Symfony

Interview Questions and Answers

Core Concepts

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

1. Explain the Symfony Request-Response lifecycle and the role of the HttpKernel component

Request-Response Lifecycle

The **HttpKernel component** is the core of Symfony's request handling mechanism. The lifecycle follows these stages:

- **kernel.request**: Event fired before controller execution, allows request modification or early response
- **Controller Resolution**: ControllerResolver determines which controller to execute based on routing
- **kernel.controller**: Event fired after controller is resolved but before execution
- **Controller Execution**: The actual controller logic runs and returns a Response or other value
- **kernel.view**: Transforms non-Response returns into Response objects
- **kernel.response**: Allows response modification before sending to client
- **kernel.terminate**: Fired after response is sent, for cleanup tasks
- **kernel.exception**: Handles exceptions thrown during any stage

This event-driven architecture enables powerful middleware patterns through **event listeners and subscribers**, allowing cross-cutting concerns like authentication, caching, and logging to be implemented cleanly without modifying core application logic.

2. How does Symfony's Dependency Injection Container work, and what are the differences between services, arguments, and autowiring?

Dependency Injection Container

Symfony's **DI Container** manages object instantiation and dependency resolution. Key concepts:

- **Services**: Objects managed by the container, defined in configuration files (YAML, XML, PHP) with unique IDs
- **Arguments**: Dependencies passed to service constructors, can be other services, parameters, or scalar values
- **Autowiring**: Automatic dependency resolution based on type-hints, eliminating manual configuration for most services

Example service definition:

```
services:
  App\Service\PaymentProcessor:
    arguments:
      $apiKey: '%payment.api_key%'
      $logger: '@logger'
    tags:
      - { name: 'app.payment_handler' }
```

With **autowiring enabled**, the container automatically injects dependencies by matching constructor parameter types to registered services. The container compiles to optimized PHP code for production, ensuring minimal runtime overhead. Advanced features include **service decoration**, **lazy loading**, and **compiler passes** for custom container manipulation.

3. What are Symfony's security components, and how do you implement a custom authentication system?

Security Architecture

Symfony's security system consists of several key components:

- **Firewall**: Defines protected areas of your application and authentication methods
- **Authentication**: Verifies user identity through authenticators (Guard, JWT, form login, etc.)
- **Authorization**: Controls access through voters, access control rules, and role hierarchy
- **User Provider**: Loads user data from storage (database, LDAP, memory)

Custom authentication example using Authenticator:

```
class ApiKeyAuthenticator extends AbstractAuthenticator
{
  public function authenticate(Request $request): Passport
  {
    $apiKey = $request->headers->get('X-API-KEY');
    return new SelfValidatingPassport(
      new UserBadge($apiKey, fn($key) => $this->loadUser($key))
    );
  }
}
```

The **Passport** system (Symfony 5.3+) provides a flexible authentication framework with badges for credentials, CSRF tokens, and remember-me functionality. **Voters** enable fine-grained authorization logic by implementing complex business rules for resource access decisions.

4. Describe Symfony's Event Dispatcher pattern and when you would use event subscribers versus event listeners

Event Dispatcher Pattern

The **EventDispatcher** implements the observer pattern, enabling decoupled communication between components. Key differences:

- **Event Listeners**: Configured externally, single method per service, priority set in configuration, better for third-party bundles
- **Event Subscribers**: Self-configured through getSubscribedEvents(), can handle multiple events, priority defined in code, better for application logic

Event Subscriber example:

```
class OrderSubscriber implements EventSubscriberInterface
{
  public static function getSubscribedEvents(): array
  {
    return [
      OrderPlacedEvent::class => ['onOrderPlaced', 10],
      OrderCancelledEvent::class => 'onOrderCancelled'
    ];
  }
}
```

Use listeners when: Configuration flexibility is needed, or integrating third-party code. **Use subscribers when**: Building application features that need to react to multiple events, or when keeping event handling logic self-contained. Subscribers provide better discoverability since event bindings are in code rather than configuration.

5. How does Doctrine ORM integrate with Symfony, and what are the best practices for handling entity relationships and query optimization?

Doctrine Integration

Doctrine ORM provides **object-relational mapping** in Symfony through entity managers, repositories, and the DQL query language. Best practices include:

- **Lazy Loading**: Use fetch="LAZY" for associations to avoid N+1 queries, but be aware of proxy objects
- **Eager Loading**: Use JOIN FETCH in DQL or QueryBuilder for predictable query patterns
- **Batch Processing**: Use iterate() and flush/clear cycles for large datasets
- **Indexes**: Add database indexes via @ORM\Index annotations on frequently queried fields

Optimized query example:

```
$query = $repository->createQueryBuilder('o')
  ->select('o', 'i', 'c')
  ->leftJoin('o.items', 'i')
  ->leftJoin('o.customer', 'c')
  ->where('o.status = :status')
  ->setParameter('status', 'pending')
  ->getQuery();
```

Use **partial objects** with select() for read-only operations, implement **result caching** for expensive queries, and leverage **second-level cache** for entity caching. Always profile queries using the Symfony profiler toolbar to identify performance bottlenecks.

6. Explain Symfony's Form component architecture and how to implement custom form types with data transformers

Form Component Architecture

Symfony Forms provide a **declarative way to build and validate forms**. The architecture consists of:

- **Form Types**: Define field structure, options, and behavior
- **Data Transformers**: Convert between view and model data formats
- **Form Events**: PRE_SET_DATA, POST_SET_DATA, PRE_SUBMIT, SUBMIT, POST_SUBMIT for dynamic form modification
- **Constraints**: Validation rules applied to data

Custom form type with transformer:

```
class TagsInputType extends AbstractType
{
  public function buildForm(FormBuilderInterface $builder, array $options)
  {
    $builder->addModelTransformer(new CallbackTransformer(
      fn($tagsArray) => implode(', ', $tagsArray),
      fn($tagsString) => array_map('trim', explode(',', $tagsString))
    ));
  }
}
```

Data transformers handle conversion between normalized (model) and denormalized (view) data. Use **model transformers** for business logic conversions and **view transformers** for presentation formatting. Form events enable dynamic field addition based on submitted data or user roles.

