Sistema integral de gestão acadêmica e comunicação estudantil Manthano

Fabio dos Santos Reszko Junior[1], Eduardo Tieppo[2]
[1,2]Instituto Federal do Paraná (IFPR) – Pinhais, PR – Brasil

s.reszkojr@gmail.com, eduardo.tieppo@ifpr.edu.br

Abstract. *The internet enabled remote communication and brought with it significant advancements, along with various challenges, particularly in distance education. The variety and dispersion of tools for academic management have created difficulties for educators and students, complicating organization and efficiency. This work proposes the creation of Manthano, a website that centralizes resources and functions for the distance education environment, aiming to unify and simplify the educational process and enhance the user experience.*

Resumo. *A internet possibilitou a comunicação remota e trouxe grandes avanços, mas também gerou problemas, especialmente na educação a distância (EAD). A variedade e dispersão de ferramentas para gestão acadêmica criaram desafios para educadores e alunos, dificultando a organização e eficiência. Este trabalho propõe a criação do Manthano, um website que centraliza recursos e funções para o ambiente EAD, buscando unificar e simplificar o processo educacional e melhorar a experiência dos usuários.*

1. Introdução

Com o surgimento da pandemia, diversas restrições sanitárias foram impostas, incluindo a proibição de reuniões e encontros presenciais. Isso desafiou instituições em todo o mundo a buscar formas alternativas de transmitir e compartilhar informações outrora presenciais. Diante dessa necessidade, uma série de propostas que utilizam a internet como meio de viabilizar essa adaptação surgiu, oferecendo soluções para a realização de reuniões e assembleias online.

Enquanto diversas empresas adotaram a modalidade de teletrabalho, escolas, universidades e outras instituições de ensino foram obrigadas a seguir um caminho similar. Assim, o ensino remoto emergencial (ERE) emergiu com a pandemia de covid-19, visando o prosseguimento das atividades educacionais de modo a virtualizar o ensino presencial com o uso das TDICs (Tecnologias Digitais de Informação e Comunicação) (COQUEIRO, 2021). O ERE surge como implementação extraordinária e temporária da educação a distância (EAD) com o objetivo de dar seguimento às atividades acadêmicas em circunstâncias que impedem a realização das atividades presenciais (PENTEADO, 2021). O EAD age como veículo que possibilita o acesso remoto à educação, tendo como objetivo específico proporcionar um meio em que avaliações, aulas e palestras possam ser realizadas por docentes e discentes por intermédio da internet. Sendo o EAD uma modalidade de educação remota, dispensa-se a necessidade de professores e alunos compartilharem o mesmo espaço físico simultaneamente (OLIVEIRA, 2020).

Como consequência disso, o discente adquire liberdade no que tange à organização de seu tempo de estudo de acordo com sua disponibilidade de horário. A

depender do método adotado por uma instituição, as atividades propostas pelos professores podem ser realizadas pelo aluno em momentos mais adequados às suas preferências de aprendizado, além de permitir a revisão do material fornecido pelos docentes por meios digitais (COQUEIRO, 2021). Dessa forma, o EAD destaca-se no cenário atual, pois se adapta às diferentes realidades de alunos que buscam formação por meio desta abordagem, tratando-se de um sistema que atende as necessidades de um público específico, atingindo-se cada vez mais segmentos (FARIA; SALVADORI, 2010) e diferindo-se de outras formas de ensino remoto pois, segundo Penteado (2021), "A EaD se distingue de outras formas de ensino remoto pela sua característica planejada e por exigir inovações de ordem pedagógica, didática e organizacional com metodologias, ambientes de aprendizagens, gestão e avaliação peculiares".

Assim, ao considerar-se o impacto e a versatilidade da educação a distância no contexto educacional, surge a necessidade de explorar como essas diferentes soluções não somente fornecem conteúdo educacional, como também recriam a experiência de aprendizado presencial por meio de uma variedade de recursos não-presenciais interativos.

Essas soluções são reproduzidas por meios digitais (videoaulas, conteúdos organizados em plataformas virtuais de ensino e aprendizagem, redes sociais, correio eletrônico, blogs, entre outros); por meio de programas de televisão ou rádio; pela adoção de material didático impresso com orientações pedagógicas distribuído aos alunos e seus pais ou responsáveis; e pela orientação de leituras, projetos, pesquisas, atividades e exercícios indicados nos materiais didáticos. A comunicação é essencial neste processo, assim como a elaboração de guias de orientação das rotinas de atividades educacionais não presenciais para orientar famílias e estudantes, sob a supervisão de professores e dirigentes escolares. (BRASIL, 2020).

Nesse raciocínio, o ensino a distância não se limita apenas a transferir o ensino tradicional para o ambiente virtual, bem como abre espaço para novas formas de aprendizado e colaboração. Enquanto a participação presencial de um aluno em uma sala de aula oferece-o interação com os colegas, realização de atividades avaliativas e esclarecimento de dúvidas, e a participação presencial do professor em uma sala de aula demanda tutoria, correção de avaliações, comunicação bilateral com os alunos a fim de esclarecer dúvidas e administração da sala de aula, fazer uso das TDICs por meio de plataformas EAD pode oferecer a mesma experiência de forma on-line com o Google Meet, para videochamadas; Google Forms, para formulários avaliativos; Gmail ou Whatsapp, para contato assíncrono; e o Google Classroom para gestão de notas/avaliações.

Entretanto, Cecílio e Reis (2016) enfatizam que o emprego das TDICs apresenta diversos desafios para os docentes no contexto do ensino a distância, especialmente no âmbito do ensino superior. Eles ressaltam a importância de estudos que investiguem a interação entre o ambiente digital e o presencial de uma sala de aula. De acordo com os autores, as TDICs contribuem para a diversificação das tarefas, aumentando a complexidade de trabalho docente e intensificando as atividades laborais, o que gera impactos negativos na saúde dos professores e, pois, impactos negativos na experiência de aprendizado dos alunos.

As tecnologias estão se transformando progressivamente de meio a um fim em si mesmas e exigem que os indivíduos e a sociedade se adaptem a elas; e tal adaptação se processa de maneira abrupta, gerando consequências desumanas para seus usuários (PUCCI; CERASOLI, 2010, p. 172-173).

Essas tecnologias crescem em nível exponencial, segundo Schlemmer e Zanela (2007), cuja velocidade acaba por acelerar os processos de mudança, gerando uma transformação tecnológica sobre as "categorias segundo as quais pensamos todos os processos", de acordo com Castells (1999). Dessa forma, surge um novo paradigma; denominado "paradigma da Complexidade" por Morin (1999) e Moraes (2003), requerendo um pensamento dialógico e multidimensional.

Ainda de acordo com Schlemmer e Zanela (2007), um EaD eficaz deve propiciar, a fim de se evitar o paradigma da complexidade, interação constante entre os sujeitos, as tecnologias e a informação, assim que insere-se em um novo contexto de aprendizagem em diferentes meios e metodologias. A criação de meios de comunicação escritas e faladas explora esse conceito de forma a destruir resistências existentes no processo de compartilhamento de informação pela *web*. Nesse processo, a abordagem mais comum utilizada por professores é a interconexão entre múltiplas plataformas de gestão virtual de salas de aula, algo que gera uma abordagem fragmentada e descoordenada, caindo no paradigma da Complexidade novamente.

Dessa forma, a necessidade de transição entre diferentes interfaces e ambientes digitais sobrecarrega alunos e educadores, aumentando a probabilidade de erros e inconsistências nos dados, comprometendo a integridade e confiabilidade das informações acadêmicas. A incompatibilidade de informações entre diferentes sistemas e a falta de padronização na integração dos dados levam a uma experiência complexa e desfavorável ao usuário, contribuindo para uma gestão escolar menos eficiente e coesa.

Para enfrentar esses desafios, emerge a necessidade de uma abordagem mais unificada na administração acadêmica, centralizando dados e informações em um único sistema. Neste projeto, propõe-se a implementação do Manthano, um website que centraliza diversas ferramentas, dados e informações em um único sistema, focando em usabilidade e clareza de uso. O Manthano, como plataforma integral, atenderá às diversas necessidades de administração exigidas por um professor, diminuindo os obstáculos e desafios associados à descentralização de informações e ferramentas no meio digital. Com suporte a videochamadas, chat por texto, programação e agendamento de avaliações, e administração de canais interativos e da sala de aula, essa solução mitigará os problemas de fragmentação e promoverá uma experiência de usuário mais fluida e eficiente para estudantes, docentes e pedagogos.

2. Métodos

O esquema que representa a arquitetura básica do Manthano está ilustrado na Figura 1.

