Complete Technische Documentatie - Django Backend Project (van products en services)

Complete Technische Documentatie - Django Backend Project

1. Overzicht van Technologieën

Core Technologieën

- **Python 3.8+**: Hoofdprogrammeertaal
- **Django 4.2+**: Web framework voor snelle ontwikkeling
- **Django REST Framework**: API development framework
- **MongoDB**: NoSQL database via Djongo
- **PostgreSQL**: Relationele database backup (optioneel)

Database Layer

- **Djongo 1.3.6+**: MongoDB connector voor Django
- **pymongo 4.3+**: Native MongoDB driver
- **psycopg2-binary**: PostgreSQL adapter
- **Django ORM**: Object-Relational Mapper

API & Serialization

- **djangorestframework 3.14+**: REST API framework
- **djangorestframework-simplejwt**: JWT authenticatie

- **django-cors-headers**: Cross-Origin Resource Sharing
- **django-filter**: Geavanceerde filtering

Security & Authentication

- **JWT (JSON Web Tokens)**: Stateless authenticatie
- **bcrypt**: Password hashing (via Django)
- **python-decouple**: Environment variable management
- **django-csp**: Content Security Policy

File Handling & Media

- **Pillow 10.0+**: Image processing
- **python-magic**: File type detection
- **Whitenoise 6.5+**: Static file serving

Development & Testing

- **pytest**: Testing framework
- **django-debug-toolbar**: Development debugging
- **django-extensions**: Extra development tools
- **Faker**: Test data generation

Deployment & Production

- **Gunicorn 21.2+**: WSGI HTTP server
- **uvicorn**: ASGI server (voor async)
- **Docker & Docker Compose**: Containerization
- **NGINX**: Reverse proxy & load balancing

2. Gedetailleerde Bibliotheek Uitleg

Django & Core Dependencies

Django (django)

```
# Primair web framework
# Functies: MVC architectuur, ORM, admin interface, template engine
# Gebruik: Complete web applicatie structuur
INSTALLED_APPS = [
    'django.contrib.admin',      # Admin interface
    'django.contrib.auth',       # Authenticatie systeem
    'django.contrib.contenttypes',  # Content type framework
    'django.contrib.sessions',   # Session management
    'django.contrib.messages',   # Message framework
    'django.contrib.staticfiles', # Static file handling
]
```

Django REST Framework (djangorestframework)

```
# REST API framework voor Django
# Functies: Serializers, ViewSets, Authentication, Permissions
# Gebruik: API endpoint creatie en management
from rest_framework import serializers, viewsets, permissions

# Serializers: Data validation en transformation
# ViewSets: CRUD operations voor modellen
# Permissions: Access control mechanismen
```

Djongo (djongo)

```
# MongoDB connector voor Django ORM
# Functies: Django ORM met MongoDB backend
# Gebruik: NoSQL database integratie
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'company_services',
        'CLIENT': {
            'host': 'localhost',
            'port': 27017,
        }
    }
}
# Converteert Django ORM queries naar MongoDB queries
# Ondersteunt Django migrations met MongoDB
```

Django Simple JWT (djangorestframework-simplejwt)

```
# JWT implementatie voor Django REST Framework
# Functies: Token creation, validation, refresh
# Gebruik: Stateless user authenticatie
from rest_framework_simplejwt.authentication import JWTAuthentication
from rest_framework_simplejwt.tokens import RefreshToken

# Token types:
# - Access Token: Kort leven (60 min)
# - Refresh Token: Langer leven (1 dag)
# - Blacklist: Token invalidatie
```

Django CORS Headers (corsheaders)

```
# Cross-Origin Resource Sharing handling
# Functies: Cross-domain request toegang
# Gebruik: Frontend-backend communicatie
CORS_ALLOWED_ORIGINS = [
    "<http://localhost:3000>",
    "<http://127.0.0.1:3000>",
]
# Voegt CORS headers toe aan responses
# Beveiligt tegen ongeautoriseerde cross-domain requests
```

Django Filter (django-filter)

```
# Geavanceerde filtering voor QuerySets
# Functies: Dynamic filter creation, complex queries
# Gebruik: API filtering en searching
from django_filters.rest_framework import DjangoFilterBackend

class ProductFilter(django_filters.FilterSet):
    search = django_filters.CharFilter(method='filter_search')
    min_price = django_filters.NumberFilter(field_name='price', lookup_expr='g
te')
```

Python Decouple (python-decouple)

```
# Environment variable management
# Functies: Config scheiding, security best practices
# Gebruik: Gevoelige data afscherming
from decouple import config

SECRET_KEY = config('SECRET_KEY')
DEBUG = config('DEBUG', default=False, cast=bool)
# Scheidt configuratie van code
# Ondersteunt .env bestanden voor development
```

Pillow (PIL)

```
# Image processing library
# Functies: Image manipulation, validation, conversion
# Gebruik: File upload handling
from PIL import Image

# Ondersteunt:
# - Image validation
# - Thumbnail generation
# - Format conversion
# - Metadata extraction
```

3. Models - Gedetailleerde Uitleg

Core Models (core/models.py)

CustomUser Model

```
class CustomUser(AbstractBaseUser, PermissionsMixin):
    """
    Aangepast user model met email als primary identifier.

    TECHNISCHE CONCEPTEN:
    - AbstractBaseUser: Django's base user class
    - PermissionsMixin: Permission system integratie
    - USERNAME_FIELD: Primary login field
    - CustomUserManager: Custom manager voor user creation

    VELDEN:
    - email: Primary identifier (uniek)
    - first_name/last_name: User identification
    - phone: Contact informatie
    - is_staff: Admin toegang
    - is_active: Account status
    - date_joined: Registratie timestamp

    METHODEN:
    - create_user(): Standard user creation
    - create_superuser(): Admin user creation
    - get_full_name(): Formatteerde naam
    """
```

SiteConfig Model

```
class SiteConfig(models.Model):
    """
    Centrale configuratie voor de website.

    TECHNISCHE CONCEPTEN:
    - Singleton pattern (één config instantie)
    - JSON serializable voor caching
    - Dynamic settings management

    VELDEN:
    - Company info: naam, email, telefoon, adres
    - Social media: platform URLs
    - SEO: meta tags voor search engines
    - Configuration: onderhoudsmodus, registratie
    """
```

Product Models (products/models.py)