7. What are Symfony's caching strategies, and how do you implement HTTP caching with ESI and reverse proxies?

Caching Strategies

Symfony provides multiple caching layers:

- **HTTP Cache**: Browser and proxy caching using Cache-Control, ETag, and Last-Modified headers
- **ESI (Edge Side Includes)**: Fragment caching for partial page updates
- **Application Cache**: PSR-6/PSR-16 compliant cache pools for data caching
- **Reverse Proxy**: Symfony HttpCache or Varnish for server-side HTTP caching

ESI implementation:

```
// Controller
$response->setSharedMaxAge(3600);

// Template
{{ render_esi(controller('App\\Controller::sidebar', {
  'maxAge': 600
})) }}

// Enable in kernel
$kernel = new AppKernel($env, $debug);
```

```
$kernel = new HttpCache($kernel);
```

Cache invalidation can be handled via cache tags (with Varnish xkey module) or BAN requests. Use **validation caching** (ETag/Last-Modified) for frequently changing content and **expiration caching** (Cache-Control) for stable content. The **HttpCache** kernel wrapper provides Varnish-like functionality without additional infrastructure.

8. How do you implement API versioning and content negotiation in Symfony REST APIs?

API Versioning & Content Negotiation

Symfony supports multiple API versioning strategies:

- **URI Versioning**: /api/v1/users - simple but creates URL proliferation
- **Header Versioning**: Accept: application/vnd.myapi.v2+json - cleaner URLs, proper HTTP semantics
- **Parameter Versioning**: /api/users?version=2 - flexible but non-standard

Content negotiation with FOSRestBundle:

```
// config/packages/fos_rest.yaml
fos_rest:
  format_listener:
    rules:
      - { path: '^/api', priorities: ['json', 'xml'], prefer_extension: false }
  versioning:
    enabled: true
    resolvers:
      header:
        enabled: true
```

Implement **custom version resolvers** by extending AbstractVersionResolver. Use **serializer groups** to control field exposure per version. The **Accept header** should drive content type negotiation, while API version can be in custom headers (X-API-Version) or media type parameters. Always maintain backwards compatibility and provide clear deprecation notices in response headers.

9. Explain Symfony's Messenger component and how to implement asynchronous message processing with different transports

Messenger Component

The **Messenger component** enables asynchronous processing through message buses, handlers, and transports:

- **Message Bus**: Dispatches messages to handlers
- **Messages**: Plain PHP objects representing commands, events, or queries
- **Handlers**: Process messages, can be synchronous or asynchronous
- **Transports**: Queue systems (AMQP, Redis, Doctrine, Amazon SQS)

Configuration example:

```
// config/packages/messenger.yaml
messenger:
  transports:
    async: '%env(MESSENGER_TRANSPORT_DSN)%'
    failed: 'doctrine://default?queue_name=failed'
  routing:
    App\Message\SendEmail: async
    App\Message\ProcessOrder: async
```

Message handler:

```
class SendEmailHandler implements MessageHandlerInterface
{
  public function __invoke(SendEmail $message)
  {
    // Process message
  }
}
```

Use **middleware** for cross-cutting concerns like validation, logging, and transaction handling. Implement **retry strategies** with exponential backoff for failed messages. The **stamp system** allows attaching metadata to messages for routing, delays, and priority handling.

10. What are Symfony's best practices for testing, including functional tests, unit tests, and database fixtures?

Testing Best Practices

Symfony provides comprehensive testing tools built on PHPUnit:

- **Unit Tests**: Test individual classes in isolation using mocks and stubs
- **Functional Tests**: Test HTTP requests/responses using WebTestCase and the test client
- **Integration Tests**: Test service interactions with real container
- **Fixtures**: Use DoctrineFixturesBundle or Foundry for test data

Functional test example:

```
class UserControllerTest extends WebTestCase
{
  public function testUserCreation(): void
  {
    $client = static::createClient();
    $client->request('POST', '/api/users', [], [],
      ['CONTENT_TYPE' => 'application/json'],
      json_encode(['email' => 'test@example.com'])
    );
    $this->assertResponseIsSuccessful();
  }
}
```

Use **test database** with doctrine:database:create --env=test, implement **data fixtures** with dependencies, and leverage **Foundry** for factory-based test data generation. The **Profiler** can be used in tests to assert on events, queries, and emails. Use **DAMADoctrineTestBundle** to wrap tests in transactions for database isolation.

Data Structures and Algorithms

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

1. How would you implement a stack data structure in Symfony/PHP and what is its time complexity?

Stack Implementation

A stack follows **LIFO (Last In First Out)** principle. In PHP, you can use **SplStack** or a simple array.

```
class Stack {
  private array $items = [];
  public function push($item) { $this->items[] = $item; }
  public function pop() { return array_pop($this->items); }
  public function peek() { return end($this->items); }
  public function isEmpty(): bool { return empty($this->items); }
}
```

Time Complexity:

- Push: O(1)
- Pop: O(1)
- Peek: O(1)
- Space Complexity: O(n)

2. Implement an LRU (Least Recently Used) Cache in PHP with O(1) operations. How would you integrate it with Symfony's cache system?

LRU Cache Implementation

Use a **doubly linked list** with a **hash map** for O(1) get/put operations.

```
class LRUCache {
  private $capacity, $cache = [], $list = [];
  public function __construct(int $cap) { $this->capacity = $cap; }
  public function get($key) {
    if (!isset($this->cache[$key])) return -1;
    $this->moveToFront($key);
    return $this->cache[$key];
  }
}
```

Symfony Integration: Implement **CacheItemPoolInterface** or extend **AbstractAdapter**. Use it as a custom cache adapter in config/packages/cache.yaml with PSR-6 compliance.

Time Complexity: Get: O(1), Put: O(1), Space: O(capacity)

3. How do you find all pairs in an array that sum to a target value? Optimize for Symfony service handling large datasets.

Pair Sum Problem

Use a **hash map** for O(n) time complexity instead of nested loops O(n²).

```
function findPairs(array $nums, int $target): array {
  $seen = [];
  $pairs = [];
  foreach ($nums as $num) {
    if (isset($seen[$target - $num])) {
      $pairs[] = [$num, $target - $num];
    }
    $seen[$num] = true;
  }
  return $pairs;
}
```

Symfony Optimization:

- Use **MessageBus** for async processing of large datasets
- Implement **ChunkIterator** for memory-efficient batch processing
- Cache results using **TagAwareCacheInterface**
- Use **Doctrine pagination** for database queries

Time Complexity: O(n), Space: O(n)

4. Implement a sliding window algorithm to find the maximum sum of k consecutive elements. How would you apply this in Symfony for analytics?

