NOVA PROJECT

Complete Implementation Guide

Step-by-Step Instructions for Week 1

⚠️ **Deadline: Week of 09/02/2026 | All code must be on GitHub BEFORE validation**

🚀 Quick Start Guide (All Developers)

Day 1: Initial Setup

☐ Install Symfony 6.4 and create new project

→ Open terminal and run: composer create-project symfony/skeleton:^6.4 nova-project

→ Navigate to project: cd nova-project

```
composer create-project symfony/skeleton:^6.4 nova-project
```

☐ Install required Symfony packages

→ Install ORM: composer require symfony/orm-pack

→ Install maker bundle: composer require --dev symfony/maker-bundle

→ Install validator: composer require symfony/validator

→ Install forms: composer require symfony/form

```
composer require symfony/orm-pack symfony/validator symfony/form
```

☐ Configure database connection

→ Open .env file in project root

→ Update DATABASE_URL line with your MySQL credentials

→ Example: DATABASE_URL="mysql://root:@127.0.0.1:3306/nova_db?serverVersion=8.0"

```
DATABASE_URL="mysql://username:password@127.0.0.1:3306/nova_db"
```

☐ Create database

→ Run command: php bin/console doctrine:database:create

💡 **Note:** *If you get an error, make sure XAMPP/WAMP MySQL is running*

Day 2: Template Integration

☐ Download Front Office template

→ Search for free Bootstrap admin templates (e.g., AdminLTE, CoreUI, SB Admin)

→ Download and extract template files

→ Copy CSS, JS, and image files to public/ folder in Symfony

☐ Install Twig for templating

→ Run: composer require twig

→ Create templates/base.html.twig as main layout

```
composer require twig
```

☐ Create base template structure

→ Copy HTML from template's index.html
→ Replace content area with {% block body %}{% endblock %}
→ Update CSS/JS paths to /css/ and /js/

☐ Download Back Office template (Admin)
→ Download separate admin template or use different layout
→ Create templates/admin/base.html.twig

☐ Test templates
→ Create a test controller to render templates
→ Access http://localhost:8000 and verify CSS/JS load correctly

```
php bin/console server:run
```

Day 3: GitHub Setup

☐ Initialize Git repository (ONE PERSON ONLY)
→ In project folder: git init
→ Create .gitignore file

```
git init
```

☐ Create .gitignore file
→ Add these lines to .gitignore:
→ /vendor/
→ /var/
→ /.env.local
→ /.env.local.php
→ /public/bundles/

☐ Create GitHub repository
→ Go to github.com and create new repository named 'NOVA-Project'
→ Copy the repository URL

☐ Push initial code to GitHub

```
git add .
git commit -m 'Initial Symfony project setup'
git remote add origin https://github.com/username/NOVA-Project.git
git push -u origin main
```

☐ Add team members as collaborators
→ Go to repository Settings > Collaborators
→ Add all team members by their GitHub usernames

☐ Team members clone repository
→ Each member runs: git clone <repository-url>
→ Navigate to folder: cd NOVA-Project
→ Install dependencies: composer install
→ Copy .env to .env.local and configure database
→ Create database: php bin/console doctrine:database:create

👤 MODULE 1: Users & Authentication (Oussama)

Entities: User, StudentProfile, TutorProfile

Step 1: Create User Entity

☐ Generate User entity with Maker Bundle

```
php bin/console make:entity User
```

☐ Add all User fields when prompted:

→ email (string, 180, not nullable)
→ password (string, 255, not nullable)
→ username (string, 100, not nullable)
→ role (string, 50, not nullable, default: STUDENT)
→ isActive (boolean, default: true)

☐ Edit src/Entity/User.php to add timestamps

→ Add: use Doctrine\ORM\Mapping as ORM;
→ Add property: private ?\DateTimeImmutable $createdAt = null;
→ Add property: private ?\DateTime $updatedAt = null;
→ Add #[ORM\Column] above each property
→ Generate getters/setters