ProductCategory Model

```
class ProductCategory(models.Model):
    """
    Hiërarchische categorie structuur voor producten.

    TECHNISCHE CONCEPTEN:
    - Self-referential ForeignKey (parent)
    - Recursive relationships
    - Slug field voor SEO URLs
    - Tree structure traversal

    METHODEN:
    - get_descendants(): Recursive child retrieval
    - get_all_products_count(): Aggregated product counting
    """
```

Product Model

```
class Product(models.Model):
    """
    Centraal product model voor verkoop items.

    TECHNISCHE CONCEPTEN:
    - Choice fields met constraints
    - Decimal fields voor prijzen
    - ManyToMany relationships (categories)
    - Property methods voor berekeningen
    - Database indexes voor performance

    PROPERTIES:
    - is_active: Dynamic status check
    - final_price: Price calculation
    - discount_percentage: Discount calculation

    BUSINESS LOGIC:
    - increment_views(): View counter
    - decrease_stock(): Inventory management
    - is_low_stock(): Inventory alert
    """
```

ProductImage Model

```
class ProductImage(models.Model):
    """
    Image management voor product galleries.

    TECHNISCHE CONCEPTEN:
    - ForeignKey relationship naar Product
    - ImageField met upload_to path
    - Ordering via display_order
    - Primary image flag
    """
```

ProductFeature Model

```
class ProductFeature(models.Model):
    """
    Specificaties en kenmerken van producten.

    TECHNISCHE CONCEPTEN:
    - Key-value pair storage
    - Icon field voor UI representatie
    - Ordering voor consistent display
    """
```

ProductReview Model

```
class ProductReview(models.Model):
    """
    Klantbeoordelingen en ratings systeem.

    TECHNISCHE CONCEPTEN:
    - Rating validation (1-5)
    - Verified purchase tracking
    - Helpful voting system
    - Moderation system (is_approved)

    PROPERTIES:
    - helpful_score: Percentage calculation
    """
```

ProductView Model

```
class ProductView(models.Model):
    """
    Analytics tracking voor product views.

    TECHNISCHE CONCEPTEN:
    - User session tracking
    - IP address logging
    - Referrer tracking
    - Time-based analytics
    """
```

Service Models (services/models.py)

ServiceCategory Model

```
class ServiceCategory(models.Model):
    """
    Dienst categorieën volgens business requirements.

    TECHNISCHE CONCEPTEN:
    - Choice field voor vaste categorieën
    - Icon field voor UI
    - Homepage display flag
    - SEO metadata fields

    CATEGORY_TYPES:
    - demontage_montage: Tools icon
    - mobel_verkauf: Couch icon
    - auto_ankauf: Car icon
    - renovierung: Hammer icon
    - entsorgung: Trash icon
    - transport: Truck icon
    - import_export: Globe icon
    """
```

Service Model

```
class Service(models.Model):
    """
    Individuele diensten binnen categorieën.

    TECHNISCHE CONCEPTEN:
    - Price flexibility (fixed/quote)
    - Service requirements tracking
    - Online booking capability
    - Emergency service flag
    - View and quote statistics

    BUSINESS LOGIC:
    - increment_views(): Popularity tracking
    - increment_quote_requests(): Lead generation tracking
    """
```

ServiceImage Model

```
class ServiceImage(models.Model):
    """
    Before/after images voor portfolio.

    TECHNISCHE CONCEPTEN:
    - Boolean flags voor before/after
    - Portfolio gallery management
    - Ordering voor slideshows
    """
```

FAQ Model

```
class FAQ(models.Model):
    """
    Veelgestelde vragen per dienst.
    TECHNISCHE CONCEPTEN:
    - Question-Answer pairs
    - Display ordering
    - Active/inactive toggle
    """
```

ServiceFeature Model

```
class ServiceFeature(models.Model):
    """
    Kenmerken en voordelen van diensten.

    TECHNISCHE CONCEPTEN:
    - Icon-based feature display
    - Descriptive feature lists
    - Ordering voor UI consistency
    """
```

ServicePackage Model

```
class ServicePackage(models.Model):
    """
    Bundels en pakketten voor diensten.

    TECHNISCHE CONCEPTEN:
    - Package pricing
    - Inclusion/exclusion lists
    - Duration specification
    - Popular package flag
    """
```

ServiceArea Model

```
class ServiceArea(models.Model):
    """
    Geografische beschikbaarheid van diensten.

    TECHNISCHE CONCEPTEN:
    - City-based service areas
    - Postal code granularity
    - Region grouping
    - Active/inactive areas
    """
```

Testimonial Model

```
class Testimonial(models.Model):
    """
    Klantbeoordelingen en referenties.

    TECHNISCHE CONCEPTEN:
    - Star rating system
    - Client information storage
    - Project reference linking
    - Featured testimonials
    - Moderation workflow
    """
```

4. Views & ViewSets - Gedetailleerde Uitleg

Product Views (products/views.py)

ProductCategoryViewSet Class

```
class ProductCategoryViewSet(viewsets.ModelViewSet):
    """
    CRUD operations voor product categorieën.
    TECHNISCHE CONCEPTEN:
    - ModelViewSet: Complete CRUD via ViewSet
    - Lookup_field: Slug-based retrieval
    - Permission classes: Role-based access
    - Custom actions: Extended functionality

    METHODS:
    - get_queryset(): Dynamic filtering
    - products(): Custom action voor categorie producten
    """

    def get_queryset(self):
        """
        Filter queryset op basis van query parameters.

        TECHNISCHE CONCEPTEN:
        - Query parameter parsing
        - Dynamic filtering
        - Hierarchical category queries

        PARAMETERS:
        - parent: Filter op parent categorie
        - root_only: Alleen root categorieën
        """
```

ProductViewSet Class

```
class ProductViewSet(viewsets.ModelViewSet):
    """
    Complete product management ViewSet.

    TECHNISCHE CONCEPTEN:
    - Multiple serializer classes
    - Filter backends integration
    - Search functionality
    - View tracking

    CUSTOM ACTIONS:
    - increment_view(): Manual view tracking
    - similar(): Recommendation engine
    - featured(): Featured products
    - bestsellers(): Top products
    - on_sale(): Discounted products
    """

    def retrieve(self, request, *args, **kwargs):
        """
        Override voor view tracking.

        TECHNISCHE CONCEPTEN:
        - Method overriding
        - Analytics integration
        - IP address tracking
        - User session logging
        """

    def _log_product_view(self, product, request):
        """
        Log product view voor analytics.

        TECHNISCHE CONCEPTEN:
        - Request metadata extraction
        - Session management
        - IP address detection
        - Error handling met logging
        """

    def _get_client_ip(self, request):
        """
        Extract client IP address.
```

```
        TECHNISCHE CONCEPTEN:
        - HTTP header parsing
        - Proxy detection (X-Forwarded-For)
        - Multiple IP handling
        """
```

