Cursor Rules for CreatorSync Development

Introduction

This document outlines the comprehensive rules and guidelines for using Cursor AI to develop the CreatorSync platform. Cursor is an AI-powered coding assistant that will be used to accelerate development while maintaining high code quality and consistency. These rules ensure that all code generated by Cursor adheres to our architectural principles, coding standards, and best practices.

Table of Contents

General Principles

1. Modularity First

- **Rule GP-1.1**: All code must be organized into small, focused modules with clear responsibilities.
- **Rule GP-1.2**: Each module should have a single responsibility and be loosely coupled with other modules.
- **Rule GP-1.3**: Use dependency injection to manage module dependencies.
- **Rule GP-1.4**: Export only what is necessary from each module.

2. Type Safety

- **Rule GP-2.1**: All code must be written in TypeScript with strict type checking enabled.
- **Rule GP-2.2**: Avoid using `any` type; use proper type definitions or generics instead.
- **Rule GP-2.3**: Create and use interfaces for all data structures.
- **Rule GP-2.4**: Use union types and type guards for handling variable types.

3. Error Handling

- **Rule GP-3.1**: Implement comprehensive error handling for all operations that may fail.
- **Rule GP-3.2**: Use custom error classes that extend the base Error class.
- **Rule GP-3.3**: Include contextual information in error messages.
- **Rule GP-3.4**: Log errors with appropriate severity levels.

4. Naming Conventions

- **Rule GP-4.1**: Use descriptive, meaningful names for all variables, functions, and classes.
- **Rule GP-4.2**: Follow camelCase for variables and functions, PascalCase for classes and interfaces.
- **Rule GP-4.3**: Prefix interfaces with "I" (e.g., IUserService).
- **Rule GP-4.4**: Use verb prefixes for functions (e.g., getUserProfile, createPost).

5. Code Comments

- **Rule GP-5.1**: Include JSDoc comments for all public functions and classes.
- **Rule GP-5.2**: Explain "why" rather than "what" in comments.
- **Rule GP-5.3**: Keep comments up-to-date with code changes.
- **Rule GP-5.4**: Use TODO, FIXME, and NOTE tags with consistent formatting.

Code Organization Rules

1. Project Structure

- **Rule CO-1.1**: Organize code by feature rather than by type.
- **Rule CO-1.2**: Follow the established directory structure for each service:

```
service-name/
  src/
      config/           # Configuration files
      controllers/      # API controllers
      services/         # Business logic
      repositories/     # Data access
      models/           # Data models
      interfaces/       # TypeScript interfaces
      utils/            # Utility functions
      middleware/       # Express middleware
      validators/       # Input validation
      index.ts          # Service entry point
  test/                  # Tests
  package.json           # Dependencies
  tsconfig.json          # TypeScript configuration
```

- **Rule CO-1.3**: Keep related files close to each other.
- **Rule CO-1.4**: Limit file size to 300 lines; split larger files into modules.

2. Import Organization

- **Rule CO-2.1**: Group imports in the following order:
  1. External libraries
  2. Internal modules
  3. Relative imports
  4. Type imports
- **Rule CO-2.2**: Sort imports alphabetically within each group.
- **Rule CO-2.3**: Use absolute imports for internal modules.
- **Rule CO-2.4**: Avoid circular dependencies.

3. Code Splitting

- **Rule CO-3.1**: Split large components into smaller, focused components.
- **Rule CO-3.2**: Use lazy loading for routes and large components.
- **Rule CO-3.3**: Implement code splitting at the route level.
- **Rule CO-3.4**: Extract reusable logic into custom hooks or utility functions.

4. Configuration Management

- **Rule CO-4.1**: Store all configuration in environment variables or configuration files.
- **Rule CO-4.2**: Use a centralized configuration module to access configuration values.
- **Rule CO-4.3**: Validate configuration values at startup.
- **Rule CO-4.4**: Provide sensible defaults for all configuration options.

