Com certeza. Agora que todas as peças da arquitetura estão prontas, aqui está o diagrama de classes completo, por escrito, detalhando as variáveis (atributos) e os métodos principais de cada classe, conforme sua solicitação.

A estrutura está dividida nas camadas da aplicação.

Proposta de Diagrama de Classes Completo

Irei descrever as classes de cada camada, seus atributos e seus métodos principais.

---

1. Camada de Domínio (Entidades)

(Pacote: `com.belem.model.entities`) *Representam as tabelas do banco de dados (PostgreSQL).*

- `User` **(Entidade)**
  - **Atributos:**
    - `id` (Long)
    - `username` (String)
    - `passwordHash` (String)
    - `role` (UserRole)
  - **Relacionamentos:**
    - `professorProfile` (Professor) - (Opcional, `OneToOne` com `Professor`)
- `School` **(Entidade)**
  - **Atributos:**
    - `id` (Long)
    - `name` (String)
    - `status` (EntityStatus)
  - **Relacionamentos:**
    - `professors` (List<Professor>) - (`OneToMany` com `Professor`)
- `Professor` **(Entidade)**
  - **Atributos:**
    - `id` (Long)
    - `registrationNumber` (String)
    - `fullName` (String)
    - `status` (EntityStatus)
  - **Relacionamentos:**

        - `userAccount` (User) - (`OneToOne` com `User`)
        - `school` (School) - (`ManyToOne` com `School`)
        - `formations` (List<AcademicFormation>) - (`OneToMany` com `AcademicFormation`)
        - `availabilities` (List<ScheduleAvailability>) - (`OneToMany` com `ScheduleAvailability`)
        - `interests` (List<DisciplineInterest>) - (`OneToMany` com `DisciplineInterest`)
        - `allocations` (List<Allocation>) - (`OneToMany` com `Allocation`)
- **`Discipline` (Entidade)**
    - **Atributos:**
        - `id` (Long)
        - `acronym` (String)
        - `description` (String)
        - `workload` (Integer)
        - `status` (EntityStatus)
    - **Relacionamentos:**
        - `interests` (List<DisciplineInterest>) - (`OneToMany` com `DisciplineInterest`)
        - `allocations` (List<Allocation>) - (`OneToMany` com `Allocation`)
- **`TimeSlot` (Entidade)**
    - **Atributos:**
        - `id` (Long)
        - `dayOfWeek` (DayOfWeek)
        - `startTime` (LocalTime)
        - `endTime` (LocalTime)
        - `shift` (ClassShift)
        - `status` (EntityStatus)
    - **Relacionamentos:**
        - `availabilities` (List<ScheduleAvailability>) - (`OneToMany` com `ScheduleAvailability`)
- **`AcademicFormation` (Entidade)**
    - **Atributos:**
        - `id` (Long)
        - `degreeCategory` (DegreeCategory)
        - `institutionName` (String)
        - `courseName` (String)
        - `conclusionYear` (Integer)
    - **Relacionamentos:**

        - `professor` (Professor) - (`ManyToOne` com `Professor`)
- **`ScheduleAvailability` (Entidade Associativa)**
    - **Atributos:**
        - `id` (Long)
        - `semester` (String)
    - **Relacionamentos:**
        - `professor` (Professor) - (`ManyToOne` com `Professor`)
        - `timeSlot` (TimeSlot) - (`ManyToOne` com `TimeSlot`)
- **`DisciplineInterest` (Entidade Associativa)**
    - **Atributos:**
        - `id` (Long)
        - `semester` (String)
        - `priority` (Integer)
    - **Relacionamentos:**
        - `professor` (Professor) - (`ManyToOne` com `Professor`)
        - `discipline` (Discipline) - (`ManyToOne` com `Discipline`)
- **`Allocation` (Entidade Associativa)**
    - **Atributos:**
        - `id` (Long)
        - `semester` (String)
    - **Relacionamentos:**
        - `professor` (Professor) - (`ManyToOne` com `Professor`)
        - `discipline` (Discipline) - (`ManyToOne` com `Discipline`)
        - `timeSlot` (TimeSlot) - (`ManyToOne` com `TimeSlot`)

---

2. Camada de Dados (Repositórios)

(Pacote: `com.belem.model.repository`) *Interfaces que estendem `JpaRepository` e definem o acesso aos dados.*