ProductSearchView Class

```
class ProductSearchView(generics.ListAPIView):
    """
    Geavanceerde zoekfunctionaliteit.

    TECHNISCHE CONCEPTEN:
    - Generic API View
    - Search parameter validation
    - Complex query construction
    - Pagination integration

    SEARCH PARAMETERS:
    - q: Text search
    - category: Categorie filter
    - min/max_price: Price range
    - condition: Product condition
    - brand/material/color: Attribute filters
    - sort_by: Sorting options
    """

    def get_queryset(self):
        """
        Build dynamic queryset op basis van search parameters.

        TECHNISCHE CONCEPTEN:
        - Q objects voor complex queries
        - Annotate voor calculated fields
        - Distinct voor duplicate prevention
        - Chainable filters
        """
```

ProductReviewViewSet Class

```
class ProductReviewViewSet(viewsets.ModelViewSet):
    """
    Review management systeem.

    TECHNISCHE CONCEPTEN:
    - User ownership validation
    - Moderation workflow
    - Helpful voting system
    - Report functionality

    CUSTOM ACTIONS:
    - mark_helpful(): Voting system
    - report(): Content reporting
    """
```

ProductStatisticsView Class

```
class ProductStatisticsView(generics.GenericAPIView):
    """
    Admin statistieken endpoint.

    TECHNISCHE CONCEPTEN:
    - Permission-based access
    - Aggregate calculations
    - Data transformation voor visualisatie
    - Time-based analytics

    STATISTICS:
    - Total products count
    - Stock levels
    - Category distribution
    - Revenue calculations
    """
```

Service Views (services/views.py)

ServiceCategoryViewSet Class

```
class ServiceCategoryViewSet(viewsets.ModelViewSet):
    """
    Dienst categorie management.

    TECHNISCHE CONCEPTEN:
    - Homepage filtering
    - Category type filtering
    - Service counting
    """

    @action(detail=True, methods=['get'])
    def services(self, request, slug=None):
        """
        Haal alle diensten in een categorie op.

        TECHNISCHE CONCEPTEN:
        - Nested resource retrieval
        - Pagination voor grote datasets
        - Active service filtering
        """
```

ServiceViewSet Class

```
class ServiceViewSet(viewsets.ModelViewSet):
    """
    Complete dienst management.
    TECHNISCHE CONCEPTEN:
    - Service area filtering
    - Emergency service flagging
    - Online booking filtering
    - Popularity tracking

    CUSTOM ACTIONS:
    - increment_quote_request(): Lead tracking
    - before_after_images(): Portfolio images
    - homepage_services(): Homepage display
    - popular(): Popular services
    """
```

ServiceSearchView Class

```
class ServiceSearchView(generics.ListAPIView):
    """
    Dienst-specifieke zoekfunctionaliteit.

    TECHNISCHE CONCEPTEN:
    - Service area filtering (city-based)
    - Emergency service search
    - Online booking availability
    - Fixed price filtering
    """
```

TestimonialViewSet Class

```
class TestimonialViewSet(viewsets.ModelViewSet):
    """
    Testimonial management systeem.

    TECHNISCHE CONCEPTEN:
    - Moderation workflow
    - Featured content management
    - Service-based filtering
    - Auto-approval voor staff
    """
```

5. Serializers - Gedetailleerde Uitleg

Product Serializers

ProductCategorySerializer Class

```
class ProductCategorySerializer(serializers.ModelSerializer):
    """
    Serializer voor categorie data.

    TECHNISCHE CONCEPTEN:
    - Nested serialization (subcategories)
    - Computed fields (product_count)
    - Read-only fields voor metadata
    """

    product_count = serializers.SerializerMethodField()
    subcategories = serializers.SerializerMethodField()

    def get_product_count(self, obj):
        """Dynamic product count calculation."""

    def get_subcategories(self, obj):
        """Recursive subcategory serialization."""
```

ProductListSerializer Class

```
class ProductListSerializer(serializers.ModelSerializer):
    """
    Lightweight serializer voor product lists.

    TECHNISCHE CONCEPTEN:
    - Optimized voor list views
    - Computed prices (final_price, discount)
    - Rating aggregations
    - Primary image selection
    """

    primary_image = serializers.SerializerMethodField()
    final_price = serializers.DecimalField(read_only=True)
    discount_percentage = serializers.SerializerMethodField()
    avg_rating = serializers.SerializerMethodField()
    review_count = serializers.SerializerMethodField()
```

ProductDetailSerializer Class

```
class ProductDetailSerializer(ProductListSerializer):
    """
    Complete serializer voor product details.

    TECHNISCHE CONCEPTEN:
    - Inheritance van ProductListSerializer
    - Nested relationships (images, features, reviews)
    - Stock level calculations
    - Service availability flags
    """

    images = ProductImageSerializer(many=True, read_only=True)
    features = ProductFeatureSerializer(many=True, read_only=True)
    reviews = serializers.SerializerMethodField()
    is_low_stock = serializers.BooleanField(read_only=True)
```

ProductReviewSerializer Class

```
class ProductReviewSerializer(serializers.ModelSerializer):
    """
    Review data serialization.

    TECHNISCHE CONCEPTEN:
    - User auto-assignment
    - Rating display transformation
    - Helpful score calculation
    - Write-only fields voor privacy
    """

    def create(self, validated_data):
        """Auto-assign user en set moderation flags."""
```

Service Serializers

ServiceCategorySerializer Class

```
class ServiceCategorySerializer(serializers.ModelSerializer):
    """
    Dienst categorie serialization.

    TECHNISCHE CONCEPTEN:
    - Icon display formatting
    - Service count calculations
    - Category type validation
    """

    service_count = serializers.SerializerMethodField()
    icon_display = serializers.SerializerMethodField()
```

ServiceListSerializer Class

```
class ServiceListSerializer(serializers.ModelSerializer):
    """
    Lightweight service list serializer.

    TECHNISCHE CONCEPTEN:
    - Primary image selection
    - FAQ and testimonial counts
    - Service type flags
    - Price information
    """

    primary_image = serializers.SerializerMethodField()
    faq_count = serializers.SerializerMethodField()
    testimonial_count = serializers.SerializerMethodField()
```

