + 1);

        // Try to load mapped file
        $mappedFile = $this->loadMappedFile($prefix, $relativeClass);
        if ($mappedFile) {
            return $mappedFile;
        }

        // Remove trailing separator for next iteration
        $prefix = rtrim($prefix, '\\\\');
    }

    return false;
}

protected function loadMappedFile($prefix, $relativeClass) {
    if (!isset($this->prefixes[$prefix])) {
        return false;
    }

    foreach ($this->prefixes[$prefix] as $baseDir) {
        $file = $baseDir . str_replace('\\\\', '/', $relativeClass) .
'.php';

        if ($this->requireFile($file)) {
            return $file;
        }
    }

    return false;
}

protected function requireFile($file) {
    if (file_exists($file)) {
        require $file;
        return true;
    }
    return false;
}
```

```
    }

}

// Usage
$loader = new Psr4Autoloader();
$loader->addNamespace('App\\', __DIR__ . '/src/App/');
$loader->addNamespace('Vendor\\Logger\\', __DIR__ .
'/vendor/logger/src/');

spl_autoload_register([$loader, 'loadClass']);
```

Composer Autoloading

composer.json Configuration

```
{
  "name": "my-project/php-oop-course",
  "description": "PHP OOP Course with Namespaces",
  "autoload": {
    "psr-4": {
      "App\\": "src/App/",
      "Tests\\": "tests/"
    },
    "files": [
      "src/helpers.php"
    ]
  },
  "autoload-dev": {
    "psr-4": {
      "Tests\\": "tests/"
    }
  },
  "require": {
    "php": ">=8.0"
  },
  "require-dev": {
    "phpunit/phpunit": "^9.0"
  }
}
```

Using Composer Autoloader

```
<?php
// File: index.php
require_once 'vendor/autoload.php';

use App\Models\User;
use App\Controllers\UserController;
use App\Services\EmailService;
```

```
// Classes are automatically loaded
$user = new User();
$controller = new UserController();
$emailService = new EmailService();
```

Directory Structure Best Practices

PSR-4 Compatible Structure

```
project/
├── composer.json
└── vendor/
    └── autoload.php
src/
└── App/
    ├── Models/
    │   ├── User.php      → App\Models\User
    │   ├── Post.php     → App\Models\Post
    │   └── Comment.php   → App\Models\Comment
    ├── Controllers/
    │   ├── UserController.php → App\Controllers\UserController
    │   ├── PostController.php → App\Controllers\PostController
    │   └── Admin/
    │       └── UserController.php →
    │           App\Controllers\Admin\UserController
    ├── Services/
    │   ├── EmailService.php → App\Services>EmailService
    │   └── LoggerService.php → App\Services\LoggerService
    └── Traits/
        ├── Timestampable.php → App\Traits\Timestampable
        └── Cacheable.php     → App\Traits\Cacheable
tests/
└── Unit/
    ├── UserTest.php      → Tests\Unit\UserTest
    └── PostTest.php      → Tests\Unit\PostTest
```

Advanced Namespace Features

Functions dalam Namespace

```
<?php
namespace App\Helpers;

function formatCurrency($amount) {
    return '$' . number_format($amount, 2);
}

function slugify($text) {
```

```
        return strtolower(preg_replace('/[^A-Za-z0-9-]+/', '-', $text));  
    }  
  
    // Usage:  
    use function App\Helpers\formatCurrency;  
    use function App\Helpers\slugify;  
  
    echo formatCurrency(1234.56); // $1,234.56  
    echo slugify('Hello World'); // hello-world  
  
    // Or with namespace qualifier:  
    echo App\Helpers\formatCurrency(1234.56);
```

Constants dalam Namespace

```
<?php  
namespace App\Config;  
  
const DB_HOST = 'localhost';  
const DB_PORT = 3306;  
const API_VERSION = 'v1';  
  
// Usage:  
use const App\Config\DB_HOST;  
use const App\Config\DB_PORT;  
  
echo DB_HOST; // localhost  
echo DB_PORT; // 3306  
  
// Or with namespace qualifier:  
echo App\Config\API_VERSION; // v1
```

Magic Constants