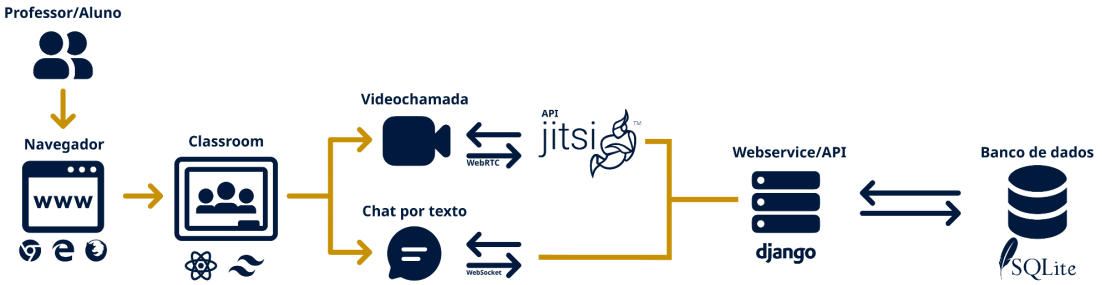

Figura 1. Arquitetura do projeto

Inicialmente, por conta da natureza *web* do Manthano, o usuário interage com a aplicação através do navegador, programa responsável pela renderização das páginas *web* que, por sua vez, exibe a interface inicial da *Classroom* ao estudante/professor.

Na *Classroom*, o usuário pode optar por escolher entre canais de voz — *VoiceChannels* — *e* canais de *chat* por texto — *Channels*; nestes, a comunicação é realizada através de uma conexão bilateral *WebSocket* persistente mantida entre o cliente e o servidor, enquanto naqueles realiza-se uma videoconferência, hospedada na plataforma *Jitsi* e renderizada na interface do Manthano de forma integrada. Ambos os tipos de canais possuem seus funcionamentos elucidados nas seções subsequentes.

O armazenamento de canais de voz, texto e suas mensagens ocorre por meio de uma conexão com o *Web Service*, implementado utilizando o *framework Python Django*, que executa requisições feitas pela área de *front-end* — interface esta que é utilizada pelo usuário para interagir com o sistema e é renderizada pelo navegador — e as retorna com uma resposta. A depender do tipo de requisição feita pelo *front-end*, o *Web Service* realiza operações de adição, edição e deleção de dados no banco de dados por meio do *Django ORM*, ou *Django Object-Relational Mapping*, que facilita essas operações abstraindo a manipulação de dados como objetos *Python*. O *Django ORM* comunica-se com o banco de dados *SQLite*, escolhido por ser a opção padrão da instalação do *Django* e de sua facilidade em termos de configuração e manutenção, excluindo a necessidade de um servidor de banco de dados dedicado.

2.1 Classroom

Denomina-se *Classroom* a interface pela qual o usuário (estudante) interage com os colegas (outros usuários) por meio de canais de texto, denominadas *Channels*. Ao criar-se uma *Classroom*, o usuário (professor ou aluno) possui o poder de escolher se esta será administrada por parte dos alunos ou dos professores. A administração de uma *Classroom* é feita por intermédio da moderação de canais de texto e, caso a opção de administração docente esteja habilitada, o professor poderá ver as mensagens que são enviadas pelos alunos dentro das *Channels*.

Uma *Classroom* é feita para representar digitalmente uma sala de aula física e contém os recursos mais importantes presentes dentro de um ambiente presencial. Por parte do aluno, interação audiovisual com outros estudantes; compartilhamento de arquivos e recursos estudantis; consulta às notas de avaliações dadas por professores; e

consulta à contagem de presenças em aulas. Por parte do professor, moderação da sala de aula — caso habilitada a opção de administração docente —; envio de atividades, avaliações e recursos estudantis; lançamento de notas e conceitos; interação com os alunos para mediação e resolução de dúvidas; e execução de chamadas.

Mediante lançamento de avaliações, um calendário será disponibilizado para todos os membros de uma sala de aula — docentes e estudantes — que tornará possível a visualização integral das atividades, provas e avaliações — e suas devidas notas —, presentes dentro de determinado período acadêmico, seja este bimestral ou trimestral. Dessa forma, possibilita aos professores melhor distribuição de atividades ao longo do período acadêmico e aos alunos melhor visualização dos eventos nele presentes, oportunizando uma melhor organização estudantil e melhor eficiência acadêmica.

Uma *Classroom* é criada após o registro de um novo usuário. A ele, é apresentada uma tela com duas opções: juntar-se a uma *Classroom* já existente ou criar uma *Classroom* nova. Caso o usuário opte por criar uma *Classroom* nova, uma requisição de criação de *Classroom* é executada para o *back-end* por parte do *front-end*. Após validação da requisição, o *back-end* cria no banco de dados um novo registro na tabela *Classroom* — utilizando a operação *objects.create* — com os dados informados pelo usuário, como o nome da *Classroom*, por exemplo.

2.2 *Channel*

Enquanto *Classrooms* representam salas de aula físicas, *Channels* têm seu papel focado na representação das matérias presentes em uma turma. Nesse raciocínio, dentro de uma Classroom, cria-se uma *Channel* para cada matéria existente na grade curricular da turma. É por seu intermédio que professores e alunos interagem direta ou indiretamente. Professores, independentemente da configuração de sua participação estar habilitada nas preferências de uma *Channel*, comunicam-se indireta e unilateralmente com alunos por meio do lançamento de notas brevemente escritas a fim de relembrar ou destacar algum assunto tratado em sala de aula. Essas notas ficam, para o estudante, fixas no cabeçalho de informações da *Channel* pertencente a matéria do respectivo professor, com uma cor destacada, indicando importância.

Uma *Channel* pode funcionar mediante texto ou videochamada. Enquanto o funcionamento das *Channels* de texto ocorre fazendo uso de chats de texto, tornando possível a comunicação por mensagens, *Channels* de videochamadas, denominadas *JitsiChannels*, possibilitam a comunicação entre alunos e professores por meio de salas de reunião *on-line*, ou videoconferências, onde a interação entre usuários acontece de maneira mais natural e intuitiva, com suporte a áudio, vídeo e mensagens de texto.

Para a criação de uma *Channel* por parte do usuário, deverá ser informado seu nome, que representará uma matéria presente na matriz curricular da turma, e o tipo de *Channel* desejado, podendo ser texto ou um *JitsiChannel*.

Além dessas funcionalidades, cada *Channel* é projetada para manter um registro organizado de todas as interações que ocorrem em seu interior. Isso inclui o armazenamento de mensagens de texto enviadas e qualquer outro tipo de comunicação realizada dentro da *Channel*. Esse registro facilita o acompanhamento das atividades pedagógicas e permite que tanto alunos como professores revisem informações trocadas

anteriormente, promovendo um ambiente de aprendizado contínuo. A estrutura do banco de dados do Manthano, gerenciado pela *ORM* do *Django*, garante eficiência no modo de armazenamento dessas informações, o que permite que o sistema escale de acordo com o crescimento de novas funcionalidades e com a adição de novas *Channels* ao longo do período acadêmico.

Dentro de uma *Channel* de texto, o usuário tem o poder de enviar mensagens de texto, editar suas próprias mensagens de texto e apagá-las. Caso um membro de uma *Channel*, estudante ou professor, acabe por ser também seu moderador, possuirá, adicionalmente, o poder de apagar mensagens de texto de outros estudantes, a fim de preservar a filosofia educacional proporcionada pela escola. Em contrapartida, é por meio das *JitsiChannels* que realizam-se videoconferências, assemelhando-se, assim, aos meios de interação presentes em uma sala de aula física. Para sua criação, utilizou-se a *API* da plataforma *Jitsi Meet*, integrando-a ao sistema Manthano. Funções como suporte à áudio, vídeo, *chat* de texto e moderação para moderadores estão presentes em *Channels* de videochamadas.

2.3 *JitsiChannels* e a *API Jitsi*

As videochamadas são possíveis por conta da conexão no sistema Manthano com a *API Jitsi Meet*, especificamente o serviço *Jitsi as a Service (8x8, 2024)*, que oferece a integração de uma sala de videoconferência dos servidores da *Jitsi* dentro de um componente *HTML*, proporcionando menos processamento por parte do servidor hóspede do Manthano. Para que essa conexão seja feita, foi criada uma *endpoint* no *back-end* do Manthano cuja responsabilidade é criar um *token* de validação do tipo *JWT* a partir de chaves privadas proporcionadas pelo painel de desenvolvedor *JaaS*. A parte do cliente, então, realiza uma requisição *HTTP* para esse *endpoint* a fim de integrar o *token* de autenticação em um componente *HTML* da parte de *front-end*. Dessa forma, a abertura da *JitsiChannels*, o cliente realiza uma requisição *HTTP* aos servidores *Jitsi* solicitando uma abertura de sala *on-line* de videoconferência passando o *token* recebido do *back-end* como cabeçalho. Caso os servidores *Jitsi* validem a requisição, uma sala de videoconferência será aberta e o componente *HTML* presente no cliente, integrado à *JitsiChannel,* a apresentará.