ServiceDetailSerializer Class

```
class ServiceDetailSerializer(ServiceListSerializer):
    """
    Complete service detail serializer.

    TECHNISCHE CONCEPTEN:
    - Nested portfolio images
    - Package and feature lists
    - Service area information
    - Testimonial filtering
    """

    images = ServiceImageSerializer(many=True, read_only=True)
    faqs = FAQSerializer(many=True, read_only=True)
    packages = ServicePackageSerializer(many=True, read_only=True)
    testimonials = serializers.SerializerMethodField()
```

6. Permissions & Authentication

Custom Permission Classes

IsAdminOrReadOnly Class

```
class IsAdminOrReadOnly(permissions.BasePermission):
    """
    Admin-only write access, read access voor iedereen.

    TECHNISCHE CONCEPTEN:
    - HTTP method checking (SAFE_METHODS)
    - User role validation
    - Permission chaining
    """

    def has_permission(self, request, view):
        if request.method in permissions.SAFE_METHODS:
            return True
        return request.user and request.user.is_staff
```

IsOwnerOrReadOnly Class

```
class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Object-level permission checking.

    TECHNISCHE CONCEPTEN:
    - Object ownership validation
    - Attribute existence checking
    - Fallback naar admin access
    """

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True

        if hasattr(obj, 'user'):
            return obj.user == request.user
        elif hasattr(obj, 'created_by'):
            return obj.created_by == request.user

        return request.user and request.user.is_staff
```

JWT Authentication Flow

```
# 1. Token Obtaining
POST /api/auth/token/
{
    "email": "user@example.com",
    "password": "password123"
}

# 2. Token Response
{
    "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1...",
    "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1..."
}

# 3. API Request met Token
GET /api/products/
Headers: Authorization: Bearer <access_token>

# 4. Token Refresh
POST /api/auth/token/refresh/
{
    "refresh": "<refresh_token>"
}
```

7. Filters & Search System

ProductFilter Class

```
class ProductFilter(django_filters.FilterSet):
    """
    Geavanceerde filtering voor producten.

    TECHNISCHE CONCEPTEN:
    - Custom filter methods
    - Lookup expression mapping
    - Boolean filtering
    - Text search integration
    """

    search = django_filters.CharFilter(method='filter_search')
    min_price = django_filters.NumberFilter(field_name='price', lookup_expr='gte')
    category = django_filters.CharFilter(method='filter_category')

    def filter_search(self, queryset, name, value):
        """
        Multi-field text search.

        TECHNISCHE CONCEPTEN:
        - Q object composition
        - Case-insensitive search (icontains)
        - Field-specific searching
        """
        return queryset.filter(
            Q(title__icontains=value) |
            Q(short_description__icontains=value) |
            Q(full_description__icontains=value)
        )
```

ServiceFilter Class

```
class ServiceFilter(django_filters.FilterSet):
    """
    Dienst-specifieke filtering.

    TECHNISCHE CONCEPTEN:
    - Service area filtering
    - Emergency service flags
    - Online booking availability
    - City-based filtering
    """

    city = django_filters.CharFilter(method='filter_city')

    def filter_city(self, queryset, name, value):
        """
        Filter op basis van service areas.

        TECHNISCHE CONCEPTEN:
        - Related model filtering
        - Case-insensitive matching
        - Distinct result ensuring
        """
        return queryset.filter(areas__city__iexact=value)
```

8. Signal Handlers & Business Logic

Product Signals

```
@receiver(pre_save, sender=Product)
def generate_product_slug(sender, instance, **kwargs):
    """
    Auto-generate slug voor producten.
    TECHNISCHE CONCEPTEN:
    - Pre-save signal handling
    - Slug generation met slugify
    - Unique slug ensuring
    - Counter-based fallback
    """

    if not instance.slug:
        instance.slug = slugify(instance.title)

        counter = 1
        original_slug = instance.slug
        while Product.objects.filter(slug=instance.slug).exists():
            instance.slug = f"{original_slug}-{counter}"
            counter += 1
```

Service Signals

```
@receiver(post_save, sender=Testimonial)
def notify_admin_on_testimonial(sender, instance, created, **kwargs):
    """
    Admin notification voor nieuwe testimonials.

    TECHNISCHE CONCEPTEN:
    - Post-save signal handling
    - Conditional logic (created flag)
    - Email notification system
    - Logging integration
    """

    if created and not instance.is_approved:
        # Send email notification to admin
        send_mail(
            'Nieuwe testimonial wacht op goedkeuring',
            f'Testimonial van {instance.client_name} voor {instance.service.nam
e}',
            'noreply@company.com',
            ['admin@company.com']
        )
```

9. Database Design & Optimization

MongoDB Schema Design

```
# Product Document Example
{
    "_id": ObjectId("..."),
    "title": "Moderne Eetkamerstoel",
    "slug": "moderne-eetkamerstoel",
    "categories": [
        {"id": 1, "name": "Meubels", "slug": "meubels"}
    ],
    "price": 149.99,
    "images": [
        {
            "url": "/media/products/2024/01/stoel.jpg",
            "alt_text": "Moderne eetkamerstoel",
            "is_primary": true
        }
    ],
    "features": [
        {"name": "Materiaal", "value": "Hout, Textiel"},
        {"name": "Kleur", "value": "Zwart"}
    ],
    "metadata": {
        "views_count": 156,
        "avg_rating": 4.5,
        "created_at": ISODate("2024-01-15T10:00:00Z")
    }
}
```

Indexing Strategy

```
# Product Model Indexes
class Meta:
    indexes = [
        # Primary lookup index
        models.Index(fields=['slug']),

        # Filtering indexes
        models.Index(fields=['status', 'is_active']),
        models.Index(fields=['price']),

        # Sorting indexes
        models.Index(fields=['created_at']),

        # Search indexes
        models.Index(fields=['title', 'brand']),
    ]
```

Query Optimization

```
def get_optimized_queryset(self):
    """
    Geoptimaliseerde queryset met select_related en prefetch_related.

    TECHNISCHE CONCEPTEN:
    - select_related: ForeignKey relationships
    - prefetch_related: ManyToMany relationships
    - only(): Field restriction
    - defer(): Field exclusion
    """
    return Product.objects.select_related('created_by') \\
        .prefetch_related(
            'categories',
            'images',
            'features',
            'reviews'
        ) \\
        .only('id', 'title', 'slug', 'price', 'status')
```

10. API Design Patterns

RESTful Endpoint Design

```
# Resource-based URL structure
/api/v1/products/              # Collection
/api/v1/products/{slug}/       # Single resource
/api/v1/products/{slug}/similar/  # Sub-resource
/api/v1/products/search/       # Action endpoint

# HTTP Method Semantics
GET     # Retrieve resources
POST    # Create resources
PUT     # Update resources
PATCH   # Partial update
DELETE  # Delete resources
```