☐ Add UniqueEntity constraint for email and username

→ At top of User class add:

```
#[ORM\Entity]
#[UniqueEntity(fields: ['email'], message: 'Email already exists')]
#[UniqueEntity(fields: ['username'], message: 'Username taken')]
```

Step 2: Create StudentProfile Entity

☐ Generate StudentProfile entity

```
php bin/console make:entity StudentProfile
```

☐ Add StudentProfile fields:

→ firstName (string, 100, not nullable)
→ lastName (string, 100, not nullable)
→ bio (text, nullable)
→ university (string, 200, nullable)
→ major (string, 100, nullable)
→ academicLevel (string, 50, nullable)
→ profilePicture (string, 255, nullable)
→ interests (text, nullable) - will store JSON

☐ Add OneToOne relationship to User

→ When prompted for relation type: choose OneToOne
→ Related entity: User
→ Add property user in StudentProfile? Yes
→ Make relationship nullable? No

Step 3: Create TutorProfile Entity

☐ Generate TutorProfile entity

```
php bin/console make:entity TutorProfile
```

☐ Add TutorProfile fields:

→ firstName (string, 100, not nullable)
→ lastName (string, 100, not nullable)
→ bio (text, nullable)
→ expertise (text, nullable) - will store JSON array
→ qualifications (text, nullable)
→ yearsOfExperience (integer, default: 0)
→ hourlyRate (decimal, precision: 10, scale: 2, nullable)
→ isAvailable (boolean, default: true)
→ profilePicture (string, 255, nullable)

☐ Add OneToOne relationship to User

Step 4: Create and Run Migration

☐ Generate migration file

```
php bin/console make:migration
```

💡 **Note:** *This creates a file in migrations/ folder with SQL commands*

☐ Review migration file

→ Open the generated file in migrations/ folder
→ Check that all tables and columns are created correctly

☐ Execute migration

```
php bin/console doctrine:migrations:migrate
```

💡 **Note:** *Type 'yes' when prompted to execute migrations*

☐ Verify database tables created

→ Open phpMyAdmin or MySQL Workbench
→ Check that user, student_profile, and tutor_profile tables exist

Step 5: Install Security Components

☐ Install Security Bundle

```
composer require symfony/security-bundle
```

☐ Install JWT Authentication Bundle

```
composer require lexik/jwt-authentication-bundle
```

☐ Generate JWT keys

```
php bin/console lexik:jwt:generate-keypair
```

💡 **Note:** *This creates private.pem and public.pem in config/jwt/*

☐ Configure security.yaml

→ Open config/packages/security.yaml
→ Add User as user provider

→ Configure JWT authenticator

⚠️ **Warning:** This configuration is complex - refer to LexikJWTAuthenticationBundle docs

Step 6: Create Controllers & Routes

☐ Create AuthController for registration and login

```
php bin/console make:controller AuthController
```

☐ Implement register() method

→ Create route: #[Route('/api/auth/register', methods: ['POST'])]
→ Accept JSON request with email, password, username, role
→ Validate required fields
→ Hash password using PasswordHasherInterface
→ Create User entity and persist to database
→ Return JSON response with success message

☐ Implement login() method

→ Create route: #[Route('/api/auth/login', methods: ['POST'])]
→ Accept email and password
→ Verify credentials
→ Generate JWT token
→ Return token in JSON response

☐ Create StudentProfileController

```
php bin/console make:controller StudentProfileController
```

☐ Implement StudentProfile CRUD methods

→ index(): GET /api/students - list all students (admin only)
→ show($id): GET /api/students/{id} - get one student
→ update($id): PUT /api/students/{id} - update student profile
→ delete($id): DELETE /api/students/{id} - delete student

☐ Create TutorProfileController

```
php bin/console make:controller TutorProfileController
```