Ao acessar uma *JitsiChannel*, o usuário é apresentado a uma interface de videoconferência que inclui um conjunto abrangente de funcionalidades. Entre elas, estão a auto-visualização do orador ativo, a possibilidade de fixar a visualização de qualquer participante na tela, controle sobre microfone e câmera e compartilhamento de tela para apresentação de recursos visuais. A *API Jitsi* não apenas facilita a criação e gerenciamento das *JitsiChannels,* mas também oferece flexibilidade para ajuste de configurações, parâmetros das reuniões e permissões.

2.4 *WebSocket* e *chat* por texto

Ao abrir um *Channel* de texto, o cliente *web* realiza uma requisição *HTTP GET* ao servidor solicitando todas as mensagens presentes na *Channel* aberta. Ao receber essa solicitação, o *back-end* valida a requisição e realiza a operação *objects.get* utilizando o *Channel* como parâmetro a fim de filtrar as mensagens desejadas. Ao fim da operação, o servidor retorna ao cliente todas as mensagens requisitadas. Entretanto, o

envio e recebimento de mensagens posterior à requisição inicial feita na abertura da *Channel* é realizada com uma conexão *WebSocket*.

A conexão *WebSocket* é responsável pelo envio, recebimento e transmissão de mensagens em *Channels* de texto. Esse protocolo foi escolhido por conta de sua dualidade conectiva, permitindo comunicação bilateral simultânea entre a parte do cliente e a parte do servidor. Dessa forma, a capacidade do *WebSocket* de manter uma conexão persistente entre o cliente e o servidor possibilita que mensagens sejam enviadas e recebidas instantaneamente, descartando a necessidade de requisições *HTTP* repetidas pelo uso da técnica *long polling*, que realiza repetidas requisições *HTTP* por parte do cliente durante determinado período de tempo, como meio segundo, o que gera latência e causa significativa sobrecarga por parte do servidor.

Instancia-se um servidor *WebSocket* à parte na área do *back-end* com a biblioteca *django-channels* em uma *URL* que contém informações de identificação de ambas a *Classroom* e a *Channel*. Essa biblioteca permite tratamento de eventos enviados e recebidos pelo servidor de modo a realizar ações específicas dependendo do tipo de evento. Analogamente, na área de *front-end*, um cliente *WebSocket* é instanciado para o *Channel* de texto atualmente aberto na *Classroom*, o qual é responsável por enviar e receber dados do servidor *WebSocket* instanciado no servidor. Ao clique do botão "enviar" pelo usuário na caixa de texto, o cliente *WebSocket* envia a mensagem nela escrita ao servidor *WebSocket* disparando, neste, o evento *message*. Por conseguinte, o servidor gera um registro no banco de dados na tabela *Message* com a operação *objects.create* passando informações sobre a mensagem recebida. Posteriormente, o servidor envia a mensagem recebida de volta à *Channel* de modo a transmiti-la a todos os clientes *WebSockets* conectados cuja *Channel* é a mesma da mensagem, incluindo o remetente.

2.5 Web Service/API/Back-end

O *Web Service* atua como o componente central do *back-end* que integra todas as funcionalidades descritas até o momento. Sua principal responsabilidade é unificar os diversos módulos e serviços necessários pelo *front-end* e garantir uma comunicação eficiente entre eles. No contexto do sistema Manthano, a configuração de *URLs*, descritas no arquivo *urls.py*, define as diferentes rotas que serão acessadas pelo *front-end* e quais trechos de códigos devem ser executados, descritos no arquivo *views.py*.

Cada módulo do *back-end*, denominado, no ecossistema *Django*, de *app*, possui seu próprio arquivo de rotas e *views*. Para o Manthano, criaram-se três *apps* que coordenam o funcionamento do sistema: *authentication*, que gerencia a parte de cadastro, *login*, re-autenticação de *tokens JWT* e configuração de conta; *classroom*, que gerencia o funcionamento das *Classrooms*, como criação, edição e deleção de *Classrooms*, *Channels, JitsiChannels* e mensagens; e *manthano_backend*, que configura as rotas raizes do Manthano, como */auth* e */classroom,* para qual módulo cada rota será direcionada e as configurações principais da *API*.

O arquivo de configurações da API, situado no arquivo *settings.py*, configura o funcionamento de todo o projeto Django. Ele define diversas configurações essenciais, como segurança, *apps* instalados, bancos de dados, e muito mais.

Primeiramente, ele define a estrutura do projeto, incluindo a localização do diretório base (*BASE_DIR*) e o *SECRET_KEY*, que é utilizado para fornecer criptografia e deve ser mantido em segredo. Para evitar risco de vazamentos dessa chave em repositórios *Git* online, utilizou-se a biblioteca *django-decouple* para que outro arquivo, incluído na lista *.gitignore*, hospedasse a chave secreta. No *settings.py*, então, importa-se o arquivo com a chave secreta fazendo uso da biblioteca *django-decouple* com o nome reservado *config*.

Os *apps* instalados, especificados na variável *INSTALLED_APPS,* incluem aplicativos padrão do *Django*, como *django.contrib.admin* e *django.contrib.auth,* além dos aplicativos específicos do Manthano, como *authentica*tion *e classroom*. De forma adicional, a variável *MIDDLEWARE* inclui as diversas camadas de *middleware* que interceptam as requisições *HTTP* garantido a segurança e o funcionamento adequado da aplicação. Por exemplo, o *CorsMiddleware* permite que a aplicação aceite requisições de qualquer origem, o que é útil para fins de *debug*, mas pode precisar ser restrito em ambiente de produção.

No que concerne às conexões *WebSocket*, a biblioteca *django-channels* necessita ter seu uso definido pela variável *CHANNEL_LAYERS*. No caso do Manthano, essa configuração está definida para utilizar um canal de memória *in-memory*, que permite a comunicação em tempo real em conexões *WebSocket*, crucial para o funcionamento das mensagens instantâneas.

A configuração de banco de dados também é definida no arquivo *settings.py* do *manthano_backend*, definida pela variável *DATABASES*. O banco de dados escolhido foi o padrão dado pelo sistema *Django*, *SQLite3*, situado no arquivo *db.sqlite3.* O preenchimento e gestão do banco de dados são feitos automaticamente pelo Manthano, que atua como intermédio entre o arquivo *models.py,* responsável pela criação de tabelas como classes *Python*, e a *engine* do banco de dados escolhida.

Por fim, o *settings.py* também define a aplicação *ASGI — Asynchronous Server Gateway Interface*, um padrão para lidar com requisições assíncronas, como conexões *WebSocket —* e a aplicação *WSGI — Web Server Gateway Interface*, uma especificação para a comunicação entre servidores web e aplicações *Python*, usada para requisições síncronas. A aplicação *ASGI* é executada no arquivo *asgi.py*, também situado no *app manthano_backend*, e, dentro dele, o objeto *ProtocolTypeRouter* roteia requisições *WebSocket* para o arquivo *routing.py* presente no *app classroom*.

O *app authentication*, responsável pela parte de *login*, cadastro, autenticação e configuração de conta dos usuários, gira em torno do sistema de credenciais *JWT*, ou *JSON Web Token*, que é um padrão aberto, descrito na *RFC 7519*, que define uma maneira compacta e independente de transmitir informações com segurança entre partes como um objeto JSON. Esses *tokens* são administrados pela biblioteca *djangorestframework-simplejwt*, que disponibiliza *views* para atualização e verificação de *tokens* nos caminhos */token/refresh* e */token/check*, respectivamente.

Dois *tokens* são criados no momento em que o cliente envia para a *API*, no caminho */token*, um *email* e senha válidos cuja presença se encontra no banco de dados. Esse caminho é gerenciado pela *view MyTokenObtainPairView*, presente no arquivo *views.py* do *app authentication,* que serializa os dados utilizando um serializador da biblioteca *djangorestframework-simplejwt*. A *view* retorna ao cliente dois *JWTs*, um de autenticação, responsável pela validação de requisições feitas pelo cliente, e outro de atualização, responsável por revalidar o *token* de autenticação no momento em que expira. O tempo de expiração dos *token* de autenticação e de revalidação são configurados no arquivo *settings.py* no *app manthano_backend.* No momento em que o *token* de revalidação expira, o usuário é obrigado a iniciar sessão novamente, gerando novos *tokens*.

De forma complementar, a biblioteca *djangorestframework-simplejwt* também é responsável pela criação de um *middleware*, um interceptador de tráfego, que autentica requisições feitas por clientes em *views* protegidas com a classe de permissão *IsAuthenticated*. Esse *middleware* verifica se o *token* enviado pelo cliente na variável *request* do *Django* ainda é válido. Caso não seja, o trecho de código da *view* protegida não é executado e uma resposta *HTTP* é enviada ao cliente com o *status code 403 Forbidden*, indicando o recebimento da requisição feita pelo usuário por parte do servidor mas negando sua execução.