Pagination Implementation

```
# Django REST Framework Pagination
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberP
agination',
    'PAGE_SIZE': 20,
}
```

```
# Paginated Response Structure
{
    "count": 150,           # Total items
    "next": "<http://api/products/?page=2>",
    "previous": null,
    "results": [...]        # Current page items
}

# Custom Pagination Parameters
GET /api/products/?page=2&page_size=50
```

Error Handling

```
# Standard Error Response Format
{
    "error": {
        "code": "validation_error",
        "message": "Invalid input data",
        "details": {
            "price": ["This field is required."],
            "email": ["Enter a valid email address."]
        },
        "timestamp": "2024-01-15T10:00:00Z",
        "request_id": "req_123456"
    }
}

# HTTP Status Codes
200 OK                    # Successful request
201 Created               # Resource created
400 Bad Request           # Validation error
401 Unauthorized          # Authentication required
403 Forbidden             # Permission denied
404 Not Found             # Resource not found
```

```
429 Too Many Requests     # Rate limiting
500 Internal Server Error # Server error
```

11. Security Implementation

Input Validation

```
# Model Level Validation
class Product(models.Model):
    price = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        validators=[
            MinValueValidator(0),           # Non-negative
            MaxValueValidator(1000000)      # Maximum limit
        ]
    )

# Serializer Level Validation
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'

    def validate_price(self, value):
        if value <= 0:
            raise serializers.ValidationError("Price must be positive.")
        return value

# View Level Validation
def perform_create(self, serializer):
    if serializer.validated_data.get('stock_quantity', 0) < 0:
        raise ValidationError("Stock quantity cannot be negative.")
    serializer.save()
```

XSS & SQL Injection Protection

```
# Django Automatic Protection
# 1. Template Auto-escaping
# 2. ORM Parameterized Queries
# 3. CSRF Token Protection
# 4. Content Security Policy

# Manual Protection Measures
def safe_search(query):
    """
    Veilige search query handling.
    """
    # Remove dangerous characters
    safe_query = re.sub(r'[^\\w\\s\\-]', '', query)

    # Use parameterized queries
    return Product.objects.filter(
        title__icontains=safe_query
    )
```

File Upload Security

```
# File Validation
def validate_uploaded_file(file):
    """
    Validate uploaded files voor security.
    """
    # Size limit (10MB)
    if file.size > 10 * 1024 * 1024:
        raise ValidationError("File size exceeds 10MB limit.")

    # File type validation
    allowed_types = ['image/jpeg', 'image/png', 'image/webp']
    if file.content_type not in allowed_types:
        raise ValidationError("Invalid file type.")

    # File name sanitization
    file.name = sanitize_filename(file.name)

    return file
```

12. Performance Optimization

Caching Strategy

```
# View-level Caching
@method_decorator(cache_page(60 * 15))  # 15 minutes
@method_decorator(vary_on_cookie)
def list(self, request):
    return super().list(request)

# Template Fragment Caching
{% load cache %}
{% cache 300 product_detail product.slug %}
    <!-- Product detail content -->
{% endcache %}

# Database Query Caching
from django.core.cache import cache

def get_popular_products():
    cache_key = 'popular_products'
    products = cache.get(cache_key)

    if not products:
        products = Product.objects.filter(
            is_featured=True
        ).select_related('category')[:10]
        cache.set(cache_key, products, 60 * 5)  # 5 minutes
    return products
```

Database Optimization

```
# 1. Query Optimization
# Gebruik .only() en .defer() voor field restriction
products = Product.objects.only('title', 'price', 'slug')

# 2. Batch Operations
# Gebruik bulk_create voor grote datasets
Product.objects.bulk_create(products_list)

# 3. Index Optimization
# Create indexes voor vaak gebruikte filters
class Meta:
    indexes = [
        models.Index(fields=['slug', 'is_active']),
        models.Index(fields=['category', 'created_at']),
    ]

# 4. Connection Pooling
# Configure database connection pooling
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'company_services',
        'CONN_MAX_AGE': 600,  # Connection pooling
        'POOL_SIZE': 20,      # Connection pool size
    }
}
```

13. Testing Strategy

Unit Tests Structure

```
# Product Model Tests
class ProductModelTest(TestCase):
    def setUp(self):
        self.category = ProductCategory.objects.create(
            name='Test Category',
            slug='test-category'
        )

    def test_product_creation(self):
        """Test product creation en validation."""
        product = Product.objects.create(
            title='Test Product',
            slug='test-product',
            price=100.00,
            category=self.category
        )
        self.assertEqual(product.price, 100.00)
        self.assertTrue(product.is_active)

# API Endpoint Tests
class ProductAPITest(APITestCase):
    def setUp(self):
        self.client = APIClient()
        self.user = CustomUser.objects.create_user(
            email='test@example.com',
            password='testpass123'
        )

    def test_product_list(self):
        """Test product list endpoint."""
        response = self.client.get('/api/v1/products/products/')
        self.assertEqual(response.status_code, 200)

    def test_product_detail(self):
        """Test product detail endpoint."""
        product = Product.objects.create(
            title='Test Product',
            slug='test-product',
            price=100.00
        )
        response = self.client.get(f'/api/v1/products/products/{product.slug}/')
        self.assertEqual(response.status_code, 200)
```

Integration Tests

```
class ProductSearchTest(TestCase):
    def test_search_functionality(self):
        """Test geavanceerde search functionaliteit."""
        # Create test data
        Product.objects.create(
            title='Modern Chair',
            slug='modern-chair',
            price=150.00,
            material='Wood'
        )

        # Test search queries
        response = self.client.get('/api/v1/products/search/?q=chair&material=w
ood')
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Modern Chair')
```

14. Deployment Architecture

Docker Configuration

```
# docker-compose.yml
version: '3.8'
```

```
services:
  mongodb:
    image: mongo:latest
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: ${DB_PASSWORD}
    volumes:
      - mongodb_data:/data/db
    ports:
      - "27017:27017"

  backend:
    build: ./backend
    environment:
      - DB_HOST=mongodb
      - DB_NAME=company_services
      - SECRET_KEY=${SECRET_KEY}
    depends_on:
      - mongodb
    ports:
      - "8000:8000"

  frontend:
    build: ./frontend
    environment:
      - REACT_APP_API_URL=http://localhost:8000/api/v1
    ports:
      - "3000:3000"
```