Frontend Development Rules

1. Component Structure

- **Rule FE-1.1**: Follow the functional component pattern with hooks.
- **Rule FE-1.2**: Organize components using the following structure:

```
// Imports
import React, { useState, useEffect } from 'react';
import { useQuery } from 'react-query';
import { Button } from '@/components/ui/button';

// Types
interface ContentCardProps {
  id: string;
  title: string;
  onAction: (id: string) => void;
}

// Component
export const ContentCard: React.FC<ContentCardProps> = ({
  id,
  title,
  onAction
}) => {
  // State hooks
  const [isExpanded, setIsExpanded] = useState(false);

  // Effect hooks
  useEffect(() => {
    // Effect logic
  }, [id]);

  // Event handlers
  const handleClick = () => {
    setIsExpanded(!isExpanded);
    onAction(id);
  };

  // Render
  return (
    <div className="rounded-lg p-4 bg-white shadow">
      <h3 className="text-lg font-medium">{title}</h3>
      <Button onClick={handleClick}>
        {isExpanded ? 'Collapse' : 'Expand'}
      </Button>
    </div>
  );
};
```

- **Rule FE-1.3**: Extract complex logic into custom hooks.
- **Rule FE-1.4**: Keep components focused on UI concerns; move business logic to hooks or services.

2. State Management

- **Rule FE-2.1**: Use React Query for server state management.

- **Rule FE-2.2**: Use Redux Toolkit for global application state.
- **Rule FE-2.3**: Use Zustand for simpler component-level state.
- **Rule FE-2.4**: Follow this decision tree for state management:
  1. Is it server data? → React Query
  2. Is it global state needed across many components? → Redux Toolkit
  3. Is it shared state needed by a few components? → Zustand
  4. Is it local component state? → useState

3. Styling

- **Rule FE-3.1**: Use Tailwind CSS for styling components.
- **Rule FE-3.2**: Follow utility-first approach with Tailwind.
- **Rule FE-3.3**: Extract common patterns into reusable components.
- **Rule FE-3.4**: Use CSS variables for theming and dynamic values.

4. Performance Optimization

- **Rule FE-4.1**: Memoize expensive calculations with useMemo.
- **Rule FE-4.2**: Memoize callback functions with useCallback.
- **Rule FE-4.3**: Use React.memo for components that render often but rarely change.
- **Rule FE-4.4**: Implement virtualization for long lists using react-window or react-virtualized.

5. Accessibility

- **Rule FE-5.1**: Ensure all interactive elements are keyboard accessible.
- **Rule FE-5.2**: Use semantic HTML elements.
- **Rule FE-5.3**: Include proper ARIA attributes when necessary.
- **Rule FE-5.4**: Maintain sufficient color contrast ratios.

Backend Development Rules

1. Service Structure

- **Rule BE-1.1**: Implement the controller-service-repository pattern.
- **Rule BE-1.2**: Controllers should only handle HTTP concerns.
- **Rule BE-1.3**: Services should contain business logic.
- **Rule BE-1.4**: Repositories should handle data access.

2. Dependency Injection

- **Rule BE-2.1**: Use NestJS dependency injection system.
- **Rule BE-2.2**: Define services as injectable classes.
- **Rule BE-2.3**: Inject dependencies through constructor parameters.
- **Rule BE-2.4**: Use interfaces for service contracts.

3. Error Handling

- **Rule BE-3.1**: Implement a global exception filter.
- **Rule BE-3.2**: Use custom exception classes for different error types.
- **Rule BE-3.3**: Return appropriate HTTP status codes.
- **Rule BE-3.4**: Include error codes and messages in error responses.

4. Middleware

- **Rule BE-4.1**: Implement authentication middleware.
- **Rule BE-4.2**: Use validation middleware for request validation.

- **Rule BE-4.3**: Implement rate limiting middleware.
- **Rule BE-4.4**: Use logging middleware for request/response logging.

5. Configuration

- **Rule BE-5.1**: Use environment variables for configuration.
- **Rule BE-5.2**: Validate configuration at startup.
- **Rule BE-5.3**: Use a configuration service to access configuration values.
- **Rule BE-5.4**: Support different configurations for different environments.

Database Interaction Rules

1. Data Access

- **Rule DB-1.1**: Use repositories to encapsulate data access logic.
- **Rule DB-1.2**: Implement the repository pattern for each entity.
- **Rule DB-1.3**: Use TypeORM for PostgreSQL and Mongoose for MongoDB.
- **Rule DB-1.4**: Define clear interfaces for repositories.

2. Schema Design

- **Rule DB-2.1**: Use migrations for schema changes in PostgreSQL.
- **Rule DB-2.2**: Define schemas with TypeORM entities or Mongoose schemas.
- **Rule DB-2.3**: Include proper indexes for frequently queried fields.
- **Rule DB-2.4**: Use appropriate data types for each field.

