

Curso Superior de Desenvolvimento de Software Multiplataforma

Erick Oliveira Nascimento, 3011392323039

Juliano Aparecido Ramalho, 3011392313013

Leon Pereira Pinto Fagundes, 3011392313007

Lucas Adami Gomes, 3011392313025

Matheus Garibaldi Rodrigues, 3011392313040

Vitor Ruan Firmino de Oliveira, 3011392313031

Projeto Interdisciplinar

Gestão Ágil de Projetos de Softwares

Desenvolvimento Web III

Banco de Dados Não Relacional

Interação Humano Computador

Cadastro de Professores

Orientadores: Profº Rodrigo de Paula Diver

Profa Maria Janaína da Silva Ferreira

Profo Jones Artur Gonçalves

Profo Ricardo Roberto Leme

Resumo

Este Projeto integrador foi elaborado através do tema apresentado pela coordenadora e professora do curso de Desenvolvimento de Software Multiplataforma Angelina Vitorino de Souza Melare, onde nos foi apresentado uma necessidade da criação de um sistema que Auxiliasse a secretária da Fatec no cadastro dos Professores as suas respectivas matérias lecionadas na faculdade.

1 - Descrição do Projeto

* 1. Proposta do Software

O presente projeto tem como finalidade auxiliar o trabalho da secretaria administrativa da Fatec de Votorantim, posteriormente podendo ser implementado em outras unidades, na função de cadastrar os professores e vinculá-los a matéria que os mesmos lecionam em suas respectivas unidades.

1.2 Justificativa

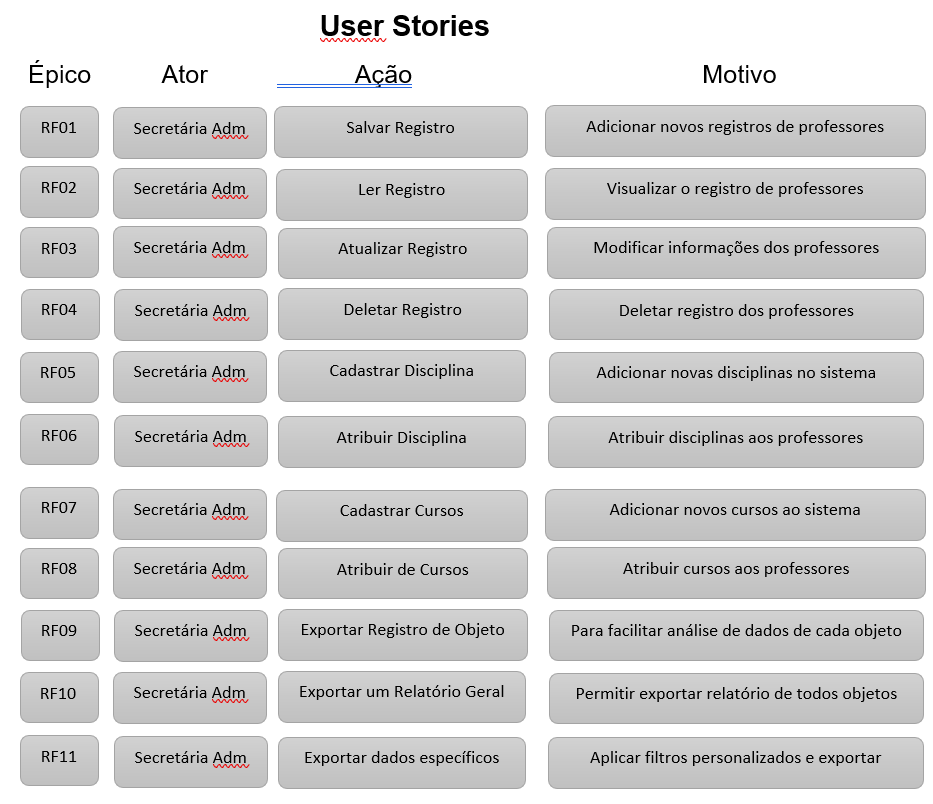
A ideia para o desenvolvimento do projeto surgiu da necessidade de se adequar ao tema proposto pela Coordenadora do curso de Desenvolvimento de software Multiplataforma e professora Angelina Vitorino de Souza Melare, onde a mesma apresentou uma dificuldade na secretária acadêmica da unidade, onde buscamos a criação de um sistema/site que facilitasse o cadastro dos professores da unidade as matérias que os mesmos lecionam na unidade e na geração de um relatório com os dados inseridos pela secretária.