Production Settings

```
# config/settings/production.py
DEBUG = False
```

```
# Security settings
SECURE_SSL_REDIRECT = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
X_FRAME_OPTIONS = 'DENY'

# Database connection pooling
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': config('DB_NAME'),
        'CLIENT': {
            'host': config('DB_HOST'),
            'port': config('DB_PORT', cast=int),
            'username': config('DB_USER'),
            'password': config('DB_PASSWORD'),
            'authSource': 'admin',
        },
        'CONN_MAX_AGE': 600,
    }
}

# Static files with Whitenoise
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFiles
Storage'
```

15. Monitoring & Logging

Logging Configuration

```
# config/settings/base.py
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d} {mes
sage}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'handlers': {
        'file': {
            'level': 'ERROR',
            'class': 'logging.FileHandler',
            'filename': BASE_DIR / 'logs/django.log',
            'formatter': 'verbose',
        },
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': True,
        },
        'products': {
            'handlers': ['file', 'console'],
            'level': 'DEBUG',
            'propagate': False,
        },
    },
}
```

Performance Monitoring

```
# Middleware voor request timing
import time
from django.utils.deprecation import MiddlewareMixin

class TimingMiddleware(MiddlewareMixin):
    def process_request(self, request):
        request.start_time = time.time()

    def process_response(self, request, response):
        if hasattr(request, 'start_time'):
            duration = time.time() - request.start_time
            # Log slow requests
            if duration > 2.0:  # 2 seconds threshold
                logger.warning(
                    f"Slow request: {request.path} took {duration:.2f}s"
                )
        return response
```

16. Conclusie & Best Practices

Architectuur Samenvatting

1. **Layered Architecture**: Models → Serializers → Views → URLs
2. **Separation of Concerns**: Elk component heeft een specifieke rol
3. **RESTful Design**: Resource-based API endpoints
4. **Security First**: JWT auth, input validation, file security
5. **Performance Focus**: Caching, indexing, query optimization
6. **Scalability Ready**: Containerized, stateless, load-balanced

Key Design Decisions

1. **MongoDB over PostgreSQL**: Voor flexibele schema's en document storage
2. **Djongo ORM**: Behoud Django ORM benefits met MongoDB
3. **JWT Authentication**: Stateless, scalable authenticatie
4. **Modular App Structure**: Elke business domain als aparte app
5. **API Versioning**: v1/v2 voor backward compatibility

Development Workflow

```
# 1. Environment setup
python -m venv venv
source venv/bin/activate
pip install -r requirements/development.txt

# 2. Database setup
python manage.py migrate
python manage.py createsuperuser

# 3. Development
python manage.py runserver

# 4. Testing
pytest
python manage.py test

# 5. Deployment
docker-compose up -d
```

Onderhoud & Scaling

1. **Database Scaling**: MongoDB sharding voor grote datasets
2. **API Scaling**: Load balancing met Nginx
3. **Caching Strategy**: Redis voor session en query caching

4. **Monitoring**: Prometheus + Grafana voor metrics
5. **Backup Strategy**: Automatische database backups

Totale Codebase: ±2,500-3,000 regels code

Technische Complexiteit: Medium-High

Scalability: High (microservices ready)

Security Level: Production-ready

Performance: Optimized voor 10K+ daily requests

Technische Concepten Tabel

Database & ORM Concepten

Technisch Concept	Omschrijving	Implementatie in Project
ORM (Object-Relational Mapper)	Techniek om database records als Python objecten te behandelen	Django ORM met Djongo voor MongoDB
Models	Python classes die database tabellen vertegenwoordigen	Product, Service, CustomUser classes
Migrations	Systeem om database schema wijzigingen bij te houden	`python manage.py makemigrations`
Foreign Key	Relatie tussen twee tabellen waar één verwijst naar een andere	`models.ForeignKey(Product, ...)`
ManyToMany Field	Relatie waar records meerdere relaties kunnen hebben	`categories = models.ManyToManyField(...)`
OneToOne Field	Één-op-één relatie tussen twee modellen	UserProfile → CustomUser
QuerySet	Lazy-evaluated collectie van database records	`Product.objects.filter(is_active=True)`
Manager	Interface voor database query operations	`objects = CustomUserManager()`

Technisch Concept	Omschrijving	Implementatie in Project
Indexes	Database optimalisatie voor snellere queries	`models.Index(fields=['slug'])`
Transactions	Atomic database operations die samen slagen of falen	`@transaction.atomic` decorator

API & REST Concepten

Technisch Concept	Omschrijving	Implementatie in Project
REST (Representational State Transfer)	Architectuurstijl voor web APIs	Resource-based endpoints
CRUD Operations	Create, Read, Update, Delete operaties	ViewSets met GET, POST, PUT, DELETE
Serializer	Converteert complex data types naar JSON en vice versa	`ProductSerializer`, `ServiceSerializer`
ViewSet	Class-based view die alle CRUD operaties bevat	`ProductViewSet`, `ServiceViewSet`
Generic Views	Pre-built views voor veelvoorkomende patronen	`ListAPIView`, `RetrieveAPIView`
Mixin	Herbruikbare class componenten	`ListModelMixin`, `CreateModelMixin`
Router	Automatische URL routing voor ViewSets	`DefaultRouter()`
Pagination	Splits grote datasets in pagina's	`PageNumberPagination`
Filtering	Dynamische data filtering via query parameters	`DjangoFilterBackend`
Search	Volledige tekst zoeken over meerdere velden	`filters.SearchFilter`
Ordering	Sorteren van resultaten	`filters.OrderingFilter`

Authentication & Security

Technisch Concept	Omschrijving	Implementatie in Project
JWT (JSON Web Token)	Token-based authenticatie systeem	`djangorestframework-simplejwt`
Access Token	Kort-levende token voor API toegang	Verloopt na 60 minuten
Refresh Token	Lang-levende token voor nieuwe access tokens	Verloopt na 1 dag
Permission Classes	Regels voor toegangscontrole	`IsAuthenticated` , `IsAdminUser`
CORS (Cross-Origin Resource Sharing)	Veilige cross-domain requests	`corsheaders.middleware.CorsMiddleware`
CSRF (Cross-Site Request Forgery)	Bescherming tegen ongeautoriseerde requests	Django CSRF tokens
XSS (Cross-Site Scripting)	Beveiliging tegen malicious scripts	Django template auto-escaping
SQL Injection	Bescherming tegen database aanvallen	Django ORM parameterized queries
HTTPS	Encrypted HTTP verbindingen	`SECURE_SSL_REDIRECT = True`
Environment Variables	Veilige opslag van gevoelige configuratie	`python-decouple` library