3. Query Optimization

- **Rule DB-3.1**: Use query builders for complex queries.
- **Rule DB-3.2**: Implement pagination for large result sets.
- **Rule DB-3.3**: Use eager loading to avoid N+1 query problems.
- **Rule DB-3.4**: Monitor and optimize slow queries.

4. Transactions

- **Rule DB-4.1**: Use transactions for operations that modify multiple records.
- **Rule DB-4.2**: Implement proper error handling and rollback for transactions.
- **Rule DB-4.3**: Keep transactions as short as possible.
- **Rule DB-4.4**: Use optimistic locking for concurrent updates.

5. Caching

- **Rule DB-5.1**: Implement Redis caching for frequently accessed data.
- **Rule DB-5.2**: Use cache invalidation strategies for data modifications.
- **Rule DB-5.3**: Implement TTL (Time To Live) for cached data.
- **Rule DB-5.4**: Use cache-aside pattern for database queries.

API Development Rules

1. RESTful API Design

- **Rule API-1.1**: Follow RESTful principles for API design.
- **Rule API-1.2**: Use appropriate HTTP methods (GET, POST, PUT, DELETE).
- **Rule API-1.3**: Use plural nouns for resource endpoints.
- **Rule API-1.4**: Implement proper status codes for responses.

2. GraphQL API Design

- **Rule API-2.1**: Define clear GraphQL schemas.
- **Rule API-2.2**: Implement resolvers for each field.
- **Rule API-2.3**: Use DataLoader for batching and caching.
- **Rule API-2.4**: Implement proper error handling in resolvers.

3. API Versioning

- **Rule API-3.1**: Include version in URL path (e.g., /api/v1/users).
- **Rule API-3.2**: Maintain backward compatibility within a version.
- **Rule API-3.3**: Document breaking changes between versions.
- **Rule API-3.4**: Support multiple API versions simultaneously during transitions.

4. Request Validation

- **Rule API-4.1**: Validate all request inputs.
- **Rule API-4.2**: Use class-validator for DTO validation.
- **Rule API-4.3**: Return descriptive validation error messages.
- **Rule API-4.4**: Implement custom validators for complex validation rules.

5. Response Formatting

- **Rule API-5.1**: Use consistent response format across all endpoints.
- **Rule API-5.2**: Include metadata in paginated responses.
- **Rule API-5.3**: Use camelCase for JSON property names.
- **Rule API-5.4**: Implement proper error response format.

Testing Rules

1. Unit Testing

- **Rule TEST-1.1**: Write unit tests for all business logic.
- **Rule TEST-1.2**: Use Jest for unit testing.
- **Rule TEST-1.3**: Mock external dependencies.
- **Rule TEST-1.4**: Aim for at least 80% code coverage.

2. Integration Testing

- **Rule TEST-2.1**: Write integration tests for API endpoints.
- **Rule TEST-2.2**: Use Supertest for HTTP testing.
- **Rule TEST-2.3**: Use test databases for integration tests.
- **Rule TEST-2.4**: Test happy paths and error scenarios.

3. End-to-End Testing

- **Rule TEST-3.1**: Implement E2E tests for critical user flows.
- **Rule TEST-3.2**: Use Cypress for E2E testing.
- **Rule TEST-3.3**: Run E2E tests in CI pipeline.
- **Rule TEST-3.4**: Use test data factories for consistent test data.

4. Test Organization

- **Rule TEST-4.1**: Organize tests to mirror the source code structure.
- **Rule TEST-4.2**: Use descriptive test names.
- **Rule TEST-4.3**: Follow the Arrange-Act-Assert pattern.
- **Rule TEST-4.4**: Group related tests in describe blocks.

5. Test Data

- **Rule TEST-5.1**: Use factories for generating test data.
- **Rule TEST-5.2**: Avoid hardcoded test data.
- **Rule TEST-5.3**: Clean up test data after tests.
- **Rule TEST-5.4**: Use realistic test data that represents production scenarios.

Security Rules

1. Authentication

- **Rule SEC-1.1**: Implement JWT-based authentication.
- **Rule SEC-1.2**: Store tokens securely (HTTP-only cookies).
- **Rule SEC-1.3**: Implement token refresh mechanism.
- **Rule SEC-1.4**: Use secure password hashing (bcrypt).

2. Authorization

- **Rule SEC-2.1**: Implement role-based access control.
- **Rule SEC-2.2**: Check permissions for all protected resources.
- **Rule SEC-2.3**: Implement principle of least privilege.
- **Rule SEC-2.4**: Use CASL for fine-grained permissions.