1. – Desenvolvimento
   1. Levantamento de Requisitos

Através de uma entrevista realizada com a secretária administrativa da Fatec de Votorantim, com o intuito de fazermos um levantamento das dificuldades que a mesma estava encontrando durante a execução de algumas tarefas relacionadas a sua função. A partir do resultado dessa entrevista pudemos concluir que havia a necessidade da criação de um sistema/aplicação que facilitasse e auxiliasse a secretária no cadastro dos professores a sua respectiva matéria lecionada e na geração de relatórios com as informações inserida pela secretaria.

Após a entrevista foi elaborado o user stories e definido um backlog de produto afim de resolver os problemas apresentados:

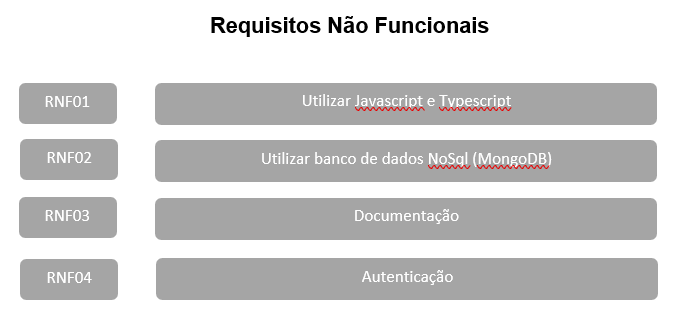
* 1. User Stories



* 1. Backlog de Produtos e   
     Requisitos funcionais

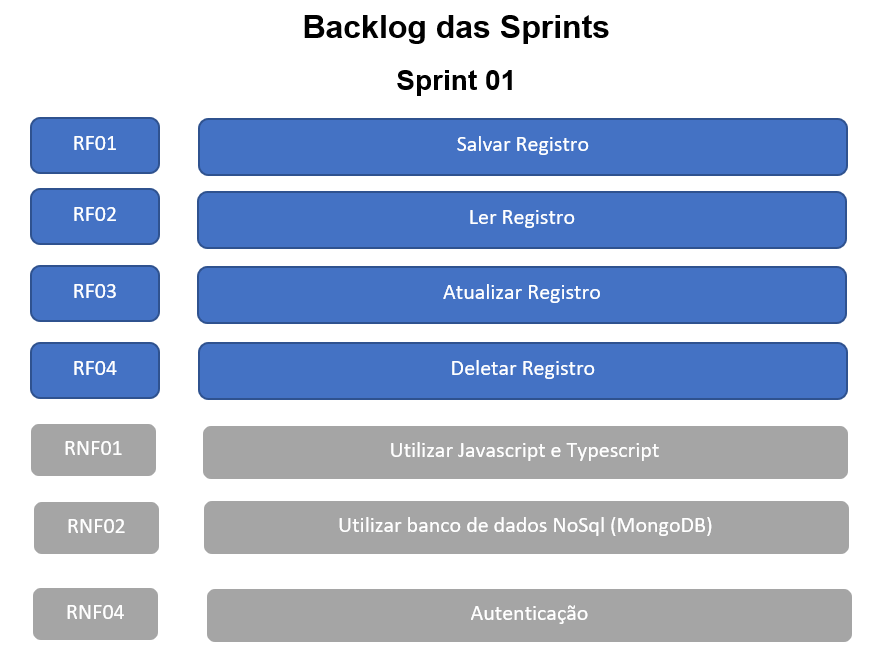


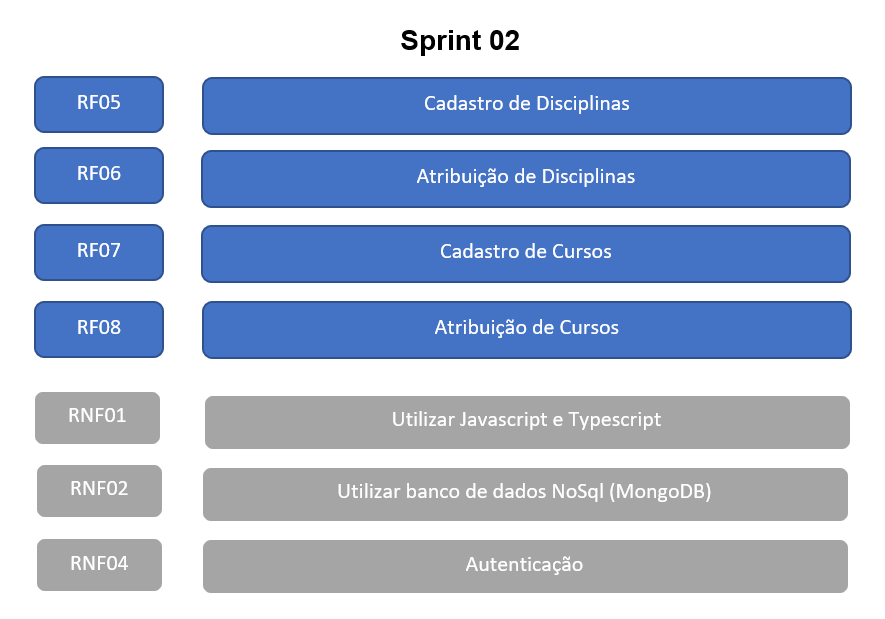
* 1. Requisitos não funcionais

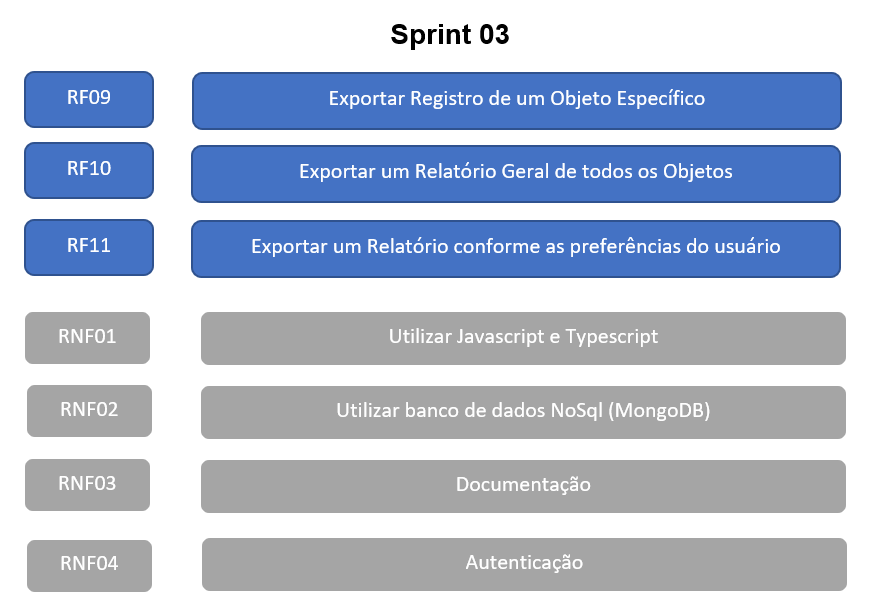


* 1. Planejamento das Sprints

O planejamento das Sprints foi pensado pelos integrantes do grupo para que o projeto fosse executado de maneira mais eficiente e visando a entrega conforme as datas determinadas:



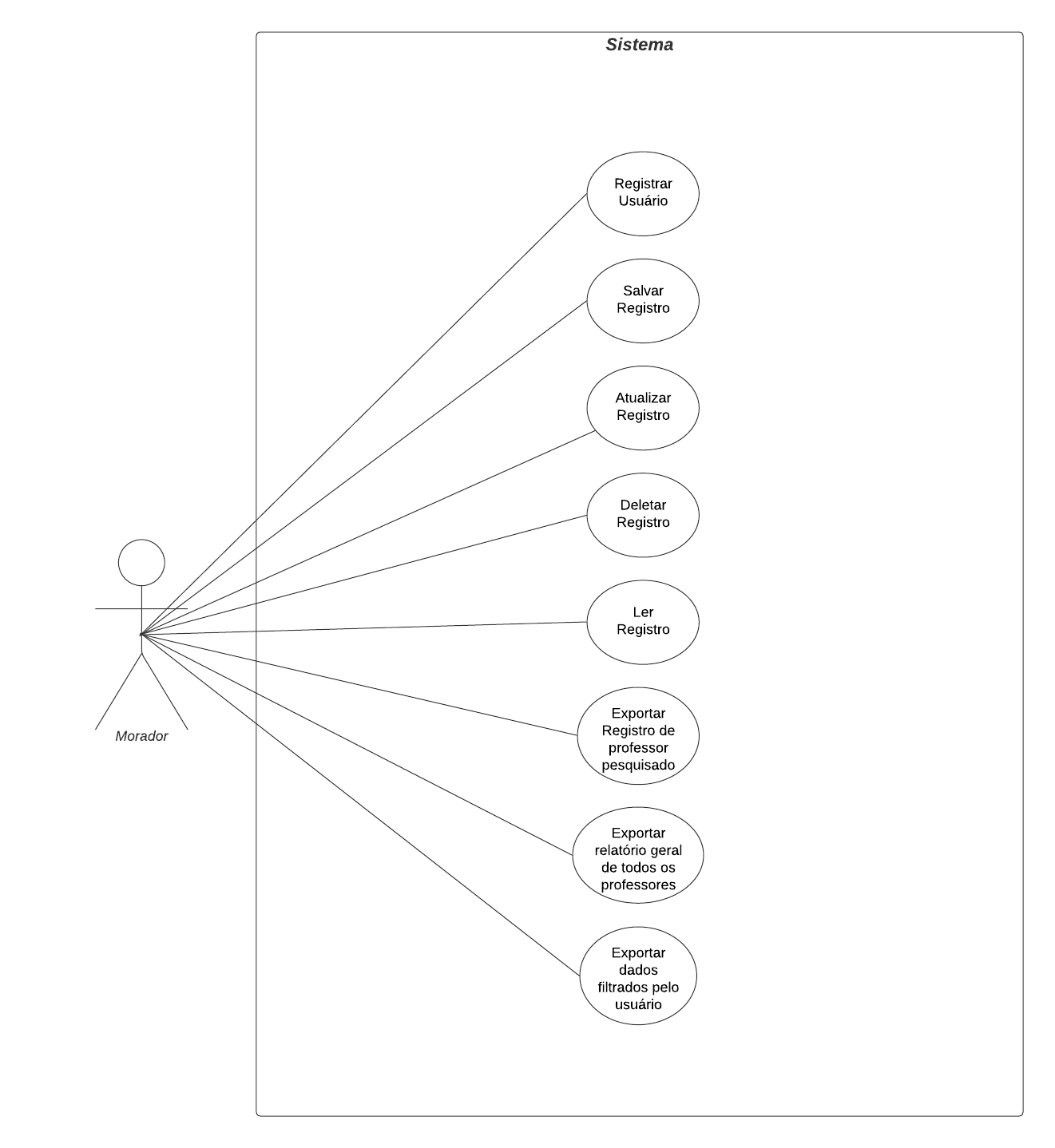




Um ponto no desenvolvimento do nosso projeto foram as reuniões para o acompanhamento e trabalho em conjunto do time para um desenvolvimento mais elaborado do projeto.

* 1. Diagramas de caso de uso

Durante as reuniões também foi desenvolvido um Diagrama de caso de Uso, que nada mais é do que a demonstração das funções do usuário do sistema.



1. - Projeto do Software

3.1 Tecnologias utilizadas

3.1.1 – Tecnologias para back-end e front-end:

* Para a criação do projeto foram utilizadas linguagem de marcação de texto, linguagem de folha de estilo, frameworks, linguagens de programação (Front-end, Back-end e linguagem de consulta de banco de dados) IDE, sistema de gerenciamento de banco de dados:
* HTML5: Linguagem de marcação utilizada para estruturar os elementos da página, como parágrafos, links, títulos, tabelas, imagens e até vídeos.
* CSS: Linguagem de estilos utilizada para definir cores, fontes, tamanhos, posicionamento e qualquer outro valor estético para os elementos da página.
* Bootstrap: Framework front-end que fornece estruturas de CSS para a criação de sites e aplicações responsivas de forma rápida e simples.
* MongoDB: é um sistema de gerenciamento de banco de dados (DBMS) não relacional, baseado em software livre, que utiliza documentos flexíveis em vez de tabelas e linhas para processar e armazenar várias formas de dados.
* Visual Studio Code: ambiente de desenvolvimento integrado (IDE do inglês Integrated Development Environment), ferramenta de desenvolvimento para editar o código, acessar um terminal, executar um script, debugar e compilar usando um único ambiente.
* Swagger: Trata-se de uma aplicação *open source*que auxilia desenvolvedores nos processos de definir, criar, documentar e consumir APIs REST.