- **`UserRepository`**
    - **Métodos:** `Optional<User> findByUsername(String)`
- **`SchoolRepository`**
    - **Métodos:** `Optional<School> findByName(String)`
- **`ProfessorRepository`**
    - **Métodos:** `List<Professor> findByStatus(EntityStatus)`, `Optional<Professor> findByRegistrationNumber(String)`, `Optional<Professor> findByUserAccount(User)`

- **`DisciplineRepository`**
  - **Métodos:** `List<Discipline> findByStatus(EntityStatus)`, `Optional<Discipline> findByAcronym(String)`
- **`TimeSlotRepository`**
  - **Métodos:** `List<TimeSlot> findByStatus(EntityStatus)`, `Optional<TimeSlot> findByDayOfWeekAndStartTimeAndEndTime(...)`
- **`ScheduleAvailabilityRepository`**
  - **Métodos:** `List<ScheduleAvailability> findByProfessorAndSemester(...)`, `void deleteByProfessorAndSemester(...)`, `boolean existsByProfessorAndTimeSlotAndSemester(...)`
- **`DisciplineInterestRepository`**
  - **Métodos:** `List<DisciplineInterest> findByProfessorAndSemester(...)`, `void deleteByProfessorAndSemester(...)`, `boolean existsByProfessorAndDisciplineAndSemester(...)`
- **`AllocationRepository`**
  - **Métodos:** `long countByDisciplineAndSemester(...)`, `List<Allocation> findBySemester(String)`
- **`AcademicFormationRepository`**
  - *(Nenhum método customizado necessário por enquanto)*

---

3. Camada de Transferência (DTOs)

(Pacote: `com.belem.dto`) *Classes simples usadas para enviar e receber dados da API (JSON).*

- **`SchoolDTO`**: `id`, `name`, `status`.
- **`DisciplineDTO`**: `id`, `acronym`, `description`, `workload`, `status`.
- **`TimeSlotDTO`**: `id`, `dayOfWeek`, `startTime`, `endTime`, `shift`, `status`.
- **`ProfessorDTO`**: `id`, `registrationNumber`, `fullName`, `status`, `schoolId`, `schoolName`, `username`, `password` (usado apenas na criação).
- **`AllocationRequestDTO`**: `semester`, `professorId`, `disciplineId`, `timeSlotId`.
- **`AllocationDTO`**: `id`, `semester`, `professorId`, `professorName`, `disciplineId`, `disciplineAcronym`, `timeSlotId`, `timeSlotDescription`.
- **`ScheduleAvailabilityRequestDTO`**: `semester`, `timeSlotIds` (List<Long>).

- **`DisciplineInterestRequestDTO`**: `semester`, `interests` (List<InterestItemDTO>).
- **`InterestItemDTO`**: `disciplineId`, `priority`.
- **`LoginRequest`**: `username`, `password`.
- **`JwtTokenResponse`**: `token`.

---

4. Camada de Negócio (Serviços)

(Pacote: `com.belem.model.service`) *Contêm a lógica de negócio e o CRUD. Injetam Repositórios.*

- **`SchoolService`**
  - **Métodos Públicos:** `createSchool(SchoolDTO)`, `getAllSchools()`, `getSchoolById(Long)`, `updateSchool(Long, SchoolDTO)`, `updateStatus(Long, EntityStatus)`, `deleteSchool(Long)`.
  - *Métodos Privados: `convertToDto(School)`, `convertToEntity(SchoolDTO)`.*
- **`DisciplineService`**
  - **Métodos Públicos:** `createDiscipline(DisciplineDTO)`, `getAllDisciplines()`, `getDisciplineById(Long)`, `getAllActiveDisciplines()`, `updateDiscipline(Long, DisciplineDTO)`, `updateStatus(Long, EntityStatus)`, `deleteDiscipline(Long)`.
  - *Métodos Privados: `convertToDto(Discipline)`, `convertToEntity(DisciplineDTO)`.*
- **`TimeSlotService`**
  - **Métodos Públicos:** `createTimeSlot(TimeSlotDTO)`, `getAllTimeSlots()`, `getAllActiveTimeSlots()`, `getTimeSlotById(Long)`, `updateTimeSlot(Long, TimeSlotDTO)`, `updateStatus(Long, EntityStatus)`, `deleteTimeSlot(Long)`.
  - *Métodos Privados: `convertToDto(TimeSlot)`, `convertToEntity(TimeSlotDTO)`.*
- **`ProfessorService`**
  - **Métodos Públicos:** `createProfessor(ProfessorDTO)`, `getAllProfessors()`, `getProfessorById(Long)`, `getProfessorsByStatus(EntityStatus)`, `updateProfessor(Long, ProfessorDTO)`, `updateStatus(Long, EntityStatus)`, `deleteProfessor(Long)`.