3. Data Protection

- **Rule SEC-3.1**: Encrypt sensitive data at rest.
- **Rule SEC-3.2**: Use HTTPS for all communications.
- **Rule SEC-3.3**: Implement proper input sanitization.
- **Rule SEC-3.4**: Follow data minimization principles.

4. API Security

- **Rule SEC-4.1**: Implement rate limiting.
- **Rule SEC-4.2**: Use CSRF protection for browser clients.
- **Rule SEC-4.3**: Set appropriate CORS headers.
- **Rule SEC-4.4**: Implement API key authentication for service-to-service communication.

5. Security Headers

- **Rule SEC-5.1**: Set Content-Security-Policy header.
- **Rule SEC-5.2**: Set X-Content-Type-Options: nosniff.
- **Rule SEC-5.3**: Set X-Frame-Options: DENY.
- **Rule SEC-5.4**: Set Strict-Transport-Security header.

Performance Rules

1. Frontend Performance

- **Rule PERF-1.1**: Minimize bundle size through code splitting.
- **Rule PERF-1.2**: Optimize images and assets.
- **Rule PERF-1.3**: Implement lazy loading for routes and components.
- **Rule PERF-1.4**: Use memoization for expensive calculations.

2. Backend Performance

- **Rule PERF-2.1**: Implement caching for frequently accessed data.
- **Rule PERF-2.2**: Optimize database queries.

- **Rule PERF-2.3**: Use pagination for large result sets.
- **Rule PERF-2.4**: Implement background processing for long-running tasks.

3. Database Performance

- **Rule PERF-3.1**: Create appropriate indexes.
- **Rule PERF-3.2**: Use query optimization techniques.
- **Rule PERF-3.3**: Implement database connection pooling.
- **Rule PERF-3.4**: Monitor and optimize slow queries.

4. API Performance

- **Rule PERF-4.1**: Implement response compression.
- **Rule PERF-4.2**: Use HTTP/2 for multiplexing.
- **Rule PERF-4.3**: Implement proper caching headers.
- **Rule PERF-4.4**: Optimize payload size.

5. Monitoring

- **Rule PERF-5.1**: Implement performance monitoring.
- **Rule PERF-5.2**: Set up alerts for performance degradation.
- **Rule PERF-5.3**: Collect and analyze performance metrics.
- **Rule PERF-5.4**: Conduct regular performance reviews.

Documentation Rules

1. Code Documentation

- **Rule DOC-1.1**: Use JSDoc comments for all public functions and classes.
- **Rule DOC-1.2**: Include parameter and return type documentation.
- **Rule DOC-1.3**: Document complex algorithms and business logic.
- **Rule DOC-1.4**: Keep documentation up-to-date with code changes.

2. API Documentation

- **Rule DOC-2.1**: Use OpenAPI/Swagger for REST API documentation.
- **Rule DOC-2.2**: Document all endpoints, parameters, and responses.
- **Rule DOC-2.3**: Include example requests and responses.
- **Rule DOC-2.4**: Document error responses and codes.

3. Architecture Documentation

- **Rule DOC-3.1**: Maintain architecture decision records (ADRs).
- **Rule DOC-3.2**: Document system architecture and components.
- **Rule DOC-3.3**: Include diagrams for complex systems.
- **Rule DOC-3.4**: Document integration points with external systems.

4. User Documentation

- **Rule DOC-4.1**: Create user guides for platform features.
- **Rule DOC-4.2**: Include screenshots and examples.
- **Rule DOC-4.3**: Organize documentation by user roles and tasks.
- **Rule DOC-4.4**: Keep documentation up-to-date with feature changes.

5. Technical Documentation

- **Rule DOC-5.1**: Document development environment setup.
- **Rule DOC-5.2**: Include deployment and configuration instructions.
- **Rule DOC-5.3**: Document troubleshooting procedures.
- **Rule DOC-5.4**: Maintain a changelog for version changes.

Workflow Integration

1. Using Cursor with Git

- **Rule WF-1.1**: Generate code in small, focused commits.
- **Rule WF-1.2**: Write descriptive commit messages.
- **Rule WF-1.3**: Review Cursor-generated code before committing.
- **Rule WF-1.4**: Follow the branch naming convention: `feature/[feature-name]`.