☐ Implement TutorProfile CRUD methods

→ index(): GET /api/tutors - list all tutors
→ show($id): GET /api/tutors/{id} - get one tutor
→ update($id): PUT /api/tutors/{id} - update tutor profile
→ toggleAvailability($id): PATCH /api/tutors/{id}/availability

Step 7: Add Server-Side Validation

☐ Add validation constraints to User entity

→ Use #[Assert\NotBlank] for required fields
→ Use #[Assert\Email] for email field
→ Use #[Assert\Length(min: 8)] for password
→ Use #[Assert\Choice(['STUDENT', 'TUTOR', 'ADMIN'])] for role

☐ Add validation to controllers

→ Inject ValidatorInterface in controller
→ Call $validator->validate($user) before persisting

→ If errors exist, return JSON error response

☐ Test validation
→ Try registering with invalid email - should get error
→ Try registering with short password - should get error
→ Try duplicate username - should get error

Step 8: Advanced Features

☐ Implement tutor search and filtering
→ Add query parameters to GET /api/tutors
→ Filter by expertise: ?expertise=Math
→ Filter by availability: ?available=true
→ Use DQL or QueryBuilder in repository

☐ Add role-based access control
→ Use #[IsGranted('ROLE_ADMIN')] on admin-only methods
→ Use #[IsGranted('ROLE_TUTOR')] for tutor-specific routes

☐ Implement JWT token refresh
→ Create /api/auth/refresh endpoint
→ Accept refresh token and return new access token

Step 9: Create Front-End Pages

☐ Create login page template
→ Create templates/auth/login.html.twig
→ Add login form with email and password fields
→ NO HTML5 validation (no 'required' attribute)
→ Form submits to /api/auth/login via JavaScript fetch

☐ Create registration page
→ Create templates/auth/register.html.twig
→ Add form with email, username, password, role selection
→ Submit to /api/auth/register

☐ Create profile page
→ Create templates/profile/index.html.twig
→ Display user info (name, email, bio, etc.)
→ Add edit button linking to edit form

☐ Create tutor directory page
→ Create templates/tutors/index.html.twig
→ Display list of tutors with search bar
→ Add filters for expertise and availability

☐ Link all pages in navigation menu
→ Update base.html.twig navigation
→ Add links to Login, Register, Profile, Tutors

Step 10: Testing & GitHub

☐ Prepare test scenario

→ Register 2 students with different profiles
→ Register 2 tutors with expertise in different subjects
→ Login as student and view profile
→ Search for tutors by expertise
→ Login as admin and view all users

☐ Test all endpoints with Postman or Insomnia

→ Test POST /api/auth/register
→ Test POST /api/auth/login
→ Test GET /api/students (with JWT token)
→ Test PUT /api/students/{id}

☐ Commit and push to GitHub

```
git add .
git commit -m 'Module 1: User authentication and profiles complete'
git push origin main
```

🎮 MODULE 2: Gamification & Rewards (Nouha)

Entities: Game, Reward

Step 1: Create Game Entity

☐ Generate Game entity

```
php bin/console make:entity Game
```

☐ Add Game fields:

→ name (string, 200, not nullable, unique)
→ description (text, not nullable)
→ type (string, 50, not nullable) - ENUM values
→ difficulty (string, 20, not nullable) - ENUM values
→ tokenCost (integer, default: 0)
→ rewardTokens (integer, default: 0)
→ rewardXP (integer, default: 0)
→ isActive (boolean, default: true)
→ createdAt (datetime)

☐ Add ManyToOne relationship to User (createdBy)

→ Relation type: ManyToOne
→ Target entity: User
→ Field name: createdBy
→ Nullable: No

☐ Add validation constraints in Game entity

```
#[Assert\NotBlank(message: 'Game name is required')]
#[Assert\UniqueEntity(fields: ['name'])]
#[Assert\Choice(['PUZZLE', 'MEMORY', 'TRIVIA', 'ARCADE'])]
#[Assert\GreaterThanOrEqual(0, message: 'Token cost cannot be negative')]
```

