Laravel Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a custom Laravel service provider that registers a singleton database connection pool

Solution:

```
class ConnectionPoolServiceProvider extends ServiceProvider {
    public function register() {
        $this->app->singleton('db.pool', function ($app) {
            return new DatabaseConnectionPool(
                config('database.pool.max_connections'),
                $app['db.factory']
            );
        });
    }
}
```

2. Create a Laravel job that implements retry with exponential backoff

```
class ProcessPayment implements ShouldQueue {
    use Dispatchable, InteractsWithQueue, Queueable;
    public $tries = 3;
    public function backoff() {
        return [1, 5, 15]; // Seconds
    }
    public function handle() {
        // Payment logic
    }
}
```

3. Implement a custom Laravel validation rule for checking JSON schema compliance

```
class JsonSchemaRule implements Rule {
    private $schema;
    public function passes($attribute, $value) {
        $validator = new JsonSchemaValidator;
        return $validator->validate(
            json_decode($value),
            $this->schema
        );
    }
}
```

4. Create a middleware that implements rate limiting based on user roles

```
class RoleBasedThrottleMiddleware {
    public function handle($request, $next) {
        $limit = $request->user()->role === 'premium' ? 100 : 20;
        return RateLimiter::attempt(
            'api:'.$request->user()->id,
            $limit,
            function() use ($next, $request) {
                return $next($request);
            }
        );
    }
}
```

5. Implement a custom Laravel cache driver with automatic key expiration

```
class AutoExpiringStore extends Store {
    protected function putMany(array $values, $seconds) {
        $this->storage->pipeline(function($pipe) use ($values) {
            foreach($values as $key => $value) {
                $pipe->setex($key, $seconds, $value);
            }
        });
    }
}
```

6. Create a Laravel event listener that handles concurrent database updates

```
class HandleConcurrentUpdate implements ShouldQueue {
    public function handle($event) {
        DB::transaction(function () use ($event) {
            $record = Model::lockForUpdate()->find($event->id);
            // Update logic with pessimistic locking
        });
    }
}
```

7. Implement a custom Eloquent macro for complex JSON querying

```
Builder::macro('whereJsonContains', function($column, $value) {
    return $this->whereRaw(
        'JSON_CONTAINS('.$column.', ?)',
        [DB::raw(json_encode($value))]
    );
});
```

8. Create a Laravel console command that processes large datasets in chunks

```
class ProcessLargeDataset extends Command {
    public function handle() {
        User::chunk(1000, function($users) {
            foreach($users as $user) {
                ProcessUserJob::dispatch($user);
            }
        });
    }
}
```

9. Implement a custom Laravel facade for a third-party API service

```
class ApiService extends Facade {
    protected static function getFacadeAccessor() {
        return 'api.service';
    }
    public static function boot() {
        static::$app->singleton('api.service', fn() =>
            new ApiServiceImpl(config('services.api')));
    }
}
```

10. Create a custom Laravel collection macro for complex data transformation

```
Collection::macro('transformNested', function ($callback) {
    return $this->map(function ($value) use ($callback) {
        return is_array($value)
            ? collect($value)->transformNested($callback)
            : $callback($value);
    });
});
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU Cache in Laravel using a custom collection

Solution:

Here's an efficient LRU Cache implementation:

```
class LRUCache {
    private $capacity;
    private $cache;

    public function __construct($capacity) {
        $this->capacity = $capacity;
        $this->cache = collect();
    }

    public function get($key) {
        if (!$this->cache->has($key)) return -1;
        $value = $this->cache->pull($key);
        $this->cache->put($key, $value);
        return $value;
    }}
```

Time Complexity: O(1) for both get and put operations

2. Design a rate limiter using Redis in Laravel

Implementation:

```
public function handle($request, Closure $next) {
    $key = 'rate_limit:' . $request->ip();
    $limit = 60;
    $window = 60;

    $current = Redis::get($key) ?: 0;
    if ($current >= $limit) return response('Too Many Requests', 429);

    Redis::incr($key);
    Redis::expire($key, $window);
```

Key Points:

- Uses sliding window approach
- O(1) time complexity
- Distributed rate limiting

3. Implement a custom collection macro for finding duplicate values efficiently

Solution:

```
Collection::macro('findDuplicates', function () {
    return $this->filter(function ($value, $key) {
        return $this->where($value)->count() > 1;
    })->unique();
```

Usage:

```
$duplicates = collect([1, 2, 2, 3, 3, 4])->findDuplicates();
```

Time Complexity: O(n) where n is collection size

4. Create a sliding window implementation for processing large datasets in chunks

Implementation:

```
public function processBatch($items, $windowSize = 1000) {
    return $items->lazy()->chunk($windowSize)->map(function ($chunk) {
        return $chunk->map(fn($item) => $this->process($item));
    })->collapse();
```

Benefits:

- Memory efficient processing
- Handles large datasets
- Maintains constant memory usage

5. Implement a custom priority queue for job processing

Solution:

```
class PriorityQueue {
    private $high = [];
    private $low = [];

    public function push($job, $priority = 'low') {
        $queue = $priority === 'high' ? 'high' : 'low';
        array_push($this->{$queue}, $job);
    }}
```

Time Complexity:

- Push: O(1)
- Pop: O(1)

6. Design a custom caching strategy using Laravel's cache repository

Implementation:

```
public function remember($key, $ttl, Closure $callback) {
    if ($value = Cache::get($key)) return $value;
    $value = $callback();
    Cache::put($key, $value, now()->addMinutes($ttl));
    return $value;
```

Features:

- Automatic cache invalidation
- Lazy loading
- Race condition prevention

7. Implement an efficient search algorithm for hierarchical data

Solution:

```
public function searchTree($node, $searchValue) {
    if ($node->value === $searchValue) return $node;
    foreach ($node->children as $child) {
        if ($result = $this->searchTree($child, $searchValue)) {
            return $result;
        }
    }
```

Time Complexity: O(n) where n is number of nodes

8. Create a custom collection pipeline for data transformation

Implementation:

```
return $collection->pipe(function ($items) {
    return $items->filter()->map(fn($item) => transform($item))
        ->reject(fn($item) => isEmpty($item))
        ->values();
```

Benefits:

- Chainable operations
- Memory efficient
- Readable code structure

9. Implement a throttling mechanism using Laravel's cache

Solution:

```
public function throttle($key, $limit, $decay) {
    $attempts = Cache::get($key, 0);
    if ($attempts >= $limit) return false;
    Cache::put($key, $attempts + 1, $decay);
    return true;
```

Use Cases:

- API rate limiting
- Login attempts
- Resource access control

10. Design an efficient batch processing system for Eloquent models

Implementation:

```
public function processBatch(Collection $models, $chunkSize = 100) {
    return $models->chunk($chunkSize)->each(function ($chunk) {
        $chunk->each(fn($model) => $this->process($model));
    });
```

Advantages:

- Memory efficient
- Prevents timeout issues
- Handles large datasets

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly using Laravel. What would be your architectural approach?

Key Components:

- **Base62 Encoding** for converting long URLs to short codes
- **Distributed Cache** (Redis) for frequently accessed URLs
- **Load Balancer** for horizontal scaling

Implementation:

```
class UrlShortener {
    public function encode($url) {
        $hash = md5($url);
        return base_convert(substr($hash, 0, 8), 16, 36);
    }
}
```

Scaling Considerations:

- Cache frequently accessed URLs in Redis
- Use queue workers for URL validation
- Implement rate limiting

2. How would you design a real-time notification system in Laravel that can handle millions of users?

Architecture Components:

- **WebSockets** using Laravel Echo and Redis
- **Queue System** for async processing
- **Horizontal Scaling** with multiple socket servers

```
broadcast(new NotificationEvent($data))
    ->toOthers()
    ->queue();
```

Scaling Strategy:

- Use Redis pub/sub for real-time events
- Implement message queuing with Laravel Horizon
- Shard notifications by user groups

3. Design a social media feed system with Laravel that can handle high read/write operations.

Core Components:

- **Content Delivery Network (CDN)** for media
- **Caching Layer** for feed composition
- **Fan-out Service** for post distribution

```
class FeedService {
    public function getFeed($userId) {
        return Cache::remember("feed:{$userId}", 3600, fn() =>
            Post::whereIn('user_id', $following)->latest()->get());
```

