
**Exam : Acquia Certified Back
End Specialist D10**

**Title : Acquia Certified Back End
Specialist - Drupal 10**

Version : DEMO

1.(Plugin API & Dependency Injection)

Scenario:

You are developing a custom Block plugin (`src/Plugin/Block/UserStatsBlock.php`). This block requires the `entity_type.manager` service to query user statistics. The class currently extends `BlockBase`.

According to Drupal 10 best practices, what is the required approach to inject this service into your block plugin?

- A. Implement `ContainerInjectionInterface`, extract the service in the `create()` method, and assign it to a property in `__construct()`.
- B. Implement `ContainerFactoryPluginInterface`, override the `create()` method to extract the service from the container, and pass it to `__construct()`.
- C. Call `\Drupal::entityManager()` directly inside the block's `build()` method to ensure lazy loading.
- D. Add the `@EntityManager` annotation to the block's docblock metadata.

Answer: B

Explanation:

For Plugin APIs (such as Blocks, Field Formatters, and QueueWorkers), dependency injection MUST be implemented using `ContainerFactoryPluginInterface`.

Option A (`ContainerInjectionInterface`) is strictly reserved for Controllers, Forms, and Route Access Checkers. Using it on a Plugin will result in a fatal method signature mismatch error during instantiation.

2.(Controller Dependency Injection Implementation)

Scenario:

You are writing a custom controller (`ReportController`) that implements `ContainerInjectionInterface` to inject the database service.

Which code snippet correctly implements the `create()` method?

- A.

```
public static function create(ContainerInterface $container) {  
    return new static($container->get('database'))  
};  
}
```
- B.

```
public function create(ContainerInterface $container) {$this->database = $container->get('database');  
}
```
- C.

```
public static function create() {  
    return new static(  
        \Drupal::database()  
    );  
}
```
- D.

```
public static function create(ContainerInterface $container) {  
    return $container->get('my_module.report_controller');  
}
```

Answer: A

Explanation:

The `create()` method defined by `ContainerInjectionInterface` must be declared as `public`

`static`. It acts as a factory method, extracting dependencies via `$container->get()` and returning a new instance of the current class (`new static(...)`), passing those dependencies directly into the constructor.

3.(Service Definition & YAML Syntax)

Scenario:

You are defining a custom data synchronization service in `my_module.services.yml`. The service constructor requires the core `config.factory` and `logger.factory` services as arguments.

What is the correct YAML syntax to pass these services?

A. services:

```
my_module.sync:
  class: Drupal\my_module\SyncService
  arguments: ['config.factory', 'logger.factory']
```

B. services:

```
my_module.sync:
  class: Drupal\my_module\SyncService
  arguments: ['@config.factory', '@logger.factory']
```

C. services:

```
my_module.sync:
  class: Drupal\my_module\SyncService
  inject: [config.factory, logger.factory]
```

D. services:

```
my_module.sync:
  class: Drupal\my_module\SyncService
  dependencies: ['@config.factory', '@logger.factory']
```

Answer: B

Explanation:

In Symfony/Drupal YAML service definitions, dependencies passed as arguments must be prefixed with the `@` symbol. If omitted (as in Option A), the container passes them as literal strings instead of service instances, triggering a PHP `TypeError` exception in the constructor.

4.(Altering Existing Core Services)

Scenario:

You need to override Drupal core's `breadcrumb` builder service to completely replace the underlying class with your own implementation (`\Drupal\my_module\CustomBreadcrumbBuilder`).

What is the standard Drupal 10 mechanism to silently alter an existing service definition without hacking core files?

A. Implement `hook_services_alter(&$services)` in `my_module.module`.

B. Copy the core service definition into `my_module.services.yml` and modify the class value.

C. Create a class named `MyModuleServiceProvider` extending `ServiceProviderBase` in the `src/` directory, and override the `alter()` method to change the class.

D. Create an `EventSubscriber` listening to `KernelEvents::BOOT` to overwrite the container instance.

Answer: C

Explanation:

The only officially supported architecture to alter existing services dynamically is utilizing a `ServiceProvider` class. The file must be placed in the module's `src/` directory and strictly named `[CamelCaseModuleName]ServiceProvider.php`. Inside its `alter(ContainerBuilder $container)` method, you can manipulate definitions (e.g., `$container->getDefinition('breadcrumb')->setClass(...)`).

5.(Procedural Code & Static Service Calls)

Scenario:

You are implementing `hook_user_login($account)` inside `my_module.module` to log a specific message whenever an administrator logs in.

Because `.module` files are procedural and cannot implement object-oriented injection interfaces, what is the correct way to utilize the `logger.factory` service in this context?

- A. Procedural files should never perform logging; you must extract the logic into a dedicated `EventSubscriber`.
- B. Call `\Drupal::logger('my_module')->info('Admin logged in.');` directly inside the hook implementation.
- C. Use the global `$container;` variable to extract the logger service.
- D. Add `ContainerInterface $container` as an optional secondary parameter to the hook definition.

Answer: B

Explanation:

While calling the global `\Drupal::service()` or its wrapper methods (like `\Drupal::logger()`) is strictly prohibited as an anti-pattern in Object-Oriented classes, it is the officially accepted and necessary pattern within procedural contexts like traditional `.module` hook implementations.

Acquia Certified Back End Specialist D10 - Practice Test

Part 2: Routing, Events, and Entity API Strictness (Q6 - Q10)