Sliding Window Maximum Sum

The **sliding window technique** optimizes consecutive element operations from O(n*k) to O(n).

```
function maxSumSubarray(array $arr, int $k): int {
  $maxSum = $windowSum = array_sum(array_slice($arr, 0, $k));
  for ($i = $k; $i < count($arr); $i++) {
    $windowSum = $windowSum - $arr[$i - $k] + $arr[$i];
    $maxSum = max($maxSum, $windowSum);
  }
  return $maxSum;
}
```

Symfony Analytics Use Cases:

- Calculate rolling averages for metrics dashboards
- Process time-series data in **Doctrine repositories**
- Implement real-time analytics with **Mercure** updates
- Use with **Messenger** for streaming data processing

Time Complexity: O(n), Space: O(1)

5. How do you implement a priority queue in PHP and use it with Symfony Messenger for task scheduling?

Priority Queue Implementation

PHP's **SplPriorityQueue** implements a max-heap by default.

```
class TaskQueue extends \SplPriorityQueue {
  public function compare($priority1, $priority2): int {
    return $priority1 <=> $priority2;
  }
}
$queue = new TaskQueue();
$queue->insert('task1', 5);
$queue->insert('task2', 10);
```

Symfony Messenger Integration:

- Use **transport priorities** in messenger.yaml
- Implement **StampInterface** for custom priority stamps
- Configure multiple transports with **routing** by priority
- Use **Doctrine transport** with queue_name for priority lanes

Time Complexity: Insert: O(log n), Extract: O(log n)

6. Implement a binary search algorithm and explain how to use it with Symfony's Doctrine for optimized queries.

Binary Search Implementation

Binary search requires a sorted array and provides O(log n) search time.

```
function binarySearch(array $arr, $target): int {
  $left = 0;
  $right = count($arr) - 1;
  while ($left <= $right) {
    $mid = $left + (int)(($right - $left) / 2);
    if ($arr[$mid] === $target) return $mid;
    $arr[$mid] < $target ? $left = $mid + 1 : $right = $mid - 1;
  }
  return -1;
}
```

Doctrine Optimization:

- Add **indexes** on searchable columns (enables B-tree search)
- Use **BETWEEN** queries for range searches
- Implement **QueryBuilder** with ORDER BY for sorted results
- Use **Result Cache** for frequently searched data

Time Complexity: O(log n), Space: O(1)

7. How do you detect a cycle in a linked list? Relate this to detecting circular dependencies in Symfony services.

Cycle Detection - Floyd's Algorithm

Use **Floyd's Cycle Detection** (tortoise and hare) with two pointers.

```
function hasCycle($head): bool {
  $slow = $fast = $head;
  while ($fast && $fast->next) {
    $slow = $slow->next;
    $fast = $fast->next->next;
    if ($slow === $fast) return true;
  }
  return false;
}
```

Symfony Circular Dependency Detection:

- Symfony's **DependencyInjection component** detects cycles at compile time
- Use **debug:container** command to visualize service graph
- Implement **setter injection** or **service locators** to break cycles
- Review **ServiceCircularReferenceException** traces
- Use **lazy services** with proxies to defer instantiation

Time Complexity: O(n), Space: O(1)

8. Implement a hash map from scratch in PHP. How does Symfony's ParameterBag use hash maps internally?

Hash Map Implementation

A hash map uses an array with **hash function** for O(1) average access.

```
class HashMap {
  private array $buckets = [];
  private int $size = 16;
  public function put($key, $value) {
    $index = $this->hash($key);
    $this->buckets[$index] = $value;
  }
  private function hash($key): int {
    return crc32($key) % $this->size;
  }
}
```

Symfony ParameterBag:

- Uses PHP's native **associative arrays** (hash tables)
- Provides **get(), set(), has(), all()** methods with O(1) access
- Implements **case-insensitive** key access for headers
- Used in **Request, Session, and Container** components

Time Complexity: Get/Put: O(1) average, O(n) worst case

9. How do you implement a Trie (prefix tree) for autocomplete functionality in a Symfony application?

Trie Implementation

A **Trie** is optimal for prefix-based searches like autocomplete.

```
class TrieNode {
  public array $children = [];
  public bool $isEnd = false;
}
class Trie {
  private TrieNode $root;
  public function __construct() { $this->root = new TrieNode(); }
  public function insert(string $word) { /* traverse and insert */ }
  public function search(string $prefix): array { /* return matches */ }
}
```

Symfony Implementation:

- Store Trie in **Redis** or **APCu cache** for fast access
- Use **Elasticsearch** with completion suggester for production
- Implement as a **service** with caching layer
- Build index via **console command** or **Messenger handler**
- Expose via **API Platform** with search filter

Time Complexity: Insert: O(m), Search: O(m), m=word length

10. Implement merge sort and explain how Symfony's collection sorting leverages similar algorithms.

Merge Sort Implementation

Merge sort is a stable, divide-and-conquer algorithm with O(n log n) complexity.

```
function mergeSort(array $arr): array {
  if (count($arr) <= 1) return $arr;
  $mid = (int)(count($arr) / 2);
  $left = mergeSort(array_slice($arr, 0, $mid));
  $right = mergeSort(array_slice($arr, $mid));
  return merge($left, $right);
}
function merge($left, $right) { /* merge logic */ }
```

Symfony Collection Sorting:

- **Doctrine Collections** use PHP's usort() (Quicksort/IntroSort hybrid)
- **Criteria::orderBy()** for in-memory sorting
- Database-level ORDER BY for large datasets
- Use **Comparable interface** for custom sorting
- **ArrayCollection::matching()** with Criteria for complex sorts

Time Complexity: O(n log n), Space: O(n)

System Design

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

1. Design a URL shortener service using Symfony. What architectural decisions would you make to ensure scalability and high availability?

Architecture Overview

A URL shortener requires careful design for **high read throughput**, **low latency**, and **global distribution**.

Key Components

- **Short URL Generation:** Use base62 encoding with auto-increment IDs or distributed ID generation (Snowflake pattern)
- **Database Strategy:** PostgreSQL for writes with read replicas; Redis for caching hot URLs (80/20 rule)
- **Symfony Implementation:** Use Doctrine for persistence, Messenger for async analytics, and API Platform for REST endpoints
- **Caching Layer:** Implement multi-tier caching (Redis L1, CDN L2) with cache-aside pattern
- **Load Balancing:** Stateless application servers behind HAProxy/Nginx with sticky sessions disabled

Scalability Considerations

- **Horizontal Scaling:** Stateless Symfony apps in Kubernetes with HPA based on request rate
- **Database Sharding:** Shard by hash of short code for write distribution
- **CAP Theorem:** Favor availability and partition tolerance (AP); eventual consistency acceptable for analytics
- **Rate Limiting:** Use Symfony Rate Limiter component with Redis backend

```
// URL Shortener Service
class UrlShortener {
  public function shorten(string $url): string {
    $id = $this->idGenerator->next();
    $short = $this->base62Encode($id);
    $this->cache->set($short, $url, 3600);
    $this->repository->save($short, $url);
    return $short;
  }
}
```

Monitoring

Implement distributed tracing (Jaeger), metrics (Prometheus), and centralized logging (ELK stack) for observability.