O *app classroom* é responsável pela criação, edição e resgate de informações de *Classrooms, Channels*, *JitsiChannels* e mensagens. Suas rotas, definidas no arquivo *urls.py*, contém os nomes *jitsi_channel, channel* e *message* e a operação a ser realizada pelo cliente dependerá da presença do sufixo de operação *add,* para adição de registros, *edit*, para edição e *delete*, para deleção de registros. Adicionalmente, o arquivo de rotas também contém as rotas */create*, */join* e */user*, responsáveis pela criação, entrada de usuários e verificação de filiação de usuários em *Classrooms,* respectivamente. No que concerne à segurança, requisições feitas pelo cliente em qualquer rota do *app classroom* necessita autenticação, indicando a presença da classe de permissão *IsAuthenticated* em suas *views*.

Ademais, como dito antes, o módulo *classroom* é responsável por gerenciar requisições *WebSocket* que a *API* recebe. A configuração das rotas *WebSocket* é definida no arquivo *asgi.py,* no *app manthano_backend*, onde as requisições *HTTP* e *WebSocket* são discernidas direcionando requisições *HTTP* para o aplicativo *Django* padrão, enquanto as requisições *WebSocket,* antes de serem enviadas ao arquivo *routing.py*, são tratadas através do *middleware TokenAuthMiddleware*.

O *TokenAuthMiddleware,* definido no arquivo *token_auth.py* no *app manthano_backend,* é responsável por validar o token *JWT* presente nas requisições *WebSocket* e autenticar o usuário correspondente. Caso o *token* seja inválido ou o usuário não esteja autenticado, a conexão *WebSocket* é fechada automaticamente. Sua implementação foi necessária pois a biblioteca responsável pela autenticação JWT lida apenas com conexões HTTP, exigindo um middleware de autenticação específico para conexões WebSocket.

As rotas específicas para as conexões *WebSocket* são definidas dentro do arquivo *routing.py* na variável *websocket_urlpatterns*, onde cada rota é associada a um *consumer*

específico que lida com mensagens e eventos da sala de aula. Um *consumer* é uma classe pertencente ao *Django Channels* que gerencia requisições *WebSocket* entre o servidor e os clientes e assimila-se a uma *view* do *Django*. Um *consumer* define como lidar com eventos de entrada e saída, como recepção de mensagens, abertura e fechamento de conexões e transmissão de dados. O único *consumer* presente na aplicação é o *ChannelConsumer*, presente no arquivo *consumers.py* do módulo *classroom*.

Quando um usuário se conecta, o *ChannelConsumer* adiciona-o ao grupo de canais correspondente à sala de aula, permitindo que as mensagens sejam enviadas para todos os membros da sala de forma simultânea. Este processo envolve a autenticação do usuário e a verificação de sua associação com a *Classroom*, garantindo que apenas usuários autorizados possam acessar e interagir na *Channel*. Além disso, o *ChannelConsumer* gerencia a lógica de eventos, incluindo a criação, modificação e exclusão de mensagens, assegurando que todas as interações sejam sincronizadas e corretamente transmitidas para todos os participantes em tempo real.

2.6 Banco de dados

O gerenciamento do banco de dados na *API* Manthano é realizado por meio de modelos definidos nos arquivos *models.py* presentes em ambos os *apps classroom* e *authentication*, que se traduzem em tabelas no banco de dados. Assim, faz-se uso do *Django ORM*, utilizando-o para facilitar a interação com essas tabelas, abstraindo as operações de *SQL* e permitindo manipulações de dados através de operações de alto nível.

No módulo *classroom*, existem os seguintes modelos (ou *models*):

1. *Classroom*: Representa uma sala de aula com atributos como nome, código único, descrição, cronograma e timestamps de criação e atualização. As relações entre salas de aula e outros elementos são estabelecidas através de chaves estrangeiras.
2. *Channel*: Define canais de texto dentro de uma *Classroom*, associados a uma *Classroom* específica por meio de uma chave estrangeira. Esse modelo permite a organização de comunicações dentro de uma sala.
3. *JitsiChannel*: Similar ao *Channel*, mas para integração com a plataforma *Jitsi*, permitindo a criação de canais de vídeo dentro de uma sala de aula.
4. *Message*: Armazena mensagens enviadas em canais, associadas a um usuário e a um canal específico. Inclui atributos para texto da mensagem, data e uma flag para mensagens editadas.

No módulo *authentication,* existem o seguintes modelos:

1. *ManthanoUser*: Estende o modelo padrão de usuário do *Django* para incluir campos adicionais específicos, como email acadêmico e status de administrador. Relaciona-se com *Classrooms* através de uma chave estrangeira, estabelecendo a associação entre usuários e suas respectivas salas.
2. *Student*: Representa estudantes como instâncias de *ManthanoUser* com um campo adicional para matrícula. Estabelece uma relação um-para-um com o modelo *ManthanoUser*.

3. *Professor*: Similar ao *Student*, mas para professores, incluindo um campo para o posto acadêmico e uma relação muitos-para-muitos com o modelo *Subject*, permitindo a vinculação de professores a múltiplas disciplinas e vice-versa.
4. *Profile*: Define perfis de usuários, incluindo imagens de perfil e fundo, além de uma descrição opcional. Estabelece uma relação um-para-um com o modelo *ManthanoUser*.

3. Resultados

As subseções apresentadas a seguir fornecem uma descrição detalhada de cada uma das subseções homônimas da seção anterior, Métodos. Cada subseção é discutida de forma abrangente de modo a abordar seus elementos específicos e as metodologias empregadas, oferecendo uma visão aprofundada de cada etapa do processo descrito. Esse detalhamento visa proporcionar uma compreensão completa das técnicas utilizadas no desenvolvimento do Manthano assim como os procedimentos adotados.

3.1 Classroom

Ao abrir uma *Classroom*, será apresentada ao usuário uma tela contendo: i) uma coluna lateral exibindo os usuários membros da *Classroom*, com sua responsabilidade — professor ou estudante— indicada por um símbolo — capelo para estudantes e um livro aberto para professores; ii) um painel lateral que exibe o nome da *Classroom*, os *Channels* e *JitsiChannels* a ela pertencentes e um botão, representado pelo ícone de adição, cujo clique exibe uma janela modal solicitando o nome da *Channel*/*JitsiChannel* a ser criada; e iii) uma área cuja função é exibir o conteúdo da *Channel*/*JitsiChannel* selecionada. Caso nenhuma seja selecionada, uma tela acolhendo o usuário será apresentada oferecendo diferentes escolhas a serem tomadas.

O conteúdo gráfico da tela inicial de uma *Classroom* está representada na Figura 2, criada por meio de uma captura de tela do sistema em funcionamento.

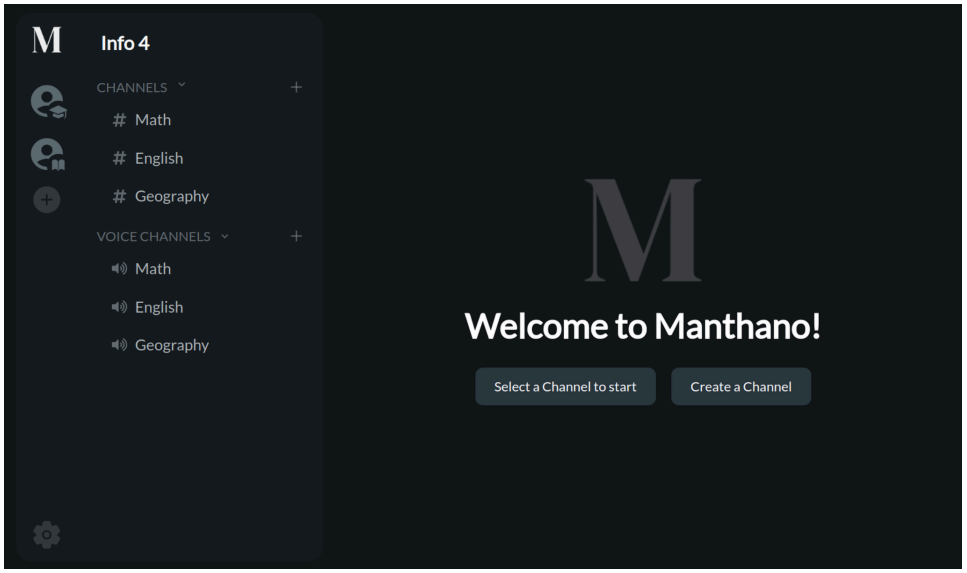

Figura 2. Visão da tela inicial da *Classroom*

3.2 *Channel*

Uma *Channel* é exibida à seleção de uma das opções presentes na seção *CHANNELS* do painel lateral pelo usuário. Nela, existe: i) um cabeçalho com um botão de expandir/retrair o painel lateral, uma cerquilha indicando a característica textual da *Channel*, e seu nome, geralmente representando uma matéria da turma; ii) a área de mensagens, onde são exibidas todas as mensagens, representadas pelo nome do usuário remetente, sua foto de perfil e o texto da mensagem em si, enviadas pelos usuários na *Channel* selecionada; iii) a área de envio de mensagem, cuja entrada de texto se dá pela caixa de texto e o envio da mensagem ocorre pelo botão cujo símbolo representa um avião de papel; e iv) a área miscelânea, cujas funções são de envio de arquivos, envio de *GIFs*, seleção de *emojis* e marcação de usuário.