```
<?php  
namespace App\Models;  
  
class User {  
    public function getInfo() {  
        return [  
            'namespace' => __NAMESPACE__, // App\Models  
            'class' => __CLASS__, // App\Models\User  
            'file' => __FILE__, // Full file path  
            'line' => __LINE__, // Current line number  
            'method' => __METHOD__, // App\Models\User::getInfo  
            'function' => __FUNCTION__ // getInfo  
        ];  
    }  
}
```

Practical Examples

1. MVC Structure dengan Namespaces

```
<?php
// File: src/App/Models/User.php
namespace App\Models;

use App\Traits\Timestampable;

class User {
    use Timestampable;

    private $id;
    private $name;
    private $email;

    public function __construct($name, $email) {
        $this->name = $name;
        $this->email = $email;
        $this->touch();
    }

    public function getName() { return $this->name; }
    public function getEmail() { return $this->email; }
}
```

```
<?php
// File: src/App/Controllers/UserController.php
namespace App\Controllers;

use App\Models\User;
use App\Services\EmailService;
use App\Services\LoggerService;

class UserController {
    private $emailService;
    private $logger;

    public function __construct(EmailService $emailService, LoggerService $logger) {
        $this->emailService = $emailService;
        $this->logger = $logger;
    }

    public function create($name, $email) {
        $user = new User($name, $email);
    }
}
```

```
        $this->logger->info("User created: {$name}");
        $this->emailService->sendWelcomeEmail($user);

        return $user;
    }
}
```

2. Service Layer Pattern

```
<?php
// File: src/App/Services/EmailService.php
namespace App\Services;

use App\Models\User;
use App\Services\Contracts\EmailServiceInterface;

class EmailService implements EmailServiceInterface {
    private $mailer;

    public function sendWelcomeEmail(User $user) {
        $subject = "Welcome to our platform!";
        $body = "Hello {$user->getName()}, welcome!";

        return $this->send($user->getEmail(), $subject, $body);
    }

    private function send($to, $subject, $body) {
        // Email sending logic
        return true;
    }
}
```

```
<?php
// File: src/App/Services/Contracts/EmailServiceInterface.php
namespace App\Services\Contracts;

use App\Models\User;

interface EmailServiceInterface {
    public function sendWelcomeEmail(User $user);
}
```

3. Repository Pattern

```
<?php
// File: src/App/Repositories/UserRepository.php
namespace App\Repositories;
```

```
use App\Models\User;
use App\Repositories\Contracts\UserRepositoryInterface;

class UserRepository implements UserRepositoryInterface {
    private $users = [];

    public function save(User $user) {
        $this->users[] = $user;
        return $user;
    }

    public function findByEmail($email) {
        foreach ($this->users as $user) {
            if ($user->getEmail() === $email) {
                return $user;
            }
        }
        return null;
    }

    public function findAll() {
        return $this->users;
    }
}
```

Error Handling dan Debugging

Class Not Found Errors

```
<?php
namespace App\Controllers;

// Wrong: Class not found because namespace is missing
$user = new User(); // PHP Fatal error: Class 'App\Controllers\User' not
found

// Correct options:
use App\Models\User;
$user = new User(); // Works

$user = new \App\Models\User(); // Works

$user = new Models\User(); // Only works if App\Controllers\Models\User
exists
```

Debugging Autoloader

```
<?php
class DebuggingAutoloader {
    public function loadClass($class) {
        echo "Trying to load: {$class}\n";

        // Convert to file path
        $file = str_replace('\\', '/', $class) . '.php';
        $fullPath = __DIR__ . '/src/' . $file;

        echo "Looking for file: {$fullPath}\n";

        if (file_exists($fullPath)) {
            echo "File found, loading...\n";
            require_once $fullPath;
            return true;
        } else {
            echo "File not found!\n";
            return false;
        }
    }
}

spl_autoload_register([new DebuggingAutoloader(), 'loadClass']);
```

Testing dengan Namespaces

PHPUnit Test Classes