2. Code Review Process

- **Rule WF-2.1**: Submit Cursor-generated code for peer review.
- **Rule WF-2.2**: Address all review comments.
- **Rule WF-2.3**: Run linting and tests before submitting for review.
- **Rule WF-2.4**: Document complex implementations in pull request descriptions.

3. Continuous Integration

- **Rule WF-3.1**: Configure CI pipeline to run tests and linting.
- **Rule WF-3.2**: Fix CI failures before merging.
- **Rule WF-3.3**: Include code coverage reports in CI.
- **Rule WF-3.4**: Run security scans in CI pipeline.

4. Deployment Process

- **Rule WF-4.1**: Use automated deployments.
- **Rule WF-4.2**: Implement blue-green deployments.
- **Rule WF-4.3**: Run smoke tests after deployment.
- **Rule WF-4.4**: Implement rollback procedures.

5. Feature Flags

- **Rule WF-5.1**: Use feature flags for new features.
- **Rule WF-5.2**: Implement gradual rollout for major features.
- **Rule WF-5.3**: Clean up feature flags after full rollout.
- **Rule WF-5.4**: Document feature flags and their purpose.

Troubleshooting Guide

1. Common Issues and Solutions

Frontend Issues

- **Issue**: Component re-rendering too often
  - **Solution**: Check dependency arrays in useEffect and useCallback
  - **Solution**: Use React.memo for pure components
  - **Solution**: Move state up the component tree if necessary
- **Issue**: State updates not reflecting
  - **Solution**: Check for closure issues in event handlers
  - **Solution**: Ensure state updates are properly batched

  - **Solution**: Use functional updates for state based on previous state

Backend Issues

- **Issue**: Memory leaks
  - **Solution**: Check for unhandled promises
  - **Solution**: Ensure proper cleanup in event listeners
  - **Solution**: Monitor memory usage and implement garbage collection
- **Issue**: Slow API responses
  - **Solution**: Check database query performance
  - **Solution**: Implement caching
  - **Solution**: Use profiling tools to identify bottlenecks

2. Debugging Strategies

- **Rule TS-2.1**: Use logging strategically (not console.log everywhere).
- **Rule TS-2.2**: Implement structured logging with context.
- **Rule TS-2.3**: Use debugger statements for complex issues.
- **Rule TS-2.4**: Implement error boundaries in React components.

3. Performance Troubleshooting

- **Rule TS-3.1**: Use React DevTools Profiler for component performance.
- **Rule TS-3.2**: Implement performance monitoring for API endpoints.
- **Rule TS-3.3**: Use database query analyzers for slow queries.
- **Rule TS-3.4**: Monitor and analyze frontend bundle size.

4. Error Reporting

- **Rule TS-4.1**: Implement centralized error logging.
- **Rule TS-4.2**: Include context information in error reports.
- **Rule TS-4.3**: Set up alerts for critical errors.
- **Rule TS-4.4**: Analyze error patterns regularly.

Conclusion

These Cursor rules provide a comprehensive framework for developing the CreatorSync platform with consistency, quality, and efficiency. By following these guidelines, the development team can leverage Cursor AI to accelerate development while maintaining high standards of code quality and architecture.

The rules should be reviewed and updated regularly as the project evolves and new best practices emerge. All team members should be familiar with these rules and apply them consistently throughout the development process.