Python & Django Specifiek

Technisch Concept	Omschrijving	Implementatie in Project
Class-Based Views (CBV)	Views als Python classes i.p.v. functions	Alle views zijn CBV's
Function-Based Views (FBV)	Traditionele Django views als functions	Weinig gebruikt in dit project
Middleware	Software laag die requests/response verwerkt	`CorsMiddleware` , `SecurityMiddleware`
Signals	Systeem voor gebeurtenis-gebaseerde	`@receiver(pre_save, sender=Product)`

Technisch Concept	Omschrijving	Implementatie in Project
	acties	
Context Processors	Voegt data toe aan alle template contexts	`django.contrib.auth.context_processors.auth`
Template Tags	Custom template functies	`{% load custom_tags %}`
Management Commands	Custom command-line scripts	`python manage.py seed_products`
Static Files	CSS, JavaScript, images	`STATIC_URL` , `STATIC_ROOT`
Media Files	Geüploade gebruikersbestanden	`MEDIA_URL` , `MEDIA_ROOT`
Internationalization (i18n)	Meertalige ondersteuning	`gettext_lazy()` , locale directories

Design Patterns & Architecture

Technisch Concept	Omschrijving	Implementatie in Project
MVC (Model-View-Controller)	Software architectuur patroon	Django volgt MTV variant
MTV (Model-Template-View)	Django's variant van MVC	Models, Templates, Views
Singleton Pattern	Slechts één instantie van een class	`SiteConfig` model
Factory Pattern	Object creatie zonder specificatie	`CustomUserManager.create_user()`
Observer Pattern	Objecten reageren op gebeurtenissen	Django signals
Decorator Pattern	Dynamisch gedrag toevoegen aan objecten	`@method_decorator` , `@action`
Strategy Pattern	Verwisselbare algoritmes	Permission classes
Dependency Injection	External dependencies injecteren	Django settings, request object
Loose Coupling	Minimale afhankelijkheid tussen componenten	Apps zijn losjes gekoppeld

Technisch Concept	Omschrijving	Implementatie in Project
Separation of Concerns	Elk component heeft specifieke verantwoordelijkheid	Models vs Views vs Serializers

Data Structuren & Types

Technisch Concept	Omschrijving	Implementatie in Project
SerializerMethodField	Dynamisch berekende velden in serializers	`product_count = serializers.SerializerMethodField()`
Choice Field	Gelimiteerde set van mogelijke waarden	`CONDITION_CHOICES` , `STATUS_CHOICES`
Slug Field	URL-vriendelijke identificatie	`slug = models.SlugField(max_length=200, unique=True)`
Decimal Field	Exacte decimale getallen voor valuta	`price = models.DecimalField(max_digits=10, decimal_places=2)`
DateTime Field	Datum en tijd opslag	`created_at = models.DateTimeField(auto_now_add=True)`
Image Field	Afbeelding opslag	`image = models.ImageField(upload_to='products/')`
Boolean Field	True/False waarden	`is_active = models.BooleanField(default=True)`
TextField	Lange tekst opslag	`description = models.TextField()`
Email Field	Email validatie en opslag	`email = models.EmailField(unique=True)`
URL Field	URL validatie en opslag	`facebook_url = models.URLField(blank=True)`

Performance & Optimization

Technisch Concept	Omschrijving	Implementatie in Project
Query Optimization	Database queries efficiënter maken	`select_related()` , `prefetch_related()`
Lazy Loading	Data alleen laden wanneer nodig	QuerySets zijn lazy evaluated

Technisch Concept	Omschrijving	Implementatie in Project
Eager Loading	Alle benodigde data in één keer laden	`prefetch_related()` voor M2M
Caching	Snellere toegang tot vaak gebruikte data	`@cache_page(60 * 15)` decorator
Database Indexing	Snellere data retrieval	`models.Index(fields=['slug', 'is_active'])`
Connection Pooling	Hergebruik database connecties	`CONN_MAX_AGE = 600`
Pagination	Limiteren hoeveelheid data per request	`PageNumberPagination` met `PAGE_SIZE = 20`
Compression	Verkleinen van data grootte	`whitenoise.storage.CompressedManifestStaticFilesStorage`
Minification	Verwijderen onnodige characters	JavaScript/CSS minification in production
CDN (Content Delivery Network)	Distributed file serving	Static files via CDN in production

Testing Concepten

Technisch Concept	Omschrijving	Implementatie in Project
Unit Tests	Testen van individuele componenten	`ProductModelTest`, `ServiceViewTest`
Integration Tests	Testen van component samenwerking	`ProductSearchTest`, `OrderFlowTest`
Functional Tests	Testen vanuit gebruikersperspectief	API endpoint tests

Technisch Concept	Omschrijving	Implementatie in Project
Mocking	Simuleren van dependencies	`unittest.mock` voor externe services
Test Fixtures	Vooraf gedefinieerde test data	JSON fixtures voor initial data
Test Coverage	Percentage code dat getest wordt	`coverage.py` voor coverage reports
TDD (Test-Driven Development)	Eerst tests schrijven, dan code	Niet veel gebruikt in dit project
Continuous Integration	Automatisch testen bij code changes	GitHub Actions workflows
Assertions	Controleren of resultaten kloppen	`self.assertEqual()` , `self.assertContains()`
Test Client	Simuleren van HTTP requests	`APIClient()` voor API testing

Deployment & DevOps

Technisch Concept	Omschrijving	Implementatie in Project
Docker	Containerization platform	`Dockerfile` , `docker-compose.yml`
Container	Geïsoleerde applicatie omgeving	Docker containers voor app, DB, cache
Docker Compose	Multi-container applicatie beheer	Orchestration van MongoDB, Redis, Django
Virtual Environment	Geïsoleerde Python omgeving	`venv` directory
WSGI (Web Server Gateway Interface)	Python web server interface	`gunicorn` als WSGI server
ASGI (Asynchronous Server Gateway Interface)	Async web server interface	`uvicorn` voor async support
Reverse Proxy	Request forwarding en load balancing	Nginx voor static files en SSL
Load Balancing	Verdelen van requests over servers	Nginx load balancing config

Technisch Concept	Omschrijving	Implementatie in Project
Environment Variables	Configuratie via OS variabelen	`.env` file voor development
CI/CD Pipeline	Automatische build en deployment	GitHub Actions workflows