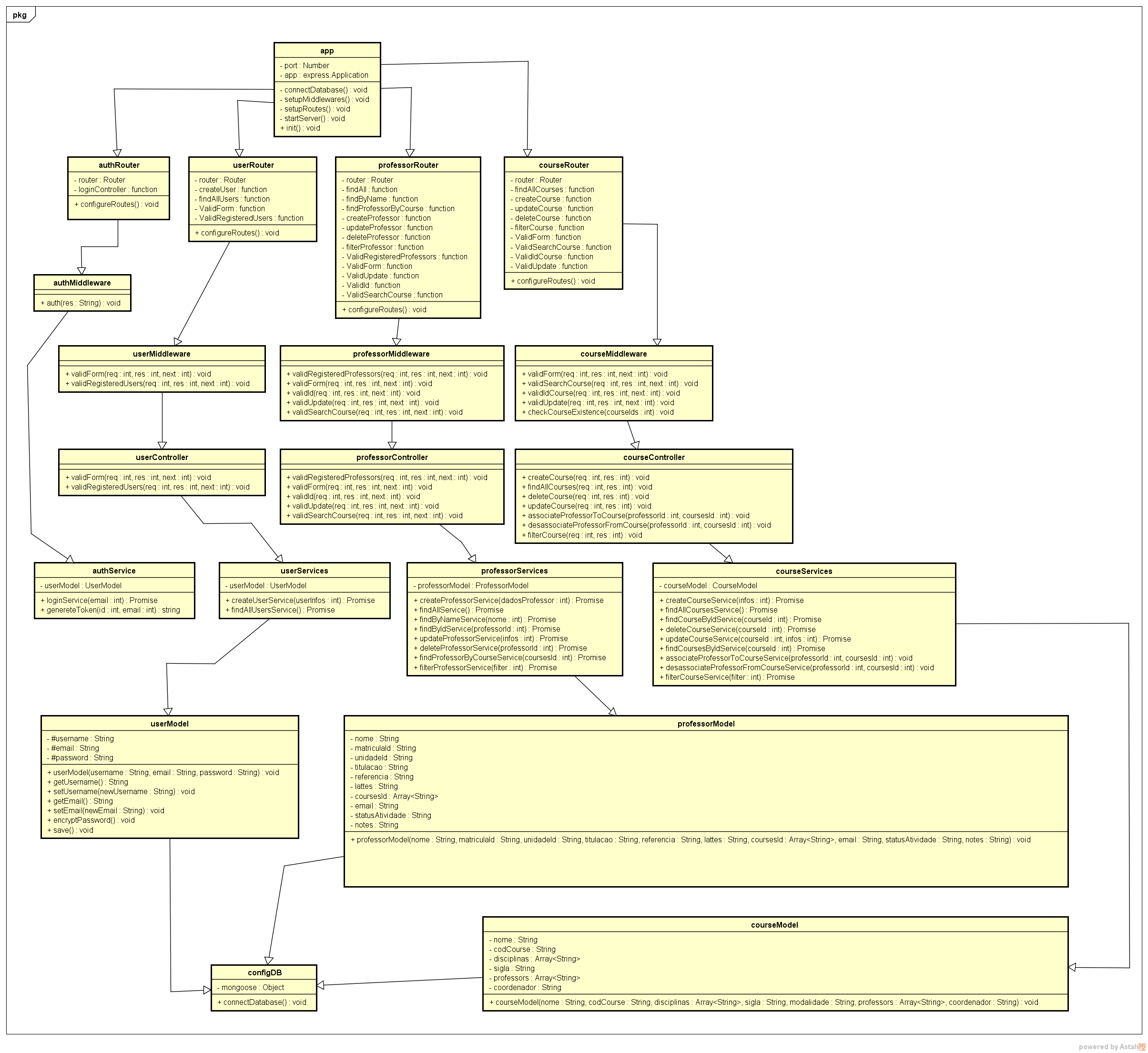
3.1.2 – Tecnologias para versionamento de códigos:

* Discord: é um aplicativo de voz sobre IP e comunicação textual. É uma aplicação gratuita que foi projetado inicialmente para comunidades específicas de jogos.
* Github: é uma plataforma de hospedagem de código-fonte e arquivos com controle de versão usando o Git.
* VS Code: Visual Studio Code: ambiente de desenvolvimento integrado (IDE do inglês Integrated Development Environment), ferramenta de desenvolvimento para editar o código, acessar um terminal, executar um script, debugar e compilar usando um único ambiente.

3.1.3 – Tecnologias para gerenciamento do time

* Figma: é um editor gráfico de vetor e prototipagem de projetos de design baseado principalmente no navegador web
* Discord: é um aplicativo de voz sobre IP e comunicação textual. É uma aplicação gratuita que foi projetado inicialmente para comunidades específicas de jogos.
* Trello: é a ferramenta visual que possibilita ao time o gerenciamento de qualquer tipo de projeto, fluxo de trabalho ou monitoramento de tarefas.

3.2 - Diagrama de Classes



3.2.1 - Descrição

O nosso projeto foi pensado e modelado para que os componentes do sistema fossem as classes, abaixo vou detalhar as classes, seus atributos e seus métodos:

**Classe app** - A classe representa a aplicação inicial e contém os métodos para a inicialização da aplicação, ela depende das classes de rotas.

Atributos: port: Number, app: Express.

Métodos: setupMiddlewares(): void, setupControllers(): void, setupRoutes(): void, setupConfigs(): void, start(): void.

Routers – As classes do modelo router, são as classes que representem as rotas especificas da aplicação, são 4 rotas ao todo:

**Classe authRouter** – Classe que representa a rota para a autenticação do login dos usuários.

Atributos: router: Router

Métodos: post(path: String, controller: Function): void.

**Classe userRouter** – a classe faz a validação dos dados de um formulário e faz a busca de usuários cadastrados.

Atributos: router: Router, createUser: Function, findAllUsers: Function, validForm: Function, validRegisteredUsers: Function.

Métodos: configureRoutes(): void.

**Classe professorRouter** – Essa classe define rotas para gerenciar recursos relacionados aos professores na aplicação.

Atributos: router: Router, findAll: function, findByName: function, findProfessorByCourse: function, createProfessor: function, updateProfessor: function, deleteProfessor: function, filterProfessor: function, validRegisteredProfessor: function, validForm: function, validUpdate: function, validId: function, validSearchCourse: function.

Métodos: configureRoutes(): void

**Classe courseRouter** – essa classe é responsável pelas rotas relacionadas aos cursos lecionados pelos professores na aplicação

Atributos: router: Router, findAllCourses: function, createCourse: function, updateCourse: function, deleteCourse: function, filterCourse: function, validForm: function, validSearchCourse: function, ValidIdCourse: function, ValidUpdate: function.

Métodos: configureRoutes(): void.

**Middleware** – As classes do modelo middleware são responsáveis por verificar e processar os dados antes de passar aos controladores.

**Classe authMiddleware** – Responsável por verificar dados do login

Métodos: authorize(): (req, res, next) => void.

**Classe userMiddleware** – classe responsável por validar as requisições relacionadas ao usuário da aplicação.

Métodos: validForm(req, res, next): void, validRegistredUser(req, res, next): void.

**Classe professorMiddleware** – classe responsável pela validação de cursos e professores relacionados a aplicação.

Métodos: validForm(req, res, next): void, validUpdate(req, res, next): void, validRegisteredProfessors(req, res, next): void, validId(req, res, next): void, validSearchCourse(req, res, next): void.

**Classe courseMiddleware** – Classe com função de auxiliar a validação dos cursos cadastrados na aplicação.

Métodos: validSearchCourse(req, res, next): void, validForm(req, res, next): void, validIdCourse(req, res, next): void, validUpdate(req, res, next): void, checkCourseExistence(courseIds): void.

**Controllers** – Classes do tipo controller recebem e processam as requisições das rotas para executar as regras de negócios.