Appendix: Code Examples

Frontend Component Example

```
// src/components/ContentCard/ContentCard.tsx
import React, { useState, useCallback } from 'react';
import { useQuery } from 'react-query';
import { Card, CardContent, CardFooter, CardHeader, CardTitle } from '@/components/ui/card';
import { Button } from '@/components/ui/button';
import { getContentEngagement } from '@/services/analytics';
import { EngagementChart } from '@/components/EngagementChart';
import { IContent } from '@/interfaces/content';
```

```
interface ContentCardProps {
  content: IContent;
  onPublish: (contentId: string) => void;
  onEdit: (contentId: string) => void;
}

export const ContentCard: React.FC<ContentCardProps> = ({
  content,
  onPublish,
  onEdit
}) => {
  const [isExpanded, setIsExpanded] = useState(false);

  const { data: engagementData, isLoading } = useQuery(
    ['contentEngagement', content.id],
    () => getContentEngagement(content.id),
    { enabled: isExpanded }
  );

  const handleExpandClick = useCallback(() => {
    setIsExpanded(prev => !prev);
  }, []);

  const handlePublishClick = useCallback(() => {
    onPublish(content.id);
  }, [content.id, onPublish]);

  const handleEditClick = useCallback(() => {
    onEdit(content.id);
  }, [content.id, onEdit]);

  return (
    <Card className="w-full max-w-md">
      <CardHeader>
        <CardTitle>{content.title}</CardTitle>
      </CardHeader>
      <CardContent>
        <p className="text-sm text-gray-500">
          {content.platform} • {new Date(content.scheduledAt).toLocaleDateString()}
        </p>
        <p className="mt-2">{content.description}</p>

        {isExpanded && (
          <div className="mt-4">
            {isLoading ? (
              <p>Loading engagement data...</p>
            ) : (
              <EngagementChart data={engagementData} />
            )}
          </div>
        )}
      </CardContent>
      <CardFooter className="flex justify-between">
```

```
        <Button variant="outline" onClick={handleExpandClick}>
          {isExpanded ? 'Hide Details' : 'Show Details'}
        </Button>
        <div className="space-x-2">
          <Button variant="outline" onClick={handleEditClick}>
            Edit
          </Button>
          <Button onClick={handlePublishClick}>
            Publish
          </Button>
        </div>
      </CardFooter>
    </Card>
  );
};
```

Backend Service Example

```
// src/services/content.service.ts
import { Injectable, NotFoundException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Content } from '../entities/content.entity';
import { CreateContentDto } from '../dto/create-content.dto';
import { UpdateContentDto } from '../dto/update-content.dto';
import { PlatformService } from './platform.service';
import { ContentStatus } from '../enums/content-status.enum';
import { IContentService } from '../interfaces/content-service.interface';

@Injectable()
export class ContentService implements IContentService {
  constructor(
    @InjectRepository(Content)
    private contentRepository: Repository<Content>,
    private platformService: PlatformService,
  ) {}

  /**
   * Create a new content item
   * @param userId - The ID of the user creating the content
   * @param createContentDto - The content creation data
   * @returns The created content item
   */
  async createContent(userId: string, createContentDto: CreateContentDto): Promise<Content> {
    const content = this.contentRepository.create({
      ...createContentDto,
      userId,
      status: ContentStatus.DRAFT,
      createdAt: new Date(),
      updatedAt: new Date(),
    });

    return this.contentRepository.save(content);
  }
```

```typescript
/**
 * Get content by ID
 * @param contentId - The ID of the content to retrieve
 * @returns The content item
 * @throws NotFoundException if content is not found
 */
async getContentById(contentId: string): Promise<Content> {
  const content = await this.contentRepository.findOne({ where: { id: contentId } });

  if (!content) {
    throw new NotFoundException(`Content with ID ${contentId} not found`);
  }

  return content;
}

/**
 * Get all content for a user
 * @param userId - The ID of the user
 * @param status - Optional filter by content status
 * @returns Array of content items
 */
async getUserContent(userId: string, status?: ContentStatus): Promise<Content[]> {
  const query = this.contentRepository.createQueryBuilder('content')
    .where('content.userId = :userId', { userId });

  if (status) {
    query.andWhere('content.status = :status', { status });
  }

  return query.orderBy('content.scheduledAt', 'DESC').getMany();
}

/**
 * Update existing content
 * @param contentId - The ID of the content to update
 * @param updateContentDto - The content update data
 * @returns The updated content item
 * @throws NotFoundException if content is not found
 */
async updateContent(contentId: string, updateContentDto: UpdateContentDto): Promise<Content> {
  const content = await this.getContentById(contentId);

  Object.assign(content, {
    ...updateContentDto,
    updatedAt: new Date(),
  });

  return this.contentRepository.save(content);
}

/**
 * Publish content to its platform
```

```
   * @param contentId - The ID of the content to publish
   * @returns The published content item
   * @throws NotFoundException if content is not found
   */
  async publishContent(contentId: string): Promise<Content> {
    const content = await this.getContentById(contentId);

    // Publish to the appropriate platform
    await this.platformService.publishContent(content);

    // Update content status
    content.status = ContentStatus.PUBLISHED;
    content.publishedAt = new Date();
    content.updatedAt = new Date();

    return this.contentRepository.save(content);
  }

  /**
   * Delete content
   * @param contentId - The ID of the content to delete
   * @returns True if deletion was successful
   * @throws NotFoundException if content is not found
   */
  async deleteContent(contentId: string): Promise<boolean> {
    const content = await this.getContentById(contentId);

    await this.contentRepository.remove(content);
    return true;
  }
}
```