2. How would you design a real-time notification system in Symfony that supports millions of concurrent WebSocket connections?

System Architecture

Real-time notifications require a **push-based architecture** with persistent connections and message distribution.

Technology Stack

- **WebSocket Server:** Ratchet or Swoole for PHP, or Node.js microservice for better connection handling

- **Message Broker:** RabbitMQ or Redis Pub/Sub for message distribution across WebSocket servers
- **Symfony Integration:** Use Messenger to publish events, consumed by WebSocket workers
- **Connection Management:** Redis to track user-to-server mapping for targeted delivery

Scalability Design

- **Horizontal Scaling:** Multiple WebSocket servers behind TCP load balancer (not HTTP)
- **Sticky Connections:** Use consistent hashing to route reconnections to same server
- **Stateful vs Stateless:** WebSocket servers are stateful; application logic remains stateless
- **Backpressure Handling:** Implement client-side buffering and server-side queue limits

```
// Notification Publisher
class NotificationPublisher {
  public function notify(User $user, array $data) {
    $message = new NotificationMessage($user->getId(), $data);
    $this->bus->dispatch($message);
    $this->redis->publish('notifications', json_encode($message));
  }
}
```

Reliability Patterns

- **Delivery Guarantees:** At-least-once delivery with client-side deduplication using message IDs
- **Offline Support:** Queue notifications in database for offline users, deliver on reconnect
- **Fallback Mechanism:** Long-polling endpoint for clients that cannot maintain WebSocket

3. Design a multi-tenant SaaS application in Symfony. How would you handle data isolation, tenant identification, and schema management?

Multi-Tenancy Approaches

Three main strategies exist, each with tradeoffs:

- **Separate Database per Tenant:** Maximum isolation, complex maintenance
- **Shared Database, Separate Schema:** Good isolation, moderate complexity
- **Shared Schema with Tenant ID:** Easiest to scale, requires careful query filtering

Recommended Hybrid Approach

Use **shared schema with tenant discrimination** for most tenants, dedicated databases for enterprise clients.

Symfony Implementation

- **Tenant Identification:** Subdomain-based routing or JWT claims; store in request attributes
- **Doctrine Filters:** Global filter to automatically add tenant_id to all queries
- **Connection Management:** Dynamic DBAL connection switching based on tenant
- **Cache Isolation:** Prefix cache keys with tenant ID to prevent data leakage

```
// Doctrine Tenant Filter
class TenantFilter extends SQLFilter {
  public function addFilterConstraint($entity, $alias) {
    if (!$entity->hasField('tenant_id')) return '';
    $tenantId = $this->getParameter('tenant_id');
    return "$alias.tenant_id = $tenantId";
  }
}
```

Security Considerations

- **Row-Level Security:** Database-enforced policies as second defense layer
- **API Authentication:** Tenant context embedded in JWT, validated on every request
- **Cross-Tenant Prevention:** Voter system to verify resource ownership before access

Schema Evolution

Use Doctrine migrations with tenant-aware execution; implement blue-green deployment for zerodowntime updates.

4. How would you architect a distributed job queue system in Symfony for processing millions of background tasks with priority handling and retry logic?

Architecture Design

A robust job queue requires **reliable message delivery**, **horizontal scalability**, and **observability**.

Core Components

- **Message Broker:** RabbitMQ for complex routing or Amazon SQS for managed solution
- **Symfony Messenger:** Central abstraction layer with multiple transports
- **Worker Pools:** Multiple consumer processes per queue with configurable concurrency
- **Dead Letter Queue:** Failed messages routed to DLQ after max retries

Priority Queue Implementation

- **Multiple Queues:** Separate queues for high/medium/low priority
- **Worker Distribution:** More workers assigned to high-priority queues
- **Dynamic Routing:** Route messages based on message properties

```
// Message Handler with Retry
#[AsMessageHandler]
class ProcessImageHandler {
  public function __invoke(ProcessImage $msg) {
    try {
      $this->processor->process($msg->getPath());
    } catch (\Exception $e) {
      throw new UnrecoverableException($e->getMessage());
    }
  }
}
```

Scalability Patterns

- **Horizontal Scaling:** Kubernetes jobs with queue-based autoscaling (KEDA)
- **Partitioning:** Shard queues by tenant or workload type
- **Backpressure:** Prefetch limits and consumer acknowledgment to prevent overload
- **Circuit Breaker:** Pause consumption when downstream services fail

Reliability Features

- **Idempotency:** Store processed message IDs in Redis with TTL
- **Retry Strategy:** Exponential backoff with jitter (1s, 2s, 4s, 8s)
- **Monitoring:** Track queue depth, processing rate, error rate per queue

5. Design a high-performance API gateway in Symfony that handles authentication, rate limiting, request routing, and response caching for a microservices architecture.

API Gateway Responsibilities

The gateway serves as the **single entry point** for all client requests, handling cross-cutting concerns.

Core Features

- **Authentication:** JWT validation with public key verification; support OAuth2 and API keys
- **Authorization:** Role-based and attribute-based access control using Symfony Security
- **Rate Limiting:** Token bucket algorithm with Redis backend, per-user and per-endpoint limits
- **Request Routing:** Dynamic routing to microservices based on URL patterns
- **Response Caching:** HTTP cache with Varnish or Symfony HTTP Cache

Symfony Implementation

```
// Rate Limiter Configuration
```

```
class ApiRateLimiter {
  public function check(Request $req): void {
    $limiter = $this->factory->create(
      $req->getClientIp(),
      new TokenBucketLimiter(100, new Rate(60))
    );
    $limiter->consume(1)->ensureAccepted();
  }
}
```

Performance Optimization

- **Connection Pooling:** HTTP client with persistent connections to backend services
- **Circuit Breaker:** Fail fast when backend services are down, return cached or default responses
- **Response Streaming:** Stream large responses without buffering entire payload
- **Async Processing:** Use Symfony HTTP Client async mode for parallel service calls

Scalability Considerations

- **Stateless Design:** No session storage; all state in JWT or distributed cache
- **Horizontal Scaling:** Deploy multiple gateway instances behind L7 load balancer
- **Cache Strategy:** Multi-tier caching (local APCu, distributed Redis, CDN)
- **Service Discovery:** Integrate with Consul or Kubernetes DNS for dynamic routing

Observability

Implement distributed tracing with correlation IDs, structured logging, and metrics for latency, error rates, and throughput per endpoint.