A exibição de uma *Channel* de texto contendo mensagens de um usuário está representada na Figura 3, logo abaixo.

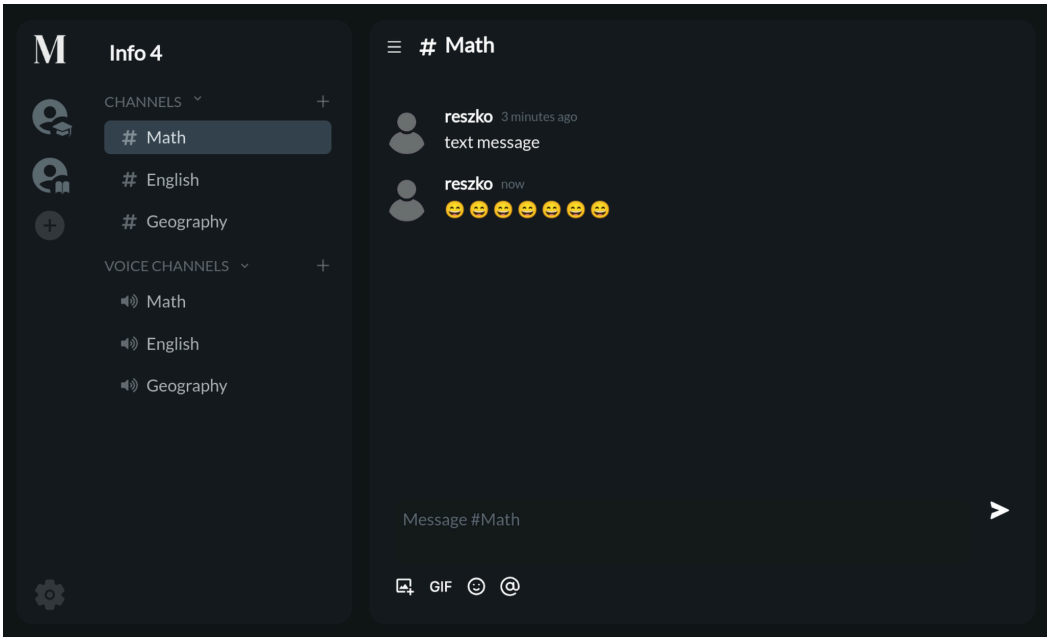

Figura 3. Visão da *Channel*

3.3 *JitsiChannels* e a API Jitsi

JitsiChannels são exibidas à seleção do usuário de uma das opções presentes na seção *VOICE CHANNELS* do painel lateral. Elas funcionam como salas de videoconferência e possuem diversas funcionalidades remotas. A interface de uma *JitsiChannel* inclui: i) auto-visualização do orador ativo: a tela automaticamente foca no participante que está falando no momento, garantindo que os usuários na sala presentes possam acompanhar quem está contribuindo com a conversa. Além disso, o usuário tem a opção de clicar em qualquer participante para fixar a visualização de seu vídeo; ii) controle de microfone e câmera: os participantes têm a flexibilidade de ativar ou desativar seus microfones e câmeras conforme necessário; iii) compartilhamento de tela:

o usuário pode compartilhar sua tela ou janela do navegador com outros participantes da reunião a fim de facilitar a troca de informações visuais; iv) *chat* de texto a parte: além da comunicação por vídeo e voz, os usuários podem enviar mensagens de texto durante a reunião pela própria interface do *JitsiChannel*; v) levantar/abaixar a mão: essa função permite ao usuário indicar para outros participantes da reunião seu desejo de falar, facilitando a organização da reunião, algo particularmente útil em reuniões de grande escala, onde é necessário o controle mais rígido de quem possui a palavra; vi) *layout* da reunião: a interface permite ao usuário personalizar o *layout* da interface da reunião de modo a atender melhor suas necessidades; vii) modo *push-to-talk*: essa funcionalidade permite ao usuário ativar o microfone apenas ao clique de um botão, reduzindo o ruído de quando não está falando; viii) reprodução de vídeos do *YouTube*: é possível iniciar a reprodução de um vídeo no *YouTube* para todos os participantes presentes na reunião; e ix) encerrar a reunião.

Todas essas funcionalidades, ilustradas na Figura 4 abaixo, estão presentes como botões na interface da *JitsiChannel*. Complementarmente, a *JitsiChannel* possui um cabeçalho com um botão de expandir/recuar o painel lateral, um símbolo indicando o tipo da *JitsiChannel* — neste caso, um auto-falante — e seu nome.

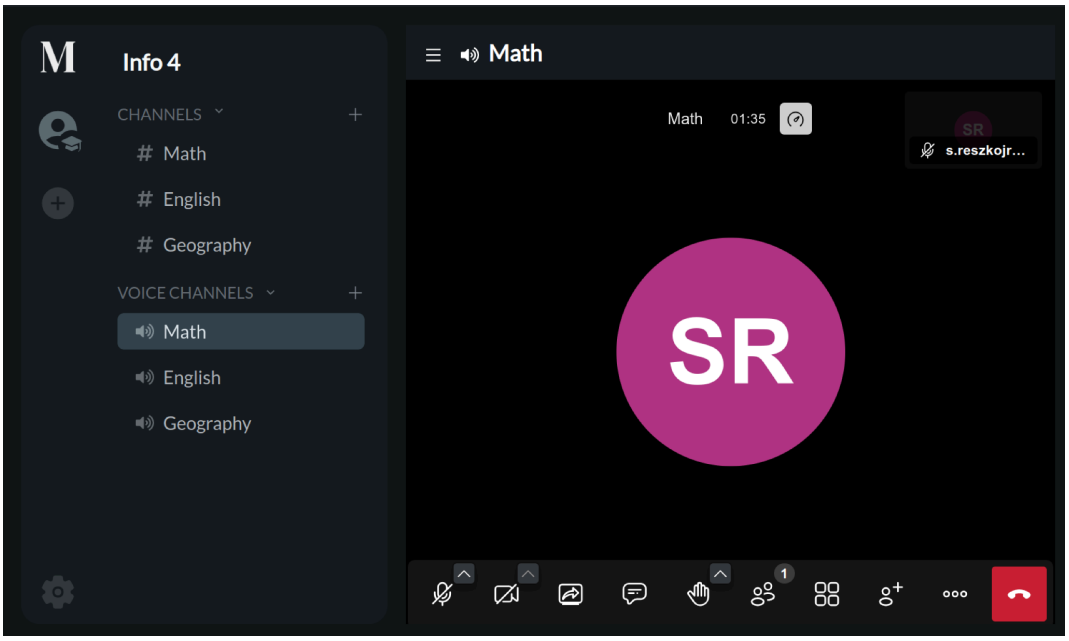

Figura 4. Visão da *JitsiChannel*

A criação de uma sala de videoconferência ocorre por meio da *API Jitsi*, que permite integração de reuniões *Jitsi* no Manthano. O processo inicia-se com a geração de um *JSON Web Token* específico para cada participante da reunião. Esses *tokens* são gerados no *backend* utilizando uma chave privada associada a uma chave pública previamente cadastrada no painel de configuração do desenvolvedor da plataforma *Jitsi*. O *JWT* inclui informações como o identificador único do usuário, permissões (como a possibilidade de ser moderador) e o nome da sala da reunião.

Analogamente, no momento em que um usuário tenta acessar uma *JitsiChannel* na interface da *Classroom*, um *token JWT* é requisitado para o *backend* para ser, então, transmitido aos servidores da *Jitsi*. Por meio dessa validação, a *API Jitsi* assegura que o usuário tem a autorização adequada para ingressar na sala da reunião.

Além do processo de autenticação via *JWT*, a *API Jitsi* permite uma personalização extensiva das *JitsiChannels* dentro do Manthano. Por meio de diversos parâmetros configuráveis pelo desenvolvedor, é possível ajustar aspectos como o tempo de duração das reuniões, as permissões padrão de novos participantes e a necessidade de autenticação adicional para funções específicas, como gravação da sessão, se suportado pelo plano *Jitsi as a Service* configurado, e moderação. Essa flexibilidade torna o sistema adaptável às necessidades de diferentes instituições educacionais,que podem exigir níveis variados de controle e segurança em suas videoconferências. Ademais, a integração com a *API Jitsi* possibilita atualizações contínuas e a inclusão de novas funcionalidades, fazendo com que as *JitsiChannels* do Manthano mantenham-se alinhadas com as melhores práticas de desenvolvimento de arquitetura de *software*.