Step 2: Create Reward Entity

☐ Generate Reward entity

```
php bin/console make:entity Reward
```

☐ Add Reward fields:

→ name (string, 200, not nullable)
→ description (text, nullable)
→ type (string, 50, not nullable) - ENUM
→ value (integer, not nullable) - XP or token value
→ requirement (text, nullable) - condition to unlock
→ icon (string, 255, nullable) - URL to icon image
→ isActive (boolean, default: true)

☐ Add validation to Reward

```
#[Assert\NotBlank]
#[Assert\Choice(['BADGE', 'ACHIEVEMENT', 'BONUS_XP', 'BONUS_TOKENS'])]
```

```
#[Assert\Positive(message: 'Value must be positive')]
```

Step 3: Create Migration and Database

☐ Generate migration

```
php bin/console make:migration
```

☐ Execute migration

```
php bin/console doctrine:migrations:migrate
```

☐ Verify tables in database

→ Check 'game' table exists with all columns
→ Check 'reward' table exists
→ Verify foreign key: game.created_by_id → user.id

Step 4: Create GameController

☐ Generate GameController

```
php bin/console make:controller GameController
```

☐ Implement Create Game (POST /api/games)

→ Create route: #[Route('/api/games', methods: ['POST'])]
→ Accept JSON: {name, description, type, difficulty, tokenCost, rewardTokens, rewardXP}
→ Set createdBy to current authenticated user
→ Validate data using Validator
→ Persist Game to database
→ Return JSON: {success: true, game: {...}}

☐ Implement List Games (GET /api/games)

→ Create route: #[Route('/api/games', methods: ['GET'])]
→ Get query parameters: ?type=PUZZLE&difficulty=EASY&free=true
→ Use GameRepository to filter results
→ Return JSON array of games

☐ Implement Get Game Details (GET /api/games/{id})

→ Find game by ID
→ Return full game details
→ Return 404 if not found

☐ Implement Update Game (PUT /api/games/{id})

→ Only admin or creator can update
→ Update fields from JSON request
→ Validate and persist

☐ Implement Delete Game (DELETE /api/games/{id})

→ Soft delete: set isActive = false
→ Only admin can delete

Step 5: Create RewardController

☐ Generate RewardController

```
php bin/console make:controller RewardController
```

☐ Implement Create Reward (POST /api/rewards)
→ Admin only route
→ Accept JSON with reward data
→ Validate and save
☐ Implement List Rewards (GET /api/rewards)
→ Filter by type if provided: ?type=BADGE
→ Return active rewards only (isActive = true)
☐ Implement Update/Delete Reward
→ PUT /api/rewards/{id}
→ DELETE /api/rewards/{id} (soft delete)

Step 6: Advanced Features - Search & Filter

☐ Create GameRepository custom methods
→ Open src/Repository/GameRepository.php
→ Add method: findByFilters($type, $difficulty, $freeOnly)
→ Use QueryBuilder to build dynamic query
☐ Implement search by name
→ Add query parameter: ?search=puzzle
→ Use LIKE query: WHERE name LIKE '%puzzle%'
☐ Implement sorting
→ Add parameter: ?sort=tokenCost&order=asc
→ Add ORDER BY clause to query
→ Support sorting by: name, tokenCost, difficulty, createdAt
☐ Filter free games only
→ Add parameter: ?free=true
→ Add WHERE tokenCost = 0

Step 7: Create Front-End Templates

☐ Create game marketplace page
→ Create templates/games/index.html.twig
→ Display grid of game cards with name, description, token cost
→ Add search bar at top
→ Add filter dropdowns: Type, Difficulty, Free/Paid
→ Fetch games from GET /api/games using JavaScript
☐ Create game details page
→ Create templates/games/show.html.twig
→ Display full game info, requirements, rewards
→ Show 'Play Game' button if user has enough tokens
☐ Create admin game management page (Back Office)
→ Create templates/admin/games.html.twig
→ Table with all games (active and inactive)
→ Add/Edit/Delete buttons
→ Create game form modal