Database Entity Example

```
// src/entities/content.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne, JoinColumn, Index, CreateDateColumn, Updatel
import { User } from './user.entity';
import { ContentStatus } from '../enums/content-status.enum';
import { Platform } from '../enums/platform.enum';

@Entity('contents')
export class Content {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ length: 255 })
  title: string;

  @Column({ type: 'text', nullable: true })
  description: string;

  @Column({ type: 'enum', enum: Platform })
  platform: Platform;
```

```
  @Column({ type: 'enum', enum: ContentStatus, default: ContentStatus.DRAFT })
  status: ContentStatus;

  @Column({ type: 'jsonb', nullable: true })
  metadata: Record<string, any>;

  @Column({ type: 'timestamp with time zone', nullable: true })
  scheduledAt: Date;

  @Column({ type: 'timestamp with time zone', nullable: true })
  publishedAt: Date;

  @Column({ name: 'user_id' })
  @Index()
  userId: string;

  @ManyToOne(() => User, user => user.contents)
  @JoinColumn({ name: 'user_id' })
  user: User;

  @CreateDateColumn({ type: 'timestamp with time zone' })
  createdAt: Date;

  @UpdateDateColumn({ type: 'timestamp with time zone' })
  updatedAt: Date;
}
```

API Controller Example

```
// src/controllers/content.controller.ts
import { Controller, Get, Post, Put, Delete, Body, Param, Query, UseGuards, Req } from '@nestjs/common'
import { ApiTags, ApiOperation, ApiResponse, ApiBearerAuth } from '@nestjs/swagger';
import { ContentService } from '../services/content.service';
import { CreateContentDto } from '../dto/create-content.dto';
import { UpdateContentDto } from '../dto/update-content.dto';
import { Content } from '../entities/content.entity';
import { JwtAuthGuard } from '../guards/jwt-auth.guard';
import { ContentStatus } from '../enums/content-status.enum';
import { Request } from 'express';

@ApiTags('content')
@Controller('content')
@UseGuards(JwtAuthGuard)
@ApiBearerAuth()
export class ContentController {
  constructor(private readonly contentService: ContentService) {}

  @Post()
  @ApiOperation({ summary: 'Create new content' })
  @ApiResponse({ status: 201, description: 'Content created successfully', type: Content })
  async createContent(@Req() req, @Body() createContentDto: CreateContentDto): Promise<Content> {
    const userId = req.user.id;
    return this.contentService.createContent(userId, createContentDto);
  }
```

```
  @Get()
  @ApiOperation({ summary: 'Get all user content' })
  @ApiResponse({ status: 200, description: 'Content retrieved successfully', type: [Content] })
  async getUserContent(
    @Req() req,
    @Query('status') status?: ContentStatus
  ): Promise<Content[]> {
    const userId = req.user.id;
    return this.contentService.getUserContent(userId, status);
  }

  @Get(':id')
  @ApiOperation({ summary: 'Get content by ID' })
  @ApiResponse({ status: 200, description: 'Content retrieved successfully', type: Content })
  @ApiResponse({ status: 404, description: 'Content not found' })
  async getContentById(@Param('id') id: string): Promise<Content> {
    return this.contentService.getContentById(id);
  }

  @Put(':id')
  @ApiOperation({ summary: 'Update content' })
  @ApiResponse({ status: 200, description: 'Content updated successfully', type: Content })
  @ApiResponse({ status: 404, description: 'Content not found' })
  async updateContent(
    @Param('id') id: string,
    @Body() updateContentDto: UpdateContentDto
  ): Promise<Content> {
    return this.contentService.updateContent(id, updateContentDto);
  }

  @Post(':id/publish')
  @ApiOperation({ summary: 'Publish content' })
  @ApiResponse({ status: 200, description: 'Content published successfully', type: Content })
  @ApiResponse({ status: 404, description: 'Content not found' })
  async publishContent(@Param('id') id: string): Promise<Content> {
    return this.contentService.publishContent(id);
  }

  @Delete(':id')
  @ApiOperation({ summary: 'Delete content' })
  @ApiResponse({ status: 200, description: 'Content deleted successfully' })
  @ApiResponse({ status: 404, description: 'Content not found' })
  async deleteContent(@Param('id') id: string): Promise<{ success: boolean }> {
    const result = await this.contentService.deleteContent(id);
    return { success: result };
  }
}
```