3.4 *WebSocket* e *chat* por texto

A implementação do sistema de chat por texto do Manthano por meio de *WebSockets* demonstrou uma significativa melhoria na abstração da infraestrutura baseada na biblioteca *Django Channels*, a qual proporcionou uma comunicação assíncrona, provendo que as mensagens entre usuários fossem entregues quase instantaneamente a todos os participantes cuja *Channel* da mensagem se encontra aberta. Esse comportamento é particularmente importante em ambientes educacionais, onde a rapidez na troca de informação pode impactar diretamente na qualidade do ensino e na interação entre estudantes e professores. A integração do *WebSocket* com a lógica de autenticação *JWT* também assegurou que somente usuários devidamente autenticados pudessem participar das conversas, mantendo a segurança e a privacidade das interações.

Ademais, o uso de um canal de memória *in-memory* para gerenciamento de conexões *WebSocket* revelou-se como uma solução eficiente, principalmente em ambiente de desenvolvimento. Essa escolha permitiu uma rápida configuração e testes, evidenciando a flexibilidade da arquitetura do Manthano. A capacidade do sistema de lidar com múltiplas conexões simultâneas sem degradação perceptível no desempenho foi um dos aspectos mais notáveis, demonstrando a robustez do sistema. Mesmo em cenários onde múltiplos usuários estavam ativos em uma mesma *Channel*, enviando e recebendo mensagens de forma contínua, o sistema manteve um nível elevado de responsividade.

No contexto das funcionalidades de *chat* por texto, a arquitetura desenvolvida foi fundamental para garantir a entrega confiável das mensagens e a correta sincronização entre todos os clientes conectados. A lógica de grupos implementada no *ChannelConsumer* permitiu que todas as mensagens fossem disseminadas para todos os membros presentes em determinada *Channel* de maneira eficaz. Este comportamento foi crucial para preservar a coesão das discussões em grupo, facilitando a colaboração e o engajamento dos participantes, A precisão e a integridade das mensagens foram

mantidas durante toda a comunicação, sem registros de perda ou desordem na sequência das mensagens.

Além disso, a validação do *token JWT* durante o estabelecimento da conexão *WebSocket* também provou ser um mecanismo eficaz de segurança, protegendo o sistema contra acessos não autorizados. Durante os testes, casos em que *tokens* expirados ou inválidos foram utilizados para tentativas de conexão ao *chat* resultaram em uma desconexão imediata, conforme esperado. Essa implementação não apenas reforçou a segurança, mas também destacou a importância de garantir que apenas usuários legítimos tenham acesso às funcionalidades sensíveis do sistema, como o envio e recebimento de mensagens em tempo real. A eficácia desse sistema de autenticação assegurou a conformidade com as melhores práticas de segurança em ambientes de comunicação online.

3.5 *Web Service/API/Back-end*

O *Web Service* do Manthano atua como o núcleo central do *backend*, assegurando diferentes funcionalidades, como autenticação, gestão de salas de aula e comunicação em tempo real, operassem de maneira integrada e eficiente. Durante os testes de desempenho, observou-se que a configuração das rotas no arquivo *urls.py*, em conjunto com as *views* em *views.py*, permitiu uma navegação fluida e uma resposta rápida às solicitações do *front-end*.

Os três apps principais que compõem a arquitetura do sistema Manthano — *authentication*, *classroom* e *manthano_backend* — desempenharam papéis fundamentais na segregação eficiente de diferentes responsabilidades dentro da aplicação. O *app authentication*, que foi projetado especificamente para gerenciar o ciclo de vida dos *tokens JWT* (*JSON Web Tokens*), demonstrou uma precisão notável no controle de acesso dos usuários. Este *app* garantiu que apenas usuários devidamente autenticados pudessem interagir com as funcionalidades protegidas do sistema, mantendo assim a integridade e a segurança da aplicação. O *JWT*, sendo um padrão robusto para autenticação, permitiu que o sistema validasse as requisições de forma ágil e segura, reforçando a proteção das informações e garantindo que apenas usuários autorizados tivessem acesso aos recursos críticos. Esse processo de autenticação se revelou particularmente eficaz em ambientes onde a segurança dos dados é uma prioridade, como no caso de aplicações educacionais que lidam com informações sensíveis de estudantes e professores.

Para ilustrar o funcionamento detalhado desse sistema, um exemplo de configuração de rotas no arquivo *urls.py* pode ser observado na Figura 5. Neste exemplo, as rotas são definidas para diferentes *views* — um exemplo delas observada na Figura 6 — que gerenciam tanto a interface de usuário quanto a lógica de *back-end*, assegurando uma eficiente manipulação de dados e resposta às requisições feitas pelo *front-end*.

```python
from django.urls import path
from rest_framework_simplejwt.views import (
    TokenRefreshView,
    TokenVerifyView,
)
from authentication import views

urlpatterns = [
    path("me/", views.UserInformation.as_view(), name="my_profile"),
    path("first_time/", views.FirstTimeLogin.as_view(), name="first_time_login"),
    path("setup/", views.UserSetup.as_view(), name="account_setup"),
]
```

Figura 5. Exemplo de configuração de rotas *urls.py*

```python
class UserInformation(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request):
        serializer = serializers.UserSerializer(request.user)
        return Response(serializer.data)
```

Figura 6. Exemplo de *view* acionada pela *url "me/"*

Além disso, a criação de um *token JWT* no Manthano é realizada utilizando a biblioteca *django-simple-jwt*, que simplifica o processo de geração e validação dos *tok*ens dentro da aplicação *Django*. Quando um usuário fornece suas credenciais de *login*, uma requisição *POST* é enviada para a *endpoint* de autenticação *"token/*", onde a *view* correspondente utiliza a classe *TokenObtainPairView*, do *django-simple-jwt*, para gerar um par de *tokens* — um de acesso e um de atualização. O *token* de acesso contém informações do usuário, como seu ID e permissões, e tem uma validade curta, enquanto o *token* de atualização pode ser usado para obter novos tokens de acesso sem que o usuário precise fazer *login* novamente. Este mecanismo não apenas facilita o gerenciamento de sessões, como também aumenta a segurança ao limitar o tempo de exposição de um *token* comprometido.

A criação e verificação dos *tokens* são configurados no *Django* por meio de ajustes no *settings.py*, onde chaves secretas e parâmetros de expiração são definidos. Por exemplo, após um usuário ser autenticado, um *token JWT* de acesso pode ter a seguinte aparência:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxLCJ1c2VybmFtZSI6ImV4YW1wbGVfdXNlciIsImV4cCI6MTY4NTc2MzAwMCwiaWF0IjoxNjg1NzU5NDAwfQ.4Ff7dy0LdHTnBv5Oo0lc3ZjqWsl2xPQnxG2jP5y0-SY.

Este *token* é divido em três partes: o cabeçalho (*header*), o *payload* (corpo), que inclui as informações codificadas do usuário e a assinatura (*signature*), que garante a autenticidade e integridade do *token*. Um exemplo de configuração típica pode ser observado na Figura 7.

```python
# Configurações do Django REST Framework
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}

# Configurações do django-simple-jwt
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),  # Tempo de expiração do token de acesso
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),    # Tempo de expiração do token de atualização
    'ROTATE_REFRESH_TOKENS': True,  # Gera um novo token de atualização toda vez que um token de acesso é atualizado
    'BLACKLIST_AFTER_ROTATION': True,  # Adiciona o token antigo a uma lista negra após a rotação
    'ALGORITHM': 'HS256',  # Algoritmo de criptografia usado para assinar o token
    'SIGNING_KEY': SECRET_KEY,  # Chave secreta usada para assinar o token
    'VERIFYING_KEY': None,  # Usado para assinaturas assimétricas
    'AUTH_HEADER_TYPES': ('Bearer',),  # Prefixo do cabeçalho de autenticação
    'USER_ID_FIELD': 'id',  # Campo de ID do usuário
    'USER_ID_CLAIM': 'user_id',  # Claim que armazena o ID do usuário
    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),  # Classes de token usadas
    'TOKEN_TYPE_CLAIM': 'token_type',  # Claim para o tipo de token
    'JTI_CLAIM': 'jti',  # Claim para identificação do token
    'SLIDING_TOKEN_REFRESH_EXP_CLAIM': 'refresh_exp',  # Claim para expiração de tokens deslizantes
    'SLIDING_TOKEN_LIFETIME': timedelta(minutes=5),  # Tempo de expiração do token deslizante
    'SLIDING_TOKEN_REFRESH_LIFETIME': timedelta(days=1),  # Tempo de expiração do token deslizante para atualização
}
```

Figura 7. Exemplo de configuração de *token* no *settings.py*.