☐ Create reward dashboard

→ Create templates/rewards/index.html.twig
→ Display available rewards
→ Show unlock requirements
→ Mark rewards user has already earned

☐ Link pages in navigation

→ Add 'Games' link in main menu
→ Add 'Rewards' link
→ Add 'Admin > Manage Games' (admin only)

Step 8: Testing

☐ Seed test data

→ Manually create 5 games via API or SQL
→ 2 free games (tokenCost = 0)
→ 3 premium games (tokenCost > 0)
→ Different types: PUZZLE, MEMORY, TRIVIA

☐ Test scenario

→ 1. Admin creates new game via POST /api/games
→ 2. User views game marketplace
→ 3. User filters by 'Free games only'
→ 4. User searches for 'puzzle'
→ 5. User sorts games by token cost (low to high)
→ 6. Admin edits game and changes difficulty
→ 7. Admin creates rewards linked to games

☐ Validate all constraints work

→ Try creating game with duplicate name - should fail
→ Try negative tokenCost - should fail
→ Try invalid game type - should fail

☐ Push to GitHub

```
git add .
git commit -m 'Module 2: Gamification complete with search and filters'
git push origin main
```

📚 MODULE 3: Library Management (Wassim)

Entities: Book, Loan

Step 1: Create Book Entity

☐ Generate Book entity

```
php bin/console make:entity Book
```

☐ Add Book fields:

→ title (string, 255, not nullable)
→ author (string, 200, not nullable)
→ isbn (string, 20, nullable, unique)
→ publisher (string, 200, nullable)
→ publishedYear (integer, nullable)
→ category (string, 100, not nullable)
→ description (text, nullable)
→ coverImage (string, 255, nullable)
→ pdfUrl (string, 255, nullable)
→ totalCopies (integer, default: 1)
→ availableCopies (integer, default: 1)
→ isActive (boolean, default: true)

☐ Add validation to Book entity

```
#[Assert\NotBlank(message: 'Title is required')]
#[Assert\NotBlank(message: 'Author is required')]
#[Assert\Isbn(type: 'isbn13', message: 'Invalid ISBN')]
#[Assert\LessThanOrEqual(value: 'current_year')]
#[Assert\Positive(message: 'Total copies must be positive')]
```

☐ Add custom validation: availableCopies <= totalCopies

→ Create validator constraint in src/Validator/
→ Check that availableCopies never exceeds totalCopies

Step 2: Create Loan Entity

☐ Generate Loan entity

```
php bin/console make:entity Loan
```

☐ Add Loan fields:

→ loanDate (date, not nullable)
→ dueDate (date, not nullable)
→ returnDate (date, nullable)
→ status (string, 50, not nullable) - ENUM
→ lateFee (decimal, precision: 10, scale: 2, default: 0.00)
→ notes (text, nullable)

☐ Add relationships:

→ ManyToOne to Book (bookId)

→ ManyToOne to User (userId)

☐ Add validation to Loan

```
#[Assert\NotBlank(message: 'Loan date required')]
#[Assert\NotBlank(message: 'Due date required')]
#[Assert\Choice(['ACTIVE', 'RETURNED', 'OVERDUE'])]
```

☐ Add custom constraint: loanDate <= dueDate

Step 3: Migration

☐ Create migration

```
php bin/console make:migration
```

☐ Execute migration

```
php bin/console doctrine:migrations:migrate
```

☐ Verify database

→ Check 'book' table created
→ Check 'loan' table created
→ Verify foreign keys: loan.book_id → book.id, loan.user_id → user.id

Step 4: Create BookController

☐ Generate BookController

```
php bin/console make:controller BookController
```

☐ Implement Create Book (POST /api/books) - Admin only

→ Route: #[Route('/api/books', methods: ['POST'])]
→ Check if user is admin: #[IsGranted('ROLE_ADMIN')]
→ Handle file upload for cover image
→ Validate book data
→ Save book to database
→ Return JSON response