```
<?php
// File: tests/Unit/UserTest.php
namespace Tests\Unit;

use PHPUnit\Framework\TestCase;
use App\Models\User;

class UserTest extends TestCase {
    public function testUserCreation() {
        $user = new User('John Doe', 'john@example.com');

        $this->assertEquals('John Doe', $user->getName());
        $this->assertEquals('john@example.com', $user->getEmail());
    }

    public function testUserNamespace() {
        $user = new User('Jane', 'jane@example.com');

        $this->assertInstanceOf(User::class, $user);
        $this->assertEquals('App\Models\User', get_class($user));
    }
}
```

Performance Considerations

Autoloader Performance

```
<?php
class OptimizedAutoloader {
    private $classMap = [];

    public function __construct() {
        // Pre-load class map for better performance
        $this->classMap = [
            'App\Models\User' => __DIR__ . '/src/App/Models/User.php',
            'App\Models\Post' => __DIR__ . '/src/App/Models/Post.php',
            // ... more mappings
        ];
    }

    public function loadClass($class) {
        // Check class map first (fastest)
        if (isset($this->classMap[$class])) {
            require_once $this->classMap[$class];
            return true;
        }

        // Fall back to PSR-4 logic
        return $this->loadPsr4($class);
    }

    private function loadPsr4($class) {
        // PSR-4 loading logic
    }
}
```

Composer Optimization

```
# Generate optimized autoloader
composer dump-autoload --optimize

# For production (creates class map)
composer dump-autoload --classmap-authoritative

# No-dev optimization
composer install --no-dev --optimize-autoloader
```

Best Practices

1. Namespace Naming

```
<?php
// ✅ Good – Clear, hierarchical structure
namespace App\Models;
namespace App\Controllers\Api\V1;
namespace App\Services\Payment;

// ❌ Bad – Too generic or unclear
namespace Stuff;
namespace Utils;
namespace Things;
```

2. Use Statements Organization

```
<?php
namespace App\Controllers;

// ✅ Good – Grouped and sorted
// Standard library
use DateTime;
use InvalidArgumentException;

// Third-party packages
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

// Application classes
use App\Models\User;
use App\Services\EmailService;
use App\Traits\Loggable;

class UserController {
    // Class implementation
}
```

3. File Organization

```
<?php
// ✅ Good – One class per file, matching namespace
// File: src/App/Models/User.php
namespace App\Models;

class User {
    // Implementation
}

// ❌ Bad – Multiple classes, namespace mismatch
// File: src/models.php
namespace App;
```

```
class User {}
class Post {}
class Comment {}
```

Common Pitfalls

1. Namespace vs File Path Mismatch

```
<?php
// File path: src/App/Models/User.php
namespace App\Model; // Wrong! Should be App\Models

class User {
    // This won't autoload correctly
}
```

2. Missing Leading Backslash

```
<?php
namespace App\Controllers;

class UserController {
    public function getDate() {
        // Wrong - looks for App\Controllers\DateTime
        return new DateTime();

        // Correct - global DateTime class
        return new \DateTime();
    }
}
```

3. Case Sensitivity

```
<?php
// File: src/App/models/user.php (lowercase)
namespace App\Models; // Uppercase

class User {} // Will cause autoloading issues on case-sensitive systems
```

Contoh Implementasi

Lihat file [example.php](#) untuk berbagai contoh implementasi namespaces dan autoloading di PHP.

Latihan

1. Buat struktur namespace untuk aplikasi blog dengan Models, Controllers, Services
2. Implementasikan PSR-4 autoloader sederhana
3. Buat system dengan multiple namespaces dan use statements
4. Handle name collision dengan alias

Tugas Rumah

Buat mini framework dengan struktur:

- Framework\Http\Request dan Response classes
- Framework\Database\Connection dan QueryBuilder
- Framework\View\Template engine
- App\Controllers\, App\Models\, App\Services\ namespaces
- PSR-4 autoloader implementation
- Composer integration dengan dependencies
- Unit tests dengan PHPUnit