Adicionalmente, nesta seção será explicado o funcionamento do *ChannelConsumer* do *Django Channels*, detalhando cada método e sua função no consumo de mensagens. Iniciando-se pela Figura 8, o método *connect()* é responsável por estabelecer a conexão *WebSocket*, verificando a validade do usuário e o canal de comunicação. Ele busca as instâncias de *Classroom* e *Channel* no banco de dados, utilizando o código e o nome do canal fornecidos na *URL* da requisição. Caso o usuário não pertença à sala de aula correta, a conexão é encerrada, garantindo que apenas usuários autorizados acessem o canal.

```python
async def connect(self):      You, há 11 meses • basic django-channels setup
    classroom_code = self.scope["url_route"]["kwargs"]["classroom_code"]
    channel_name = self.scope["url_route"]["kwargs"]["channel_name"]

    try:
        self.classroom = await database_sync_to_async(Classroom.objects.get)(code=classroom_code,)
    except:
        return await self.close()

    try:
        self.channel = await database_sync_to_async(Channel.objects.get)(name=channel_name, classroom=self.classroom)
    except:
        return await self.close()

    self.group_name = str(self.classroom.id) + "." + str(self.channel.id)

    if await self.check_user_classroom(self.scope['user'], self.classroom):
        return await self.close()

    await self.channel_layer.group_add(
        self.group_name,
        self.channel_name
    )
```

Figura 8. Método *connect()* presente no *ChannelConsumer*

Na Figura 9, o método *disconnect()* é mostrado, onde a lógica para remover o usuário do grupo de comunicação ao se desconectar é implementada, evitando que ele continue recebendo mensagens. Já a Figura 10 apresenta o método *receive(),* que processa as mensagens recebidas dos clientes, salva-as no banco de dados como instâncias de *Message* e, em seguida, distribui-as para todos os membros do grupo em tempo real. Finalmente, a Figura 11 ilustra o método *chat_message(),* que é responsável por formatar e enviar as mensagens para os clientes conectados, assegurando que todos os participantes da sala de aula virtual recebam as informações de maneira consistente e ordenada.

```python
async def disconnect(self, close_code):
    await self.channel_layer.group_discard(
        self.group_name,
        self.channel_name
    )
```

Figura 9. Método dis*connect()* presente no *ChannelConsumer*

```python
async def receive(self, text_data):
    text_data_json = json.loads(text_data)

    user = self.scope["user"]
    msg = text_data_json["text"]

    message = await database_sync_to_async(Message.objects.create)(user=user, text=msg, channel=self.channel)

    data = {
        "type": 'chat_message',
        "id": message.id,
        "user_id": message.user.id,
        "username": message.user.username,
        "date": message.date.strftime('%Y-%m-%d %H:%M:%S.%f'),
        "text": message.text,
    }

    await self.channel_layer.group_send(
        self.group_name,
        data
    )
```

Figura 10. Método receive*()* presente no *ChannelConsumer*

```python
async def chat_message(self, event):
    response = {
        'type': 'websocket.send',
        'user_id': event['user_id'],
        'username': event['username'],
        'text': event['text']
    }
    await self.send(json.dumps(event))
```

Figura 11. Método *chat_message()* presente no *ChannelConsumer*

O *app manthano_backend*, no que lhe concerne, centralizou as diferentes configurações do sistema, como segurança e banco de dados, desempenhando seu crucial papel na coordenação das funcionalidades globais do Manthano. As

configurações de segurança, como o gerenciamento da *SECRET_KEY*, foram tratadas com extremo cuidado, utilizando-se a biblioteca *django-decouple* para garantir que informações sensíveis fossem protegidas, tanto em ambientes de desenvolvimento quanto de produção. Além disso, o *manthano_backend* gerenciou a configuração do banco de dados, onde a escolha do *SQLite* como banco padrão, integrado diretamente ao *Django*, facilitou a prototipagem e a execução de testes em ambientes de desenvolvimento. Entretanto, reconhece-se que, em cenários de produção ou em ambientes de larga escala, a migração para um banco de dados mais robusto, como o *PostgreSQL* ou o *MySQL*, será necessária para garantir a escalabilidade e a capacidade de lidar com grandes volumes de dados, mantendo a performance e estabilidade da aplicação a longo prazo.

Por fim, a implementação de um *middleware* específico para a autenticação de requisições *WebSocket*, utilizando *JWT*, demonstrou-se crucial para manter a segurança e a autenticidade das comunicações em tempo real dentro do sistema Manthano. O *ChannelConsumer*, responsável pelo gerenciamento das interações nas *Channels*, assegurou que todos os eventos fossem devidamente processados e transmitidos aos usuários conectados, o que garantiu coesão e responsividade para professores e estudantes.

3.6 Banco de dados

A escolha de utilizar o banco de dados padrão do *Django, SQLite*, em conjunto com o sistema *ORM*, foi uma decisão estratégica que resultou numa significativa economia de tempo durante o desenvolvimento do Manthano. Essa escolha permitiu concentração mais focada na lógica de negócio e menos na configuração e gerenciamento do banco de dados. O *SQLite*, sendo um banco de dados leve e sem a necessidade de complexas configurações, viabilizou uma prototipagem rápida em testes locais. Isso, aliado à abstração oferecida pelo *ORM*, eliminou a necessidade de manipulação de escrever complexos comandos *SQL* manualmente, permitindo que operações de manipulação de dados fossem realizadas através de simples chamadas de métodos em *Python*. Como resultado, o tempo necessário para implementar e testar funcionalidades críticas foi substancialmente reduzido, acelerando o ciclo de desenvolvimento e permitindo concentração em refinamento de outros aspectos do sistema.

Além disso, o uso da *ORM* do *Django* simplificou significativamente a integração entre diferentes módulos do sistema, como os de *classroom* e *authentication*. A capacidade de mapear automaticamente os modelos para tabelas do banco de dados garantiu que as alterações feitas na estrutura dos dados fossem refletidas de maneira consistente e eficiente, sem a necessidade de intervenções manuais. Isso não só economizou tempo, como também minimizou a chance de erros, já que o *ORM* gerencia as complexidades das relações e integridade referencial de maneira automática. Essa eficiência se traduz em um processo de desenvolvimento mais ágil, onde novas funcionalidades podem ser adicionadas e testadas rapidamente.

Para ilustrar a estrutura e o relacionamento entre os dados no banco de dados utilizado pelo Manthano, foram gerados dicionários de dados que detalham as tabelas

principais representadas pelos modelos do *Django* e suas respectivas colunas, incluindo os tipos de dados e chaves estrangeiras.

As Tabelas 1-4 a seguir descrevem as tabelas presentes no módulo *classroom*. Já as Tabelas 5-8 Tabelas descrevem as tabelas presentes do módulo *authentication*.

Tabela 1. classroom_classroom

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
name	varchar(255)	NOT NULL	Texto	'Turma A'
code	varchar(11)	NOT NULL, UNIQUE	Código	'ABC123'
description	text	NOT NULL	Texto livre	'Descrição da turma A.'
schedule	varchar(100)	NULL	Texto	'Segunda e Quarta, 14:00 - 16:00'
created_at	datetime	NOT NULL	ISO 8601	'2024-01-01T09:00:00'
updated_at	datetime	NOT NULL	ISO 8601	'2024-01-10T09:00:00'
professor_managed	bool	NOT NULL	true/false	true

Tabela 2. classroom_channel

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
name	varchar(255)	NOT NULL	Texto	'Canal de Discussão'
classroom_id	bigint	NOT NULL, REFERENCES classroom_classroom(id) DEFERRABLE INITIALLY DEFERRED	ID da sala de aula	1

Tabela 3. classroom_jitsichannel

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
name	varchar(255)	NOT NULL	Texto	'Reunião Semanal'
room_name	varchar(255)	NOT NULL	Texto	'sala01'
classroom_id	bigint	NOT NULL, REFERENCES classroom_classroom(id) DEFERRABLE INITIALLY DEFERRED	ID da sala de aula	1

Tabela 4. classroom_message

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
text	text	NOT NULL	Texto livre	'Olá a todos!'
edited	bool	NOT NULL	true/false	false
channel_id	bigint	NOT NULL, REFERENCES classroom_channel(id) DEFERRABLE INITIALLY DEFERRED	ID do canal	1
user_id	bigint	NULL, REFERENCES authentication_manthanouser(id) DEFERRABLE INITIALLY DEFERRED	ID de usuário	1
date	datetime	NOT NULL	ISO 8601	'2024-08-31T10:30:00'