6. How would you design a social media feed system in Symfony that efficiently handles millions of posts and provides personalized, real-time updates?

Feed Architecture Patterns

Two primary approaches exist: **fan-out on write** (push) and **fan-out on read** (pull).

Hybrid Approach

- **Fan-out on Write:** For users with few followers (< 10k), pre-compute feeds when post is created
- **Fan-out on Read:** For celebrities with millions of followers, compute feed on-demand
- **Hybrid Strategy:** Combine both based on follower count threshold

Data Storage Design

- **Posts Storage:** PostgreSQL with partitioning by created_at for efficient archival
- **Feed Cache:** Redis sorted sets with post IDs and timestamps as scores
- **Graph Database:** Neo4j or adjacency list in PostgreSQL for follower relationships
- **Timeline Storage:** Cassandra for write-heavy feed storage with user_id as partition key

```
// Feed Generator Service
class FeedGenerator {
  public function generate(User $user, int $limit) {
    $cached = $this->redis->zrevrange(
      "feed:{$user->getId()}", 0, $limit
    );
    if (count($cached) >= $limit) return $cached;
    return $this->buildFromFollowing($user, $limit);
  }
}
```

Real-Time Updates

- **WebSocket Integration:** Push new posts to online followers via WebSocket
- **Message Queue:** Symfony Messenger to async distribute posts to follower feeds
- **Event Sourcing:** Store post creation events, rebuild feeds from event stream

Performance Optimizations

- **Pagination:** Cursor-based pagination using last_seen_id for consistent results
- **Feed Trimming:** Keep only recent N posts in cache, archive older posts
- **Lazy Loading:** Load post details only when scrolling, initially show just IDs
- **CDN Caching:** Cache rendered posts at edge locations for global users

7. Design a distributed session management system for a Symfony application deployed across multiple data centers with active-active configuration.

Session Storage Challenges

Multi-datacenter deployments require **low-latency access**, **consistency**, and **failover capabilities**.

Architecture Options

- **Centralized Redis Cluster:** Single Redis cluster with cross-DC replication (high latency)
- **Regional Redis with Sync:** Redis instance per DC with async replication (eventual consistency)
- **Sticky Sessions with Replication:** Route users to primary DC, replicate sessions to backup
- **JWT Stateless Sessions:** Eliminate server-side storage entirely (recommended)

Recommended Approach: Hybrid

Use **JWT for authentication state** and **Redis for mutable session data** with multi-region replication.

```
// Session Handler Configuration
class MultiRegionSessionHandler {
  public function write($id, $data) {
    $primary = $this->primaryRedis->set("sess:$id", $data);
    $this->bus->dispatch(
      new ReplicateSession($id, $data, $this->region)
    );
    return $primary;
  }
}
```

Consistency Model

- **CAP Theorem Trade-off:** Choose availability over consistency (AP system)
- **Conflict Resolution:** Last-write-wins with vector clocks for conflict detection
- **Session Versioning:** Include version number in session data to detect stale reads

Failover Strategy

- **Health Checks:** Monitor Redis availability per region with automatic failover
- **Read Preference:** Read from local DC, write to all DCs asynchronously
- **Session Migration:** On DC failure, redirect users and rebuild session from replica

Security Considerations

- **Encryption:** Encrypt session data at rest and in transit between DCs
- **Session Fixation:** Regenerate session ID after authentication
- **Timeout Strategy:** Sliding expiration with absolute maximum lifetime

8. How would you architect a search system in Symfony for an e-commerce platform with autocomplete, faceted search, and relevance ranking?

Search Architecture

Modern search requires a **dedicated search engine** optimized for full-text queries and aggregations.

Technology Stack

- **Search Engine:** Elasticsearch or OpenSearch for distributed search and analytics
- **Symfony Integration:** FOSElasticaBundle or custom integration with Elasticsearch PHP client
- **Data Sync:** Symfony Messenger for async indexing, Doctrine listeners for change detection
- **Autocomplete:** Edge n-gram tokenizer with completion suggester

Index Design

- **Document Structure:** Denormalized product documents with nested categories and attributes
- **Mapping Strategy:** Custom analyzers for different languages, keyword fields for facets
- **Sharding:** Shard by product category or hash for write distribution
- **Replicas:** Multiple replicas for read scalability and high availability

```
// Search Service
class ProductSearchService {
  public function search(SearchQuery $query) {
    $params = [
      'index' => 'products',
      'body' => [
        'query' => ['multi_match' => [
          'query' => $query->getTerm(),
          'fields' => ['name^3', 'description']
        ]]
      ]
    ];
    return $this->client->search($params);
  }
}
```

Relevance Ranking

- **Scoring Factors:** Text relevance, popularity (sales), recency, user personalization
- **Function Score:** Boost documents based on business rules (margin, inventory)
- **Learning to Rank:** ML model trained on click-through data for personalized ranking
- **A/B Testing:** Experiment with different ranking algorithms

Performance Optimization

- **Caching:** Cache popular queries in Redis with short TTL
- **Query DSL:** Use filters instead of queries for facets (cacheable)
- **Pagination:** Use search_after for deep pagination instead of from/size
- **Async Indexing:** Batch updates every few minutes, not real-time for every change

9. Design a file storage and CDN architecture for a Symfony application that handles user-uploaded media with image processing, virus scanning, and global distribution.

Storage Architecture

Separate **storage**, **processing**, and **delivery** concerns for scalability and reliability.

Components

- **Object Storage:** AWS S3, Google Cloud Storage, or MinIO for file persistence
- **CDN:** CloudFront, Cloudflare, or Fastly for edge caching and delivery
- **Processing Queue:** Symfony Messenger with workers for async image processing
- **Virus Scanner:** ClamAV integration or AWS S3 Malware Protection

Upload Flow

- **Direct Upload:** Client uploads directly to S3 using pre-signed URLs (bypass app server)
- **Validation:** Client-side validation for file type and size before upload
- **Webhook:** S3 event triggers Lambda/webhook to Symfony for post-processing
- **Quarantine:** Store in quarantine bucket until virus scan completes

```
// Pre-signed URL Generator
class UploadService {
  public function generateUploadUrl(string $key) {
    $cmd = $this->s3Client->getCommand('PutObject', [
      'Bucket' => 'uploads',
      'Key' => $key
    ]);
    return $this->s3Client->createPresignedRequest(
      $cmd, '+15 minutes'
    )->getUri();
  }
}
```