☐ Handle cover image upload

→ Install VichUploaderBundle: composer require vich/uploader-bundle
→ Configure upload directory in config/packages/vich_uploader.yaml
→ Add uploadable annotation to Book entity
→ Handle file in controller and save path

☐ Implement List Books (GET /api/books)

→ Support pagination: ?page=1&limit=10
→ Support filters: ?category=TEXTBOOK&available=true
→ Support search: ?search=python
→ Use BookRepository for queries

☐ Implement Get Book (GET /api/books/{id})

→ Find book by ID
→ Return book details
→ Include availableCopies info

☐ Implement Update Book (PUT /api/books/{id})

→ Admin only
→ Update fields from request
→ Handle cover image replacement

☐ Implement Delete Book (DELETE /api/books/{id})
→ Soft delete: set isActive = false
→ Admin only

Step 5: Create LoanController

☐ Generate LoanController

```
php bin/console make:controller LoanController
```

☐ Implement Borrow Book (POST /api/loans)
→ Accept bookId in request
→ Check if book.availableCopies > 0
→ If available, create Loan record
→ Set loanDate = today
→ Set dueDate = today + 14 days
→ Set status = ACTIVE
→ Decrease book.availableCopies by 1
→ Save loan and update book
→ Return loan details

☐ Implement View User Loans (GET /api/loans)
→ Get loans for current user
→ Include book details in response
→ Filter by status if provided: ?status=ACTIVE

☐ Implement Return Book (PATCH /api/loans/{id}/return)
→ Find loan by ID
→ Check if loan belongs to current user
→ Set returnDate = today
→ Set status = RETURNED
→ Increase book.availableCopies by 1
→ If returnDate > dueDate, calculate lateFee
→ Save and return updated loan

☐ Implement Get All Loans (GET /api/admin/loans) - Admin only
→ List all loans in system
→ Filter by status, user, or book
→ Show overdue loans at top

Step 6: Advanced Features

☐ Implement book search
→ Create BookRepository::searchBooks($query)
→ Search in title, author, isbn using LIKE
→ Example: WHERE title LIKE '%python%' OR author LIKE '%python%'

☐ Implement category filter

→ Add WHERE category = :category
→ Support multiple categories: ?category[]=TEXTBOOK&category[]=REFERENCE

☐ Implement availability filter
→ Show only available books: WHERE availableCopies > 0

☐ Implement sorting
→ Sort by: title, author, publishedYear
→ Add parameter: ?sort=title&order=asc

☐ Implement automatic overdue detection
→ Create Symfony command or scheduled task
→ Check all loans where status = ACTIVE and today > dueDate
→ Update status to OVERDUE
→ Calculate lateFee: days overdue × 1 TND per day

☐ Alternative: Check overdue in real-time
→ In GET /api/loans, check if loan is overdue
→ Update status on-the-fly

Step 7: Create Front-End Pages

☐ Create book catalog page (Front Office)
→ Create templates/books/index.html.twig
→ Display grid of books with cover images
→ Add search bar at top
→ Add category filter dropdown
→ Add 'Show Available Only' checkbox
→ Fetch books via GET /api/books

☐ Create book details page
→ Create templates/books/show.html.twig
→ Display full book info (title, author, description, etc.)
→ Show availableCopies count
→ Add 'Borrow' button (disabled if availableCopies = 0)
→ Button calls POST /api/loans with bookId

☐ Create My Loans page
→ Create templates/loans/my-loans.html.twig
→ Display table of borrowed books
→ Show loan date, due date, status
→ Highlight overdue loans in red
→ Add 'Return' button for active loans

☐ Create admin book management (Back Office)
→ Create templates/admin/books.html.twig
→ Table with all books
→ Add 'Create Book' button
→ Edit/Delete buttons for each book
→ Form modal for adding/editing books