**Classe userController** – Essa classe é responsável por lidar com as requisições relacionadas a usuários da aplicação.

Atributos: userService: Service.

Métodos: createUser(req, res): void, findAllUsers(req, res): void

**Classe professorController** – Essa classe é responsável por lidar com a requisições relacionadas a professores da aplicação

Atributos: professorService: Service

Métodos: createProfessor(req, res): void, findAll (req, res): void, findByName (req, res): void, updateProfessor(req, res): void, deleteProfessor (req, res): void, findProfessorByCourse (req, res): void, filterProfessor(req, res): void.

**Classe courseController** – Essa classe é responsável por lidar com as requisições relacionadas a cursos da aplicação.

Atributos: courseService: Service.

Métodos: createCourse(req, res): void, findAllCourses(req, res): void, deleteCourse(req, res): void, updateCourse(req, res): void, associateProfessorToCourse(req, res): void, desassociateProfessorFromCourse(req, res): void, filterCourse(req, res): void.

**Services** – As classes do tipo services são responsáveis por executar a lógica de negócio e interagir com o modelo dos dados.

**Classe authService** – classe que mantem as funções para gerar o login dos usuários e gerar token de autenticação JWT.

Atributos: userModel: Model

Métodos: loginService(email): Promise, generateToken(id, email) String.

**Classe userService** – classe que exporta serviços relacionados a criação de usuários.

Atributos: userModel: Model.

Métodos: createUserService(userInfos): Promise, findAllUsersServices(): Promise.

**Classe professorService** – essa classe exporta serviços relacionados a manipulação de professores da aplicação.

Atributos: professorModel: ProfessorModel.

Métodos: createProfessorService(dadosProfessor): Promise, findAllService(): Promise, findByNameService(nome): Promise, findByIdService(professorId): Promise, updateProfessorService(infos): Promise, deleteProfessorService(professorId): Promise, findProfessorByCourseService(coursesId): Promise, filterProfessorService(filter): Promise.

**Classe courseService** – essa classe exporta serviços relacionados a manipulação de cursos da aplicação.

Atributos: courseModel: CourseModel

Métodos: createCourseService(infos): Promise, findAllCoursesService(): Promise, findCourseByIdService(courseId): Promise, deleteCourseService(courseId): Promise, updateCourseService(courseId, infos): Promise, findCoursesByIdService(courseId): Promise, associateProfessorToCourseService(professorId, coursesId): void, desassociateProfessorFromCourseService(professorId, coursesId): void, filterCourseService(filter): Promise.

**Models** – As classes do tipo models representam a estrutura dos dados no banco de dados do projeto.

**Classe userModel** – essa classe define como deve ser a estrutura dos dados salvos do usuário no banco de dados da aplicação.

Atributos: username: String, email: String, password: String.

Métodos: save(): Promise.

**Classe professorModel** – essa classe define como deve ser a estrutura dos dados salvos de um professor no banco de dados da aplicação.

Atributos: nome: String, matriculaId: String, unidadeId: String, titulacao: String, referencia: String, lattes: String, coursesId: Array<ObjectId>, email: String, statusAtividade: String, notes: String.

Métodos: save(): Promise.

**Classe couseModel** – essa classe define como deve ser a estrutura dos dados salvos de um curso no banco de dados da aplicação.

Atributos: nome: String, codCourse: String, disciplinas: Array<String>, sigla: String, modalidade: String, professors: Array<ObjectId>, coordenador: String.

Métodos: save(): Promise.

**Classe configDB** – A classe configDB é responsável pela configuração da conexão do banco de dados da aplicação.

Atributos: mongoose: Object.

Métodos: connectDatabase(): void.

3.3 - Microsserviços

**3.3.1 - AUTH**

Esta rota é responsável pela autenticação do usuário, ou seja, o login do usuário no sistema. Essa rota verifica se o passado como parâmetro existe no banco de dados, se existir, faz a encriptação da senha passada e compara com a senha referente ao e-mail, se a comparação for equivalente, retorna True e permite o login no sistema, se não, bloqueia o usuário. O usuário logado gera um token para navegar durante 24h, a partir disso é necessário realizar o login novamente.