    - *Métodos Privados:* `convertToDto(Professor)`.
- **`AuthService`**
    - **Métodos Públicos:** `login(LoginRequest)`.
- **`AvailabilityService`**
    - **Métodos Públicos:** `saveScheduleAvailability(ScheduleAvailabilityRequestDTO)`, `saveDisciplineInterests(DisciplineInterestRequestDTO)`, `getMySchedule(String)`, `getMyInterests(String)`.
    - *Métodos Privados:* `getAuthenticatedProfessor()`.
- **`AllocationService`**
    - **Métodos Públicos:** `createAllocation(AllocationRequestDTO)`, `getAllocationsBySemester(String)`, `deleteAllocation(Long)`.
    - *Métodos Privados:* `convertToDto(Allocation)`.

---

5. Camada de API (Controladores)

(Pacote: `com.belem.controller`) *Expõem os endpoints REST para o Frontend (Angular). Injetam Serviços.*

- **`AuthController`**
    - **Métodos (Endpoints):** `authenticateUser(LoginRequest)` - `[POST /api/v1/auth/login]`
- **`SchoolController`**
    - **Métodos (Endpoints):** `createSchool(SchoolDTO)` - `[POST]`, `getAllSchools()` - `[GET]`, `getSchoolById(Long)` - `[GET /{id}]`, `updateSchool(Long, SchoolDTO)` - `[PUT /{id}]`, `updateStatus(Long, EntityStatus)` - `[PUT /{id}/status]`, `deleteSchool(Long)` - `[DELETE /{id}]`.
- **`DisciplineController`**
    - **Métodos (Endpoints):** `createDiscipline(DisciplineDTO)` - `[POST]`, `getAllDisciplines()` - `[GET]`, `getActiveDisciplines()` - `[GET /active]`, `getDisciplineById(Long)` - `[GET /{id}]`, `updateDiscipline(Long, DisciplineDTO)` - `[PUT /{id}]`, `updateStatus(Long, EntityStatus)` - `[PUT /{id}/status]`, `deleteDiscipline(Long)` - `[DELETE /{id}]`.
- **`TimeSlotController`**
    - **Métodos (Endpoints):** `createTimeSlot(TimeSlotDTO)` - `[POST]`, `getAllTimeSlots()` - `[GET]`, `getActiveTimeSlots()` - `[GET /active]`,

`getTimeSlotById(Long)` - `[GET /{id}]`, `updateTimeSlot(Long, TimeSlotDTO)` - `[PUT /{id}]`, `updateStatus(Long, EntityStatus)` - `[PUT /{id}/status]`, `deleteTimeSlot(Long)` - `[DELETE /{id}]`.

- **`ProfessorController`**
  - **Métodos (Endpoints):** `createProfessor(ProfessorDTO)` - `[POST]`, `getAllProfessors()` - `[GET]`, `getProfessorsByStatus(EntityStatus)` - `[GET /by-status]`, `getProfessorById(Long)` - `[GET /{id}]`, `updateProfessor(Long, ProfessorDTO)` - `[PUT /{id}]`, `updateStatus(Long, EntityStatus)` - `[PUT /{id}/status]`, `deleteProfessor(Long)` - `[DELETE /{id}]`.
- **`AvailabilityController`**
  - **Métodos (Endpoints):** `saveScheduleAvailability(ScheduleAvailabilityRequestDTO)` - `[POST /schedule]`, `saveDisciplineInterests(DisciplineInterestRequestDTO)` - `[POST /interests]`, `getMySchedule(String)` - `[GET /schedule]`, `getMyInterests(String)` - `[GET /interests]`.
- **`AllocationController`**
  - **Métodos (Endpoints):** `createAllocation(AllocationRequestDTO)` - `[POST]`, `getAllocationsBySemester(String)` - `[GET]`, `deleteAllocation(Long)` - `[DELETE /{id}]`.