Image Processing Pipeline

- **Async Processing:** Generate thumbnails, WebP variants, and optimized versions in background
- **Responsive Images:** Create multiple sizes for different devices and resolutions
- **Format Conversion:** Convert HEIC to JPEG, generate WebP for modern browsers
- **Metadata Extraction:** Extract EXIF data, detect faces for smart cropping

CDN Strategy

- **Cache Headers:** Set long cache TTL (1 year) with versioned URLs for immutability
- **Origin Shield:** Additional caching layer between CDN and S3 to reduce origin load
- **Image Optimization:** CDN-level optimization (Cloudflare Polish, CloudFront image processing)
- **Geo-Routing:** Route requests to nearest regional bucket for lower latency

Security

- **Access Control:** Private S3 buckets with signed URLs for authorized access
- **CORS Configuration:** Restrict origins that can upload directly
- **Content Security:** Serve user content from separate domain to prevent XSS

10. How would you design an event-driven architecture in Symfony for a complex domain with eventual consistency, saga patterns, and event sourcing?

Event-Driven Architecture Principles

Decouple services through **asynchronous events** while maintaining **data consistency** across boundaries.

Core Patterns

- **Event Sourcing:** Store state changes as sequence of events, rebuild state by replaying
- **CQRS:** Separate read and write models, optimize each independently
- **Saga Pattern:** Coordinate distributed transactions through choreography or orchestration
- **Event Store:** Append-only log of domain events (EventStoreDB or custom)

Symfony Implementation

- **Domain Events:** Dispatch events from aggregate roots using Symfony EventDispatcher
- **Event Bus:** Messenger component with multiple transports (sync for queries, async for commands)
- **Projections:** Event handlers that build read models from event stream
- **Snapshots:** Periodic state snapshots to avoid replaying entire event history

```
// Event Sourced Aggregate
class Order {
  private array $events = [];
  public function place(OrderId $id, Items $items) {
    $this->apply(new OrderPlaced($id, $items));
  }
  private function apply(DomainEvent $event) {
    $this->events[] = $event;
    $this->{'apply'.class_basename($event)}($event);
  }
}
```

Saga Implementation

- **Choreography:** Services react to events and publish new events (decentralized)
- **Orchestration:** Central coordinator (saga manager) directs the workflow

- **Compensation:** Implement compensating transactions for rollback (OrderCancelled event)
- **Idempotency:** Store processed event IDs to handle duplicate deliveries

Consistency Guarantees

- **Eventual Consistency:** Accept temporary inconsistency between services
- **Outbox Pattern:** Store events in database with business data, publish atomically
- **Event Versioning:** Support multiple event versions for backward compatibility
- **Schema Registry:** Centralized event schema management and validation

Operational Concerns

- **Monitoring:** Track event processing lag, failure rates, and replay times
- **Debugging:** Event store provides complete audit trail for troubleshooting
- **Event Replay:** Ability to rebuild projections from scratch or specific point in time

Coding and Debugging

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

1. Write a Symfony service that flattens a nested array recursively. How would you register it?

Flattening a Nested Array

Create a service that recursively flattens arrays:

```
namespace App\Service;

class ArrayFlattener
{
    public function flatten(array $array): array
    {
        $result = [];
        array_walk_recursive($array, function($value) use (&$result) {
            $result[] = $value;
        });
        return $result;
    }
}
```

Registration: With autowiring enabled, Symfony automatically registers this as a service. You can also explicitly define it in **services.yaml**:

```
services:
    App\Service\ArrayFlattener:
        autowire: true
        autoconfigure: true
```

Inject it into controllers or other services via constructor injection.

2. How do you implement a custom exception handler in Symfony to catch and log specific exceptions?

Custom Exception Handling

Create an **event subscriber** that listens to the **kernel.exception** event:

```
namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\ExceptionEvent;
use Symfony\Component\HttpKernel\KernelEvents;
use Psr\Log\LoggerInterface;

class ExceptionSubscriber implements EventSubscriberInterface
{
    public function __construct(private LoggerInterface $logger) {}

    public function onKernelException(ExceptionEvent $event): void
    {
        $exception = $event->getThrowable();
        $this->logger->error($exception->getMessage());
    }

    public static function getSubscribedEvents(): array
    {
        return [KernelEvents::EXCEPTION => 'onKernelException'];
    }
}
```

This subscriber automatically logs all exceptions. You can add conditional logic to handle specific exception types differently.

3. Write a Symfony command that reverses a string input and explain how to test it.

String Reversal Command

```
namespace App\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class ReverseStringCommand extends Command
{
    protected static $defaultName = 'app:reverse-string';

    protected function configure(): void
    {
        $this->addArgument('text', InputArgument::REQUIRED);
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $reversed = strrev($input->getArgument('text'));
        $output->writeln($reversed);
        return Command::SUCCESS;
    }
}
```

Testing: Use **CommandTester** to test commands in isolation without running the actual console.

4. How would you check if a string is a palindrome in a Symfony service and cache the result?

Palindrome Checker with Caching

```
namespace App\Service;

use Symfony\Contracts\Cache\CacheInterface;
use Symfony\Contracts\Cache\ItemInterface;

class PalindromeChecker
{
    public function __construct(private CacheInterface $cache) {}

    public function isPalindrome(string $text): bool
    {
        return $this->cache->get(
            'palindrome_' . md5($text),
            function (ItemInterface $item) use ($text) {
                $item->expiresAfter(3600);
                $clean = strtolower(preg_replace('/[^a-z0-9]/i', '', $text));
                return $clean === strrev($clean);
            }
        );
    }
}
```

This service checks for palindromes and **caches results for 1 hour** to avoid redundant computations. The cache key is based on the MD5 hash of the input.

5. What debugging tools does Symfony provide, and how do you use the Profiler to diagnose performance issues?

Symfony Debugging Tools

Key tools:

- **Symfony Profiler:** Web debug toolbar showing request/response data, database queries, cache hits, and performance metrics
- **VarDumper:** Enhanced var_dump() with dump() and dd() functions
- **Stopwatch Component:** Measures execution time of code blocks
- **MonologBundle:** Advanced logging with channels and handlers

Using the Profiler for performance:

- Access via the toolbar at the bottom of dev pages
- Check the **Performance** panel for execution time breakdown
- Review **Database queries** for N+1 problems and slow queries
- Examine **Cache** panel for hit/miss ratios
- Use **Timeline** view to identify bottlenecks

Enable profiler only in dev environment for security.