Error Handling & Logging

Technisch Concept	Omschrijving	Implementatie in Project
Exception Handling	Afhandelen van runtime errors	`try-except` blocks in views
Custom Exceptions	Applicatie-specifieke error types	`ValidationError`, `PermissionDenied`
Logging Levels	Verschillende niveaus van logging	DEBUG, INFO, WARNING, ERROR, CRITICAL
Log Handlers	Waar logs worden opgeslagen	FileHandler, StreamHandler, SMTPHandler
Structured Logging	Gestructureerde log data	JSON formatted logs
Error Middleware	Centrale error afhandeling	Django's error middleware
HTTP Status Codes	Standardized response codes	200 OK, 400 Bad Request, 404 Not Found
Validation Errors	Data validatie fouten	Serializer validation errors
Graceful Degradation	Systeem blijft werken bij fouten	Fallback mechanismen
Circuit Breaker	Voorkomen cascading failures	Niet geïmplementeerd, wel mogelijk

Business Logic & Patterns

Technisch Concept	Omschrijving	Implementatie in Project
Business Logic Layer	Regels en processen van de applicatie	Models en managers bevatten business logic
Domain Model	Kern concepten van het business domein	Product, Service, Quote, Client

Technisch Concept	Omschrijving	Implementatie in Project
Service Layer	Complexe business operaties	Views en services voor complexe logica
Repository Pattern	Abstractie van data toegang	Django ORM fungeert als repository
Unit of Work	Transaction management	Django's transaction.atomic()
DTO (Data Transfer Object)	Data containers voor transport	Serializers fungeren als DTO's
Value Objects	Objecten zonder identity, alleen waarde	Price, Address, PhoneNumber
Aggregate Root	Main entity in een cluster	Product is aggregate root voor images, reviews
Domain Events	Gebeurtenissen in het business domein	Django signals voor domain events
CQRS (Command Query Responsibility Segregation)	Scheiding van reads en writes	Niet strikt geïmplementeerd

Frontend-Backend Integration

Technisch Concept	Omschrijving	Implementatie in Project
SPA (Single Page Application)	Frontend framework architectuur	React/Vue.js frontend communicatie
API Gateway	Eén toegangspunt voor alle APIs	Django URL routing als gateway
GraphQL	Query language voor APIs	Niet gebruikt, wel REST
WebSockets	Bidirectionele real-time communicatie	Niet geïmplementeerd, wel mogelijk
Server-Sent Events	Eenrichtings real-time communicatie	Niet geïmplementeerd
JSON Schema	Validatie van JSON structuren	Serializers fungeren als schema validatie
HATEOAS (Hypermedia)	Hyperlinks in API responses	Niet geïmplementeerd, mogelijk met DRF

Technisch Concept	Omschrijving	Implementatie in Project
API Versioning	Versiebeheer van API endpoints	`/api/v1/`, `/api/v2/` structure
Rate Limiting	Beperken aantal requests per gebruiker	`django-ratelimit` of custom middleware
API Documentation	Documentatie van API endpoints	Swagger/OpenAPI (DRF heeft auto-docs)

Monitoring & Analytics

Technisch Concept	Omschrijving	Implementatie in Project
Application Performance Monitoring (APM)	Prestatie monitoring van applicatie	Django Debug Toolbar voor development
Log Aggregation	Centrale log verzameling	ELK Stack (Elasticsearch, Logstash, Kibana) mogelijk
Metrics Collection	Verzamelen van prestatie metrics	Prometheus metrics endpoint
Health Checks	Controle van applicatie gezondheid	`/health/` endpoint
Distributed Tracing	Volgen van requests door systemen	Jaeger of Zipkin voor microservices
Alerting	Meldingen bij problemen	Email/SMS alerts bij errors
Dashboarding	Visualisatie van metrics en logs	Grafana dashboards
Business Intelligence	Analyse van business data	Export functionaliteit voor data analysis
Audit Logging	Bijhouden van gebruikersacties	Audit trail in belangrijke modellen
Real-time Analytics	Directe analyse van data	MongoDB aggregatie pipeline mogelijk

Caching Strategies

Technisch Concept	Omschrijving	Implementatie in Project
Cache Invalidation	Verwijderen verouderde cache data	Time-based expiration, manual invalidation
Cache Warming	Vooraf vullen van cache	Populaire producten bij startup
Cache Aside Pattern	Cache alleen bij miss	Application checkt eerst cache, dan database
Write Through Cache	Schrijven naar cache en database	Niet gebruikt, complex voor dit project
Write Behind Cache	Schrijven eerst naar cache	Asynchrone schrijfacties
Redis	In-memory data structure store	Session storage, cache backend
Memcached	Distributed memory caching	Alternatief voor Redis
Cache Key Design	Gestructureerde cache keys	`f"product:{slug}:detail"`
Cache Regions	Logische scheiding van cache data	Separate caches voor producten, diensten, gebruikers
Cache Stampede	Veelvuldige cache misses tegelijk	Locking mechanismen bij populaire data

Security Deep Dive

Technisch Concept	Omschrijving	Implementatie in Project
OWASP Top 10	Meest kritieke web security risico's	Bescherming tegen injection, XSS, etc.
Password Hashing	Veilige opslag van wachtwoorden	Django's PBKDF2 met SHA256
Salt	Extra input voor password hashing	Automatisch door Django
Brute Force Protection	Beveiliging tegen herhaalde login pogingen	Rate limiting op login endpoint
Session Hijacking	Bescherming tegen gestolen sessions	Secure cookies, session expiration

Technisch Concept	Omschrijving	Implementatie in Project
Man-in-the-Middle	Bescherming tegen afluisteren	HTTPS enforcement, HSTS headers
Clickjacking	Bescherming tegen UI redirection	`X-Frame-Options: DENY` header
Content Security Policy	Beperken bronnen die geladen worden	CSP headers in production
SQL Injection Prevention	Bescherming tegen database aanvallen	Django ORM (parameterized queries)
File Upload Security	Veilige verwerking van uploads	File type validation, size limits

Totaal Aantal Concepten: 150+ technische concepten gedocumenteerd

Categorieën: Database, API, Security, Performance, Testing, Deployment, etc.

Complexiteitsniveau: Van basis tot geavanceerde concepten

Toepasbaarheid: Direct relevant voor Django/DRF ontwikkeling

Deze tabel geeft een uitgebreid overzicht van alle technische concepten die worden toegepast in het project, van basis ORM principes tot geavanceerde security patronen. Elk concept is gekoppeld aan de specifieke implementatie in jouw Django project.