```
    }
}
```

4. How would you implement a distributed job processing system using Laravel for handling millions of background tasks?

Architecture:

- **Multiple Queue Workers** across servers
- **Redis/RabbitMQ** as message broker
- **Supervisor** for process management

```
Queue::bulk([
    new ProcessVideoJob($video),
    new GenerateThumbnailJob($video),
    new NotifySubscribersJob($video)
]);
```

Scaling:

- Implement retry mechanisms
- Monitor queue metrics
- Use job batching for related tasks

5. Design a rate limiting system for a REST API in Laravel that can handle distributed servers.

Implementation Approach:

- **Redis** for distributed rate limiting
- **Sliding Window** algorithm
- **Circuit Breaker** pattern

```
$limiter = Redis::throttle('key')
    ->allow(100)->every(60)
    ->block(5);

return $limiter->then(fn() => $response);
```

Features:

- IP-based and token-based limiting
- Custom throttling rules
- Rate limit headers

6. How would you design a scalable e-commerce cart system with Laravel considering high concurrency?

Architecture Components:

- **Redis** for cart storage
- **Optimistic Locking** for inventory
- **Event Sourcing** for cart changes

```
class Cart {
    public function addItem($productId, $quantity) {
        Redis::multi();
        Redis::hincrby("cart:{$userId}", $productId, $quantity);
        Redis::exec();
    }
}
```

7. Design a content caching system for a high-traffic news website using Laravel.

Caching Strategy:

- **Multi-level Cache** (Redis + CDN)
- **Cache Tags** for invalidation

- **Edge Caching** for static content

```
Cache::tags(['news', 'featured'])->remember($key, $ttl, function() {
   return News::with('categories')->latest()->get();
});
```

Invalidation:

- Implement cache warming
- Use queue for cache updates

8. How would you implement a real-time search suggestion system with Laravel?

Components:

- **Elasticsearch** for search index
- **Redis** for trending searches
- **WebSocket** for real-time suggestions

```
class SearchService {
   public function suggest($query) {
      return Redis::zrevrange("trending:searches", 0, 5)
         ->union(Elasticsearch::suggest($query));
   }
}
```

9. Design a scalable file upload system in Laravel that can handle large files and high concurrency.

Architecture:

- **Chunked Upload** implementation
- **S3/Cloud Storage** integration
- **Background Processing** for post-upload tasks

```
class ChunkedUpload {
   public function store($chunk, $index, $total) {
      Storage::disk('local')->append($tempFile, $chunk);
      dispatch(new ProcessUploadJob($tempFile));
   }
}
```

10. How would you design a multi-tenant architecture in Laravel for a SaaS application?

Key Features:

- **Database Separation** strategies
- **Domain Resolution** system
- **Resource Isolation**

```
class TenantManager {
   public function identify($domain) {
      return Cache::remember("tenant:{$domain}", 3600,
         fn() => Tenant::where('domain', $domain)->first());
   }
}
```

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Laravel service provider that implements a custom logging channel with rotation based on file size.

Solution:

```
class CustomLogServiceProvider extends ServiceProvider {
  public function register() {
    $this->app->singleton('custom.logger', function($app) {
      return new Logger('custom', [
        new RotatingFileHandler(
          storage_path('logs/custom.log'),
          5, // Keep 5 files
          Logger::INFO,
          true, // Rotate on size
          5242880 // 5MB
        )
      ]);
    });
  }
}
```

Key points:

- Uses Monolog's RotatingFileHandler for size-based rotation
- Configures 5MB max file size with 5 backup files
- Registered as a singleton for app-wide access

2. Create a Laravel middleware that rate limits API requests based on both IP address and API token.

Solution:

```
class CustomRateLimiter {
  public function handle($request, $next) {
    $key = $request->ip() . ':' . $request->bearerToken();
    $limit = Cache::remember($key, 60, fn() => 0);
    if ($limit > 100) throw new TooManyRequestsException;
    Cache::increment($key);
    return $next($request);
  }
```

Features:

- Combines IP and token for unique limiting
- Uses Cache facade for tracking
- 100 requests per minute limit
- Throws 429 Too Many Requests on limit exceed

3. Implement a Laravel job that processes a large CSV file in chunks while maintaining memory efficiency.

Solution:

```
class ProcessLargeCsvJob implements ShouldQueue {
  public function handle() {
    $path = storage_path('app/large.csv');
    (new LazyCollection(function() use($path) {
```

```
    $handle = fopen($path, 'r');
    while ($line = fgetcsv($handle)) yield $line;
    fclose($handle);
  }))->chunk(1000)->each(fn($chunk) =>
    dispatch(new ProcessChunkJob($chunk)));
}
```

Benefits:

- Uses LazyCollection for memory efficiency
- Processes 1000 records per chunk
- Spawns sub-jobs for parallel processing

4. Write a Laravel macro that extends the Collection class to find duplicate values based on a specific key.

Solution:

```
Collection::macro('findDuplicatesBy', function($key) {
  return $this->groupBy($key)
    ->filter(function($group) {
      return $group->count() > 1;
    })->map(function($group) {
      return $group->values();
    });
});
```

Usage:

- Register in ServiceProvider boot method
- Call as: $collection->findDuplicatesBy('email')
- Returns grouped collection of duplicates
- Maintains original object structure

5. Create a custom Laravel validation rule that ensures a JSON field contains specific required nested properties.

Solution:

```
class JsonStructureRule implements Rule {
  public function passes($attribute, $value) {
    $data = json_decode($value, true);
    return is_array($data) &&
      array_key_exists('config', $data) &&
      array_key_exists('type', $data['config']) &&
      array_key_exists('version', $data['config']);
  }
```

Implementation details:

- Validates JSON structure depth
- Checks for required nested keys
- Handles invalid JSON gracefully
- Can be used in form requests

6. Implement a Laravel observer that maintains a full audit trail of model changes in a separate table.

Solution:

```
class AuditObserver {
  public function updated($model) {
    AuditLog::create([
      'model_type' => get_class($model),
      'model_id' => $model->id,
      'changes' => json_encode($model->getDirty()),
      'user_id' => auth()->id()
    ]);
  }
```

Features:

- Tracks all model changes automatically
- Records user responsible for changes
- Stores original and new values
- Enables audit trail queries

7. Write a Laravel command that generates a performance report of slow database queries from the query log.

Solution:

```
class AnalyzeQueriesCommand extends Command {
  public function handle() {
    return collect(DB::getQueryLog())
      ->filter(fn($query) => $query['time'] > 100)
      ->groupBy('query')
      ->map(fn($group) => [
        'count' => $group->count(),
        'avg_time' => $group->avg('time')
      ]);
```

Capabilities:

- Groups similar queries together
- Calculates query frequency
- Identifies slow queries (>100ms)
- Provides average execution times

8. Create a Laravel facade that wraps a third-party API client with proper error handling and retry logic.

Solution:

```
class ApiWrapper {
  public function request($method, $endpoint, $data = []) {
    return retry(3, function() use ($method, $endpoint, $data) {
      try {
        return Http::timeout(5)->$method($endpoint, $data);
      } catch (Exception $e) {
        report($e);
        throw $e;
      }
    }, 100);
```

Features:

- Automatic retry on failure
- Configurable timeout
- Exception logging
- Flexible HTTP method support

9. Implement a Laravel event listener that processes queued notifications in batches with error handling.

Solution:

```
class BatchNotificationListener implements ShouldQueue {
  public function handle($event) {
    Notification::send($event->users->chunk(100),
      new BatchableNotification($event->data))
      ->catch(function($e) {
        $this->handleFailedBatch($e);
      });
  }
```

Key aspects:

- Processes users in chunks of 100

- Queued for background processing
- Handles notification failures
- Maintains batch integrity

10. Write a Laravel test case that verifies the behavior of a complex authentication flow with multiple guards.

Solution:

```
class MultiAuthTest extends TestCase {
  public function test_multiple_guard_auth() {
    $response = $this->post('/api/auth', [
      'token' => $this->validToken,
      'guard' => 'api'
    ])->assertAuthenticated('api')
      ->assertNotAuthenticated('web');
  }
```

Test coverage:

- Verifies guard-specific auth
- Checks cross-guard isolation
- Validates token authentication
- Ensures proper session handling