---

6. Camada de Segurança

(Pacote: `com.belem.security`) *Classes que configuram e gerenciam o Spring Security e JWT.*

- **`SecurityConfig`**
  - **Atributos:** `jwtAuthFilter`, `authenticationProvider`.
  - **Métodos:** `securityFilterChain(HttpSecurity)`.
- **`ApplicationConfig`**
  - **Atributos:** `userRepository`.
  - **Métodos:** `userDetailsService()`, `authenticationProvider()`, `authenticationManager(AuthenticationConfiguration)`, `passwordEncoder()`.
- **`JwtService`**
  - **Atributos:** `secretKey`, `jwtExpiration`.

    - **Métodos:** `extractUsername(String)`, `generateToken(UserDetails)`, `isTokenValid(String, UserDetails)`, `extractClaim(...)`.
- **`JwtAuthFilter`**
    - **Atributos:** `jwtService`, `userDetailsService`.
    - **Métodos:** `doFilterInternal(...)`.

---

7. Camada de Exceções

(Pacote: `com.belem.exception`) *Classes para tratamento de erros padronizado.*

- **`ResourceNotFoundException`**
    - **Atributos:** `resourceName`, `fieldName`, `fieldValue`.
    - **Métodos:** (Construtor).
- **`ErrorResponse` (DTO)**
    - **Atributos:** `timestamp`, `status`, `error`, `message`, `path`.
    - **Métodos:** (Construtor).
- **`GlobalExceptionHandler` (`@ControllerAdvice`)**
    - **Métodos:** `handleResourceNotFoundException(...)` - (HTTP 404), `handleDataIntegrityViolation(...)` - (HTTP 409), `handleBadCredentialsException(...)` - (HTTP 401), `handleBusinessException(...)` - (HTTP 400), `handleGlobalException(...)` - (HTTP 500).

1. **Camada de Controller (API)** (`.controller`):
    - **Responsabilidade:** Ser a "porta de entrada". Ela lida com as URLs (endpoints), recebe dados JSON e envia dados JSON.
    - **Regra:** *Não contém lógica de negócio.* Ela apenas recebe a requisição e a repassa para o Service.
2. **Camada de Service (Negócio)** (`.service`):
    - **Responsabilidade: É O CÉREBRO DA APLICAÇÃO.** Todo o seu requisito de negócio está aqui.
    - **Regra:** Contém toda a lógica de negócio (CRUDs, validações como "máximo 5 alocações", "verificar se professor está disponível"). É ela quem coordena e decide o que fazer.
3. **Camada de Repository (Dados)** (`.repository`):
    - **Responsabilidade:** Falar com o banco de dados (PostgreSQL).
    - **Regra:** São interfaces (ex: `JpaRepository`) que o Spring implementa para nós. O Service usa o Repository para `salvar`, `buscar`, `deletar`, etc.
4. **Camada de Domain (Entidades)** (`.domain.entity`):
    - **Responsabilidade:** Mapear as tabelas do seu banco de dados.
    - **Regra:** São classes (POJOs) anotadas com `@Entity`. O `Professor.java` representa a tabela `professors` no banco.

Peças "Transversais" (Que ajudam todas as camadas):

- **Camada de Segurança (`.security`):**
    - **Função:** É o "Segurança do Prédio". Ela fica *antes* da Camada de Controller.
    - **Peças:** O `JwtAuthFilter` intercepta **toda** requisição. Ele valida o Token JWT. O `SecurityConfig` decide se o usuário (ADMIN ou PROFESSOR) tem permissão para acessar aquela URL.
- **Camada de Exceção (`.exception`):**
    - **Função:** É a "Rede de Segurança".
    - **Peças:** O `@ControllerAdvice` (`GlobalExceptionHandler`) fica "ouvindo". Se qualquer camada lançar uma exceção (como `ResourceNotFoundException` ou `BadCredentialsException`), ele a captura e envia uma resposta JSON de erro padronizada (o `ErrorResponse`) para o Angular.
- **DTOs (Data Transfer Objects) (`.dto`):**
    - **Função:** São os "Veículos Blindados" para transportar dados.
    - **Regra:** Evitam que você exponha suas Entidades (seu banco) para o mundo. O `ProfessorDTO`, por exemplo, permite que o Admin crie um professor, mas *nunca* retorna o `passwordHash` do usuário.