6. How do you profile memory usage in Symfony applications? Provide a code example using the Stopwatch component.

Memory Profiling

Use the **Stopwatch component** to measure both time and memory:

```
namespace App\Service;

use Symfony\Component\Stopwatch\Stopwatch;

class DataProcessor
{
    public function __construct(private Stopwatch $stopwatch) {}

    public function processLargeDataset(array $data): array
    {
        $this->stopwatch->start('data_processing');

        $result = array_map(fn($item) => $item * 2, $data);

        $event = $this->stopwatch->stop('data_processing');

        echo 'Duration: ' . $event->getDuration() . 'ms';
        echo 'Memory: ' . $event->getMemory() . ' bytes';

        return $result;
    }
}
```

Additionally, use **memory_get_peak_usage()** and enable **Blackfire** or **Tideways** for production profiling.

7. Explain how to implement custom error pages in Symfony for different HTTP status codes.

Custom Error Pages

Symfony uses the **TwigBundle** to render error pages. Create templates in **templates/bundles/TwigBundle/Exception/**:

- **error404.html.twig** - for 404 errors
- **error403.html.twig** - for 403 errors
- **error500.html.twig** - for 500 errors
- **error.html.twig** - fallback for all errors

Example error404.html.twig:

```
{% extends 'base.html.twig' %}
{% block body %}
    <h1>Page Not Found</h1>
    <p>The page you are looking for does not exist.</p>
    <a href="{{ path('homepage') }}">Go to homepage</a>
{% endblock %}
```

To preview error pages in dev mode, use: **/_error/404** or **/_error/500** routes. In production, these templates render automatically.

8. How do you debug Doctrine queries and identify N+1 query problems in Symfony?

Debugging Doctrine Queries

Methods to identify N+1 problems:

- **Symfony Profiler:** Check the Doctrine panel showing all executed queries with execution time
- **Enable SQL logging:** View queries in dev.log
- **DoctrineBundle configuration:** Set **logging: true** and **profiling: true**

Solutions for N+1 issues:

```
// Bad: N+1 queries
$users = $userRepository->findAll();
foreach ($users as $user) {
    echo $user->getProfile()->getBio(); // Extra query per user
}

// Good: Single query with JOIN
$users = $userRepository->createQueryBuilder('u')
    ->leftJoin('u.profile', 'p')
    ->addSelect('p')
    ->getQuery()
    ->getResult();
```

Use **fetch joins** or **Doctrine's EXTRA_LAZY** fetch mode for collections.

9. What is the equivalent of monkey patching in Symfony, and how can you override core framework behavior?

Overriding Framework Behavior

Symfony doesn't support traditional monkey patching, but provides structured approaches:

- **Service Decoration:** Wrap existing services to modify behavior
- **Compiler Passes:** Modify service definitions during container compilation
- **Event Listeners/Subscribers:** Hook into framework lifecycle
- **Bundle Extension:** Override bundle configuration

Service Decoration Example:

```
services:
    App\Service\CustomMailer:
        decorates: Symfony\Component\Mailer\MailerInterface
        arguments: ['@.inner']

class CustomMailer implements MailerInterface
{
    public function __construct(private MailerInterface $mailer) {}

    public function send(RawMessage $message): void
    {
        // Custom logic before/after
        $this->mailer->send($message);
    }
}
```

This maintains type safety and testability unlike monkey patching.

10. How do you implement and test a custom Twig extension that formats dates relative to now (e.g., '2 hours ago')?

Custom Twig Extension

```
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class DateExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [new TwigFilter('time_ago', [$this, 'timeAgo'])];
    }

    public function timeAgo(\DateTimeInterface $date): string
    {
        $diff = (new \DateTime())->diff($date);
        if ($diff->y > 0) return $diff->y . ' years ago';
        if ($diff->m > 0) return $diff->m . ' months ago';
        if ($diff->d > 0) return $diff->d . ' days ago';
        if ($diff->h > 0) return $diff->h . ' hours ago';
        return $diff->i . ' minutes ago';
    }
}
```

Usage in templates: {{ post.createdAt|time_ago }}

Testing: Create unit tests for the timeAgo method with various DateTime inputs to verify correct output.

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Describe a time when you had to optimize a slow Symfony application. What was your approach?

Situation: Our e-commerce platform built with Symfony was experiencing page load times exceeding 3 seconds during peak traffic, affecting conversion rates.

Task: I was tasked with identifying bottlenecks and improving performance to achieve sub-second response times.

Action: I implemented a multi-layered approach: enabled Symfony's HTTP cache and reverse proxy caching with Varnish, optimized Doctrine queries by adding strategic indexes and implementing query result caching, reduced N+1 query problems using joins and batch fetching, and profiled the application using Blackfire to identify memory-intensive operations. I also implemented Redis for session storage and frequently accessed data.

Result: Page load times dropped to under 800ms on average, server load decreased by 40%, and we handled 3x more concurrent users during sales events without infrastructure upgrades.

2. Tell me about a challenging bug you encountered in a Symfony project and how you resolved it.

Situation: We had a production issue where certain API requests were intermittently returning 500 errors with no clear pattern, affecting approximately 5% of requests.

Task: I needed to identify the root cause quickly as it was impacting customer transactions and revenue.

Action: I enabled detailed logging in the production environment temporarily, analyzed Monolog logs and discovered the errors occurred during high concurrency. I identified a race condition in our custom event listener that was modifying shared state. I refactored the listener to be stateless, implemented proper locking mechanisms using Symfony Lock component for critical sections, and added comprehensive unit tests to prevent regression.

Result: The 500 errors were completely eliminated within 24 hours of deployment. I also documented the issue and created coding guidelines for the team regarding stateless service design and concurrency considerations.

3. Describe a situation where you had to refactor legacy Symfony code. What was your strategy?

Situation: I inherited a Symfony 3.4 application with tightly coupled controllers, business logic mixed with presentation layer, and no test coverage.

Task: Modernize the codebase to Symfony 5.x while maintaining business continuity and improving maintainability.

Action: I created a phased migration plan: first, I wrote characterization tests to lock in existing behavior; then systematically extracted business logic into dedicated service classes following SOLID principles; implemented the repository pattern for data access; introduced value objects for domain concepts; configured autowiring and autoconfiguration; and migrated to the latest Symfony version incrementally. I used Rector to automate syntax updates and deprecation fixes.

Result: Successfully migrated to Symfony 5.4 with 85% test coverage, reduced controller code by 60%, and decreased bug reports by 45% in the following quarter. The team's development velocity increased significantly due to better code organization.