☐ Create admin loans dashboard (Back Office)
→ Create templates/admin/loans.html.twig

→ Show all loans (filter by status)
→ Highlight overdue loans
→ Show late fees
→ Sort by due date

☐ Update navigation menu
→ Add 'Library' link
→ Add 'My Loans' link
→ Add admin links: 'Manage Books', 'View All Loans'

Step 8: Testing

☐ Seed database with test books
→ Create 10 books manually or via SQL
→ Different categories: TEXTBOOK, REFERENCE, FICTION
→ Some with availableCopies = 0 (borrowed)
→ Some with cover images

☐ Test scenario
→ 1. Admin adds new book with cover image
→ 2. Student searches for 'Python' - finds relevant books
→ 3. Student filters by TEXTBOOK category
→ 4. Student borrows a book (availableCopies decreases)
→ 5. Check 'My Loans' - see borrowed book with due date
→ 6. Manually set loan to overdue (change due date in DB)
→ 7. Loan automatically shows as OVERDUE
→ 8. Student returns book (availableCopies increases)
→ 9. Late fee calculated correctly
→ 10. Admin views all overdue loans

☐ Validation tests
→ Try borrowing when availableCopies = 0 - should fail
→ Try creating book with invalid ISBN - should fail
→ Try publishing year in future - should fail

☐ Push to GitHub

```
git add .
git commit -m 'Module 3: Library management with search and overdue detection'
git push origin main
```

⏱️ MODULE 4: Study Sessions & Energy (Acil)

> 💡 **Note:** *This module has 6 entities - the most complex. Break it into 2 parts: Part A (Courses) and Part B (Sessions)*

Part A: Course Entities (Days 1-3)

- ☐ Create PersonalCourse entity with relationship to User
- ☐ Create TutorCourse entity with relationship to User (tutor)
- ☐ Create Enrollment entity with relationships to User and TutorCourse
- ☐ Add validation: prevent duplicate enrollments, check maxStudents limit
- ☐ Create controllers for all 3 entities with full CRUD
- ☐ Implement enrollment approval system (status: PENDING → APPROVED)

Part B: Study Session Entities (Days 4-6)

- ☐ Create StudySession entity (links to User and PersonalCourse)
- ☐ Create EnergyState entity (OneToOne with User)
- ☐ Create StudyPlan entity (calendar/scheduler)
- ☐ Implement start/stop session logic with energy calculation
- ☐ Implement energy regeneration (1 point per 30 min)
- ☐ Implement burnout risk detection (fatigueLevel > 70 = HIGH)

Advanced Features & UI

- ☐ Search courses by name/category, filter by difficulty
- ☐ Progress tracking: update course progress % on session completion
- ☐ Study planner calendar view
- ☐ Create course marketplace page (browse tutor courses)
- ☐ Create study timer page with energy bar visual
- ☐ Create session history page with statistics

❓ MODULE 5: Quiz System (Said)

Core Implementation

- ☐ Create Question entity with relationship to TutorCourse and User (creator)
- ☐ Create Choice entity with relationship to Question
- ☐ Validate: at least 2 choices, exactly one correct choice
- ☐ Implement question CRUD (create, list, update, delete)
- ☐ Cascade delete choices when question is deleted
- ☐ Implement choice CRUD (add/edit/delete choices)

Advanced Features

- ☐ Search questions by text or course
- ☐ Filter by difficulty (EASY, MEDIUM, HARD) and type
- ☐ Quiz randomization (select random X questions from course)
- ☐ Auto-grading for multiple-choice questions
- ☐ Display correct answers and explanations after submission

Frontend Pages

- ☐ Create question bank page (tutor creates questions)
- ☐ Create quiz taking page (student answers questions)
- ☐ Create results page (show score, correct answers)
- ☐ Create admin question management page

💬 MODULE 6: Social Features (Oumayma)