---

2. Como o Projeto Funciona (O Fluxo "Passo a Passo")

Vamos ver como as camadas interagem em dois cenários: **Login** e **Criação de Disciplina**.

Cenário 1: O Fluxo de Login (Autenticação)

1. O **Frontend (Angular)** envia um JSON (`LoginRequest` com usuário e senha) para a URL: `[POST] /api/v1/auth/login`.
2. O **`AuthController`** (Controller) recebe a requisição. Ele não faz nada, apenas chama: `authService.login(request)`.
3. O **`AuthService`** (Service) recebe. Ele pede ao `AuthenticationManager` do Spring: "Verifique este usuário e senha, por favor."
4. O `AuthenticationManager` usa o `PasswordEncoder` (para criptografar a senha) e o `UserDetailsService` (que busca no `UserRepository`) para confirmar se a senha bate com a do banco.
5. Se a senha estiver errada, ele lança `BadCredentialsException`. O **`GlobalExceptionHandler`** captura isso e retorna um JSON de erro `401 Unauthorized` ("Invalid username or password"). **FIM DO FLUXO.**
6. Se a senha estiver correta, o `AuthService` busca o `User` no banco e chama o **`JwtService`** para gerar um Token JWT.
7. O `AuthService` retorna o `JwtTokenResponse` (o token) para o `AuthController`.
8. O `AuthController` envia o JSON com o Token de volta para o Angular, com status `200 OK`.
9. O **Frontend (Angular)** recebe o token e o armazena (ex: no localStorage) para usar nas próximas requisições.

Cenário 2: O Fluxo de Dados (Admin criando uma Disciplina)

1. O **Frontend (Angular)** envia duas coisas para a URL `[POST] /api/v1/disciplines`:
    - O Token JWT (do login) no "Cabeçalho de Autorização" (Header).
    - O JSON (`DisciplineDTO`) com os dados da nova disciplina no "Corpo" (Body).
2. **A Segurança (`JwtAuthFilter`)** intercepta a requisição *antes* de chegar no Controller.
    - Ela vê o Token no Header, valida (usando o `JwtService`) e descobre que o usuário é "admin".
    - Ela autentica o usuário para esta requisição.
3. **A Segurança (`SecurityConfig`)** verifica: "O usuário 'admin' (ROLE_ADMIN) tem permissão para fazer `[POST]` em `/api/v1/disciplines`?" Sim, nós configuramos isso. A requisição é liberada.

4. O **`DisciplineController`** (Controller) finalmente recebe a requisição (o `DisciplineDTO`). Ele chama: `disciplineService.createDiscipline(dto)`.
5. O **`DisciplineService`** (Service) - O CÉREBRO - entra em ação:
    - **Regra 1:** "Esse DTO é válido?" (Vamos assumir que sim).
    - **Regra 2:** "Essa sigla já existe?" Ele chama `disciplineRepository.findByAcronym(dto.getAcronym())`.
    - **Se a sigla existir:** O Service lança `new RuntimeException("Discipline acronym already exists.")`.
    - **O `GlobalExceptionHandler`** captura essa exceção e envia um JSON de erro `400 Bad Request` para o Angular. **FIM DO FLUXO.**
    - **Se a sigla não existir:** O Service continua.
    - Ele converte o `DisciplineDTO` (DTO) para a `Discipline` (Entidade).
    - Ele seta o status: `entity.setStatus(EntityStatus.ACTIVE)`.
    - Ele chama o **`DisciplineRepository`** (Repository): `disciplineRepository.save(entity)`.
6. O **`DisciplineRepository`** (via Spring Data JPA) executa o comando SQL `INSERT INTO disciplines (...)` no **PostgreSQL**.
7. O Repositório retorna a entidade salva (agora com um ID) para o `DisciplineService`.
8. O `DisciplineService` converte a `Discipline` (Entidade) de volta para um `DisciplineDTO`.
9. O `DisciplineService` retorna o DTO para o `DisciplineController`.
10. O **`DisciplineController`** envia o JSON (`DisciplineDTO`) de volta para o Angular, com status `201 CREATED`.
11. O **Frontend (Angular)** recebe o JSON da disciplina criada e mostra a mensagem (Req 5: "resultado positivo").