4. Share an example of how you handled a disagreement with a team member about Symfony architecture decisions.

Situation: During sprint planning, a senior developer insisted on using Symfony's form component for a complex multi-step API workflow, while I believed a custom command bus pattern would be more appropriate.

Task: Resolve the architectural disagreement while maintaining team cohesion and making the best technical decision.

Action: I proposed a time-boxed proof-of-concept approach where we would each implement a simplified version of our solution. I created a prototype using Symfony Messenger with command handlers, demonstrating better testability, separation of concerns, and flexibility for async processing. We presented both solutions to the team with metrics on complexity, maintainability, and scalability. I actively listened to concerns about learning curve and acknowledged valid points about the form component's validation features, which I incorporated into my solution.

Result: The team agreed on the command bus approach after seeing the concrete comparison. The disagreeing developer became a champion of the pattern after working with it. We documented the decision in an ADR (Architecture Decision Record) for future reference.

5. Describe a time when you had to meet a tight deadline on a Symfony project. How did you manage it?

Situation: A critical payment integration feature needed to be delivered in 2 weeks instead of the planned 4 weeks due to a contractual obligation with a payment provider.

Task: Deliver a production-ready, secure payment integration feature within the compressed timeline without compromising quality.

Action: I immediately broke down the work into MVP and nice-to-have features, focusing on core payment processing first. I leveraged Symfony's existing security components and used well-tested community bundles where appropriate rather than building from scratch. I implemented comprehensive integration tests using the payment provider's sandbox environment, set up automated deployment pipelines to reduce manual overhead, and conducted daily stand-ups to identify blockers early. I also paired programmed with a junior developer to accelerate development and knowledge transfer.

Result: We delivered the MVP on time with all critical security requirements met. The feature processed over $100K in transactions in the first week with zero payment failures. The nice-to-have features were delivered in the following sprint as planned.

6. Tell me about a time you introduced a new Symfony best practice or tool to your team.

Situation: Our team was manually writing repetitive CRUD code and API endpoints, leading to inconsistencies and slower development cycles.

Task: Improve development efficiency and code consistency across the team.

Action: I researched and proposed adopting API Platform, a Symfony-based framework for building APIs. I created a presentation demonstrating how it could auto-generate REST and GraphQL APIs, handle serialization, validation, and documentation automatically. I developed a pilot feature using API Platform and documented the setup process. I then conducted hands-on workshops for the team, created internal documentation with examples, and established code review guidelines to ensure proper usage. I also set up office hours for team members to ask questions.

Result: The team adopted API Platform for new endpoints, reducing API development time by 50%. Code consistency improved significantly, and our API documentation became automatically synchronized with implementation. Three team members later presented our approach at a local Symfony meetup.

7. Describe a situation where you had to debug a complex issue in a Symfony application under pressure.

Situation: During Black Friday, our Symfony-based checkout system started failing for users with shopping carts containing more than 10 items, causing immediate revenue loss.

Task: Identify and fix the critical bug while the system was under maximum load and business stakeholders were monitoring closely.

Action: I stayed calm and methodically approached the problem: first, I replicated the issue in a staging environment; used Symfony's profiler to examine the failing requests; discovered that a recent deployment introduced a serialization issue with nested entities in the cart; identified that the entity graph was hitting PHP's max nesting level during serialization. I implemented a quick fix by adjusting serialization groups and reducing the entity graph depth, deployed the hotfix through our emergency deployment process, and monitored error rates in real-time using our logging infrastructure.

Result: The issue was resolved within 90 minutes of detection. We recovered the checkout functionality and post-incident analysis showed we prevented an estimated $50K in lost sales. I subsequently created monitoring alerts for similar serialization issues and added integration tests for large cart scenarios.

8. Share an experience where you mentored a junior developer on Symfony development.

Situation: A junior developer joined our team with basic PHP knowledge but no Symfony experience and was struggling with dependency injection and service configuration concepts.

Task: Bring the junior developer up to speed so they could contribute effectively to our Symfony projects within one month.

Action: I created a structured mentoring plan with weekly goals, starting with Symfony fundamentals. I assigned progressively complex tasks: first, simple controller modifications; then creating new services with dependency injection; followed by working with Doctrine entities and repositories. I conducted daily code reviews with detailed feedback, pair programmed on challenging features, and encouraged questions in a safe environment. I also shared curated resources including Symfony documentation, SymfonyCasts tutorials, and relevant blog posts. I created internal documentation specifically addressing common pitfalls we encountered in our codebase.

Result: Within 6 weeks, the junior developer was confidently implementing features independently and even identified and fixed a bug in our authentication system. They later became a resource for other new team members, and the documentation I created became part of our official onboarding process.

9. Describe a time when you had to make a trade-off between code quality and delivery speed in a Symfony project.

Situation: We needed to implement a reporting feature for a client demo in 3 days, but implementing it with proper domain modeling, caching, and optimization would take at least a week.

Task: Deliver a working demo while ensuring we wouldn't create significant technical debt or compromise system stability.

Action: I proposed a pragmatic approach to stakeholders: implement a simplified version using Symfony's query builder for direct database queries instead of full DDD approach, ensure proper input validation and security measures were not compromised, clearly document the technical debt with TODO comments and tickets, and commit to refactoring after demo approval. I created the feature with clean separation so it could be replaced without affecting other components, wrote integration tests to lock in behavior, and set up a dedicated namespace to isolate the temporary implementation.

Result: We delivered the demo on time, which secured client approval. I allocated time in the next two sprints to properly refactor the feature, implementing proper domain models, caching with Redis, and query optimization. The final version performed 10x faster than the demo version, and stakeholders appreciated the transparent communication about trade-offs.

10. Tell me about a time you improved the security of a Symfony application.

Situation: During a security audit, we discovered our Symfony application had several vulnerabilities including inadequate CSRF protection, missing rate limiting on authentication endpoints, and improper handling of user permissions.

Task: Address all critical and high-severity security issues within two weeks before the audit report deadline.

Action: I prioritized issues by severity and impact. I enabled and properly configured Symfony's CSRF protection across all forms, implemented rate limiting using Symfony's Rate Limiter component for login and API endpoints, refactored our security voters to properly check permissions at the entity level, added security headers using the SecurityBundle configuration, implemented proper password hashing using Symfony's PasswordHasher, and conducted a comprehensive review of all user input handling to prevent injection attacks. I also set up automated security scanning in our CI/CD pipeline using tools like PHPStan and Psalm with security rulesets.

Result: All critical and high-severity issues were resolved before the deadline. The follow-up audit showed zero critical vulnerabilities. I documented the security improvements and created a security checklist for code reviews, which became part of our development process and prevented similar issues in future features.