Core Implementation

☐ Create Post entity with relationship to User
☐ Create Comment entity with relationships to Post, User, and self (parentComment)
☐ Implement post CRUD (create, list, show, edit, delete)
☐ Implement comment CRUD with nested reply support
☐ Implement upvote functionality (increment upvotes count)
☐ Track view count (increment on each post view)

Advanced Features

☐ Search posts by title/content/tags
☐ Filter by category (DISCUSSION, QUESTION, ANNOUNCEMENT, RESOURCE)
☐ Sort by: newest, most viewed, most upvoted
☐ Pin posts (admin only - show at top)
☐ Lock posts (admin only - prevent new comments)
☐ Nested comment threading (unlimited depth)
☐ Show 'edited' badge if comment/post was edited

Frontend Pages

☐ Create forum homepage with categories
☐ Create post list page with search and filters
☐ Create post details page with nested comments
☐ Create post creation/editing form
☐ Create admin moderation dashboard

🔗 Final Integration & Deployment

Day Before Validation

☐ All developers pull latest code from GitHub

```
git pull origin main
```

☐ Resolve any merge conflicts

☐ Run all migrations on clean database

```
php bin/console doctrine:database:drop --force
php bin/console doctrine:database:create
php bin/console doctrine:migrations:migrate
```

☐ Seed database with comprehensive test data

→ 3-5 users (students, tutors, admin)
→ 10+ books, 5+ games, 10+ questions, 5+ posts
→ Some loans (active, overdue, returned)
→ Some courses with enrollments
→ Some study sessions

☐ Test complete user flow

→ Register → Login → Browse → Perform actions → Logout

☐ Verify all navigation links work

☐ Check all validations are server-side only

⚠️ **Warning:** Remove any 'required', 'pattern', or JavaScript validation

☐ Test API endpoints with Postman

☐ Prepare demo script

→ Write step-by-step scenario for validation demo
→ Practice walkthrough

☐ Document known issues

→ List any features not fully implemented

☐ Final GitHub push

```
git add .
git commit -m 'Final version ready for validation'
git push origin main
```

☐ Verify code is on GitHub

→ Go to GitHub repository and confirm latest commit

⚠️ Common Issues & Solutions

Database Issues

☐ Migration fails

→ Solution: Drop database and recreate from scratch

```
php bin/console doctrine:database:drop --force
php bin/console doctrine:database:create
php bin/console doctrine:migrations:migrate
```

☐ Foreign key constraint fails

→ Solution: Check entity relationships are correct

→ Ensure ManyToOne/OneToMany are properly configured

Validation Issues

☐ Validation not working

→ Ensure validation annotations are imported

```
use Symfony\Component\Validator\Constraints as Assert;
```

→ Inject ValidatorInterface in controller

→ Call validate() before persisting entity

☐ HTML validation still active

→ Remove ALL 'required' attributes from form inputs

→ Remove 'pattern' attributes

→ Remove JavaScript validation code

GitHub Issues

☐ Merge conflicts

→ Solution: Communicate with team before pushing

→ Pull before starting work: git pull origin main

→ If conflict occurs, resolve manually in files

→ Mark conflicts resolved: git add <file>

→ Complete merge: git commit

☐ Accidentally committed .env file

→ Add .env to .gitignore

→ Remove from Git: git rm --cached .env

→ Commit and push

API Issues

☐ CORS errors in frontend

→ Install NelmioCorsBundle

```
composer require nelmio/cors-bundle
```

→ Configure in config/packages/nelmio_cors.yaml

☐ JWT token not working

→ Verify JWT keys exist in config/jwt/
→ Check security.yaml configuration
→ Ensure token is sent in Authorization header

🎯 Success Criteria Reminder

✅ Templates integrated (Front + Back Office)
✅ All entities created with minimum 1 relationship
✅ Full CRUD operations working
✅ Server-side validation ONLY (NO HTML/JS)
✅ Advanced features (search, sort, filter)
✅ Code on GitHub BEFORE validation
✅ Test scenario prepared

Good luck! 🚀 You've got this!