**PARÂMETROS:** e-mail (String) e senha (String).

**3.3.2 - PROFESSOR**

**GET – GetAll -** Faz a requisição de todos os professores cadastrados na collection *Professors*. Retornando todos os objetos json cadastrados com o adendo de substituir o *\_id* do curso no campo *course* pelo nome do curso e a sigla, endereçados na collection *Courses*. Esta rota é interceptada por um middleware que valida se há professores cadastrados na collection, se não tiverem, já retorna de prontidão um erro.

**GET – Filter -** Essa requisição faz uma busca dos professores cadastrados de acordo com o filtro passado como parâmetro, podendo ser nome, titulação ou cursos.

**PARÂMETROS:** Titulação (String) ou Nome (String) ou Cursos (String).

**GET – Professor by Course -** Essa requisição busca todos os professores de acordo com o *\_id* do curso passado como parâmetro, ele busca todos os professores relacionados aquele *\_id*.

**PARÂMETROS:** MongoDB ObjectId (*\_id*).

**POST** **– Criar professor -** Esse microsserviço faz a requisição para criar um objeto Professor na collection Professors. Antes de chegar ao controlador e seguir seu fluxo para o cadastro desse professor a rota é interceptada pelo middleware ValidForm, que valida o formulário das informações passadas para esse cadastro, se todas os campos forem enviados de forma correta, o professor é criado.

**PARÂMETROS:** Nome (String), Número de matrícula (String), Id da Unidade de Ensino em que leciona (String), Titulação (String), Referência (String), Link do currículo lattes (String), Id do Curso em que leciona (String), e-mail (String), Status da atividade do professor (Enum: Ativo ou Inativo), Observações/Anotações (String).

**PUT – Atualizar registro -** Faz a requisição para um update do professor, atualizando um ou mais campos de cadastro do objeto professor cadastrado na collection.

**PARÂMETROS:** Nome (String), Número de matrícula (String), Id da Unidade de Ensino em que leciona (String), Titulação (String), Referência (String), Link do currículo lattes (String), Id do Curso em que leciona (String), e-mail (String), Status da atividade do professor (Enum: Ativo ou Inativo), Observações/Anotações (String).

**DELETE – Deletar Professor -** Esse microsserviço faz a requisição para remover um objeto professor da collection *Professors.*

**PARÂMETROS:** MongoDB ObjectId (*\_id*).

**3.3.3 - CURSO**

**GET – Buscar todos os cursos cadastrados -** Esse microsserviço busca todos os cursos cadastrados na collection *Courses* no banco de dados. Essa rota é interceptada por um middleware que valida se há cursos cadastrados, se não houver retorna um erro.

**GET – Filter/Filtro -** Essa requisição filtra os cursos em até 3 parâmetros que pode ser coordenador, nome ou modalidade (Enum: “Presencial”,” EAD” ou “Híbrido”).

**PARÂMETROS:** Coordenador (String ) ou Nome do curso (String) ou Modalidade (String).

**POST – Cadastrar um novo curso -** Essa requisição realiza o cadastro do curso levando em conta todo o formulário por ele preenchido, sabendo-se que ao não enviar um campo corretamente o middleware retorna um erro *Bad Request.*

**PARÂMETROS:** Nome (String), Código do curso (String), Sigla (String), Modalidade (Enum: “Presencial”, “EAD” ou “Híbrido”), Disciplina (String), Professor (ObjectId ou Array de ObjectId), Coordenador (“String”).

**PUT – Atualizar um registro de um curso -** Esse microsserviço atualiza um cadastro de um curso, recebendo como parâmetro um ou até todos os campos de cadastro. Esta rota possui dois middlewares, um que valida o ObjectId do curso, passado como parâmetro, outro que valida o formulário com os campos a serem atualizados.

**PARÂMETROS:** ObjectId do Curso a ser atualizado (ObjectId) + Os campos de cadastro do POST.

**DELETE – Remove um registro de um curso -** Essa rota deleta um objeto json (registro) do banco de dados.

**PARÂMETROS:** ObjectId do Curso a ser removido.

**3.3.4 - USUÁRIOS**

**GET – Busca todos os usuários cadastrados -** Essa requisição busca todos os usuários cadastrados no banco de dados.

**POST – Cadastra um novo usuário -** Cadastra um novo usuário para utilizar o sistema.

**PARÂMETROS:** E-mail (String) e Senha (String).