Tabela 5. authentication_manthanouser

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
password	varchar(128)	NOT NULL	Hash de senha	'pbkdf2_sha256$216000$abc...'
last_login	datetime	NULL	ISO 8601	'2024-08-31T10:30:00'
is_superuser	bool	NOT NULL	true/false	false
is_staff	bool	NOT NULL	true/false	true
username	varchar(32)	NOT NULL	Texto	'johndoe'
first_name	varchar(32)	NOT NULL	Texto	'John'
last_name	varchar(255)	NOT NULL	Texto	'Doe'
email	varchar(255)	NOT NULL, UNIQUE	Email	'john.doe@example.com'
is_admin	bool	NOT NULL	true/false	false
is_active	bool	NOT NULL	true/false	true
date_joined	datetime	NOT NULL	ISO 8601	'2024-01-01T12:00:00'
classroom_id	bigint	NULL, REFERENCES `classroom_classroom`(id) DEFERRABLE INITIALLY DEFERRED	ID da sala de aula	1
is_teacher	bool	NOT NULL	true/false	true
academic_email	varchar(255)	NULL, UNIQUE	Email	'j.doe@academic.edu'

Tabela 6. authentication_profile

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
profile_picture	varchar(100)	NULL	'path/to/image'	'uploads/profile_pics/pic1.jpg'
profile_background	varchar(100)	NULL	'path/to/image'	'uploads/backgrounds/bg1.jpg'
desc	text	NULL	Texto livre	'Professor de Matemática.'
user_id	bigint	NOT NULL, UNIQUE, REFERENCES `authentication_manthanouser`(id) DEFERRABLE INITIALLY DEFERRED	ID de usuário	1

Tabela 7. authentication_student

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
enrollment	varchar(255)	NOT NULL, UNIQUE	Texto	'20240001'
user_id	bigint	NOT NULL, UNIQUE, REFERENCES `authentication_manthanouser`(id) DEFERRABLE INITIALLY DEFERRED	ID de usuário	1

Tabela 8. authentication_professor

Coluna	Tipo de dado	Restrições	Formato Padrão	Exemplo de Instância
id	integer	PRIMARY KEY, AUTOINCREMENT	N/A	1
academic_rank	varchar(255)	NULL	Texto	'Doutor'
user_id	bigint	NOT NULL, UNIQUE, REFERENCES `authentication_manthanouser`(id) DEFERRABLE INITIALLY DEFERRED	ID de usuário	2

Como exemplo de dado utilizado em caso real de uso presente no Manthano, na Tabela 9, são exibidas as informações de um usuário cadastrado como aluno na plataforma, que faz uso da tabela *authentication_manthanouser*.

Tabela 9. Exemplo de dados reais utilizados no Manthano

Coluna	Dados presentes
id	1
password	'pbkd22_sha256$146000$ada...'
last_login	'2024-05-31T10:30:00'
is_superuser	false
is_staff	false
username	‘reszko'
first_name	'Fabio’
last_name	‘dos Santos Reszko Junior'
email	s.reszkojr@gmail.com'
is_admin	false
is_active	true
date_joined	'2023-12-21T12:00:00'
classroom_id	2
is_teacher	true
academic_email	'fabio@ifpr.edu.br'

Outro benefício notável da utilização do *SQLite* e do *ORM* do *Django* foi no desenvolvimento colaborativo. Por ser um banco de dados baseado em arquivos, o *SQLite* simplifica a configuração de ambiente de desenvolvimento para múltiplos desenvolvedores, permitindo que cada membro da equipe pudesse trabalhar localmente com a mesma versão do banco de dados sem a necessidade de configurações de *setups* complexos ou dependências adicionais, o que não só promove uma maior consistência entre diferentes desenvolvedores, como também reduz potenciais conflitos de configuração e versionamento que poderiam surgir em um ambiente multiusuário.

Adicionalmente, a facilidade de migração e implantação proporcionada pela integração nativa do *Django ORM* com o *SQLite* são as *migrations*. A ferramenta *migrations* do *Django* permite que alterações feitas no esquema do banco de dados — ou seja, sua estrutura ou arquitetura —, como adição de novos campos ou modificações de relações entre tabelas, fossem aplicadas de forma automática. Esse processo de migração, sendo automatizado e testado em ambiente local, garante que o código do ambiente de produção seja sempre sincronizado com as mudanças no banco de dados, reduzindo drasticamente o risco de inconsistências e falhas durante as atualizações.

A ferramenta *migrations,* do *Django,* também traz a capacidade de reverter para uma versão anterior do banco de dados, algo que oferece uma significativa vantagem no processo de depuração e testes. No momento em que erros ou comportamentos inesperados surgem após uma nova migração, torna-se possível retornar de maneira rápida e eficiente a um estado anterior do banco de dados para diagnosticar e corrigir problemas, sem a necessidade de operações manuais ou *backups* complexos e de alto custo de armazenamento. Isso não só acelera o processo de resolução de *bugs*, como também garante que o ambiente de desenvolvimento permaneça estável e confiável, permitindo a uma equipe de desenvolvimento foco em pontos mais específicos e importantes do sistema.

4. Conclusão

Ao longo do desenvolvimento do Manthano, diversos objetivos específicos foram alcançados, evidenciando o sucesso das funcionalidades implementadas na plataforma. A integração das ferramentas de comunicação e gestão resultou em uma solução coesa que atendeu às necessidades identificadas.

Primeiramente, a centralização das funcionalidades em uma única plataforma, como videochamadas e *chat* por texto, cumpriu o objetivo de reduzir a fragmentação dos sistemas utilizados anteriormente. Essa integração permitiu uma gestão acadêmica mais eficiente e a eliminação da necessidade de múltiplos aplicativos, facilitando a comunicação entre alunos e professores. Tais funcionalidades, como a criação de canais específicos para cada tipo de interação, atenderam, também, à necessidade de uma abordagem mais estruturada e acessível, com *Channels* proporcionando um ambiente organizado para discussões e atividades de grupo, e *JitsiChannels* garantindo a realização de videoconferências de forma integrada e simplificada.

Além das funcionalidades implementadas no sistema, ainda há um potencial significativo para aprimorar o Manthano por meio da adição de novos recursos que enriqueçam ainda mais a plataforma. Entre as melhorias planejadas, destaca-se a implementação de um sistema de compartilhamento de arquivos, que facilitaria a troca de documentos e materiais diretamente na plataforma, aprimorando a comunicação entre professores e alunos. Outra evolução proposta é a criação de atividades personalizadas por meio de formulários, permitindo que professores gerem e gerenciem tarefas e avaliações de maneira mais dinâmica e eficiente. A inclusão de um calendário compartilhado também está em consideração, permitindo a visualização clara de datas importantes e atividades, auxiliando no planejamento e acompanhamento dos prazos.

Adicionalmente, melhorias como um sistema de notificações, para alertar os usuários sobre atualizações e eventos importantes, e um módulo de *feedback*, para coleta e análise de opiniões dos usuários, poderiam fortalecer a interação e o engajamento dentro da plataforma.

Por fim, a integração com plataformas de aprendizagem externa, como ferramentas para migração de dados, ampliaria os recursos educacionais disponíveis, proporcionando uma experiência de aprendizado ainda mais abrangente. Esses aprimoramentos não apenas expandiriam as funcionalidades atuais, como também atenderiam às necessidades emergentes do ambiente educacional digital, garantindo que o Manthano continue evoluindo e atendendo a novas demandas.

Portanto as funcionalidades implementadas no Manthano não apenas atenderam aos objetivos propostos como também demonstraram um avanço significativo na integração das ferramentas de ensino, promovendo uma experiência mais fluida e coesa. O Manthano, portanto, cumpriu as metas ao fornecer uma plataforma abrangente e eficiente para a educação a distância, estabelecendo um novo padrão para a gestão e comunicação acadêmica.

Referências

8x8. Jitsi as a Service. Versão 5218. [S. l.], 2 ago. 2024. Disponível em: https://jaas.8x8.vc/. Acesso em: 2 ago. 2024

FARIA, Adriano Antonio; SALVADORI, Angela. A educação a distância e seu movimento histórico no Brasil. **Revista das Faculdades Santa Cruz**, v. 8, n. 1, 2011.

BRASIL. Resolução CNE/CP nº 2, de 10 de dezembro de 2020. Brasília, DF: Ministério da Educação, 2020.

CASTELLS, Manuel. **A sociedade em rede**. São Paulo: Paz e terra, 2005.

CECÍLIO, Sálua; REIS, Briana Manzan. Trabalho docente na era digital e saúde de professores universitários. **Educação: Teoria e Prática**, v. 26, n. 52, p. 295-311, 2016.

DE SOUSA OLIVEIRA, Eleilde et al. A educação a distância (EaD) e os novos caminhos da educação após a pandemia ocasionada pela Covid-19. **Brazilian Journal of Development**, v. 6, n. 7, p. 52860-52867, 2020.

JONES, Michael;, BRADLEY, John. JSON Web Token (JWT). RFC 7519, 2015.

KANTOR Ilya. Long polling. **Javascript.Info**, 12 de dez. de 2022. Disponível em https://javascript.info/long-polling. Acesso em: 7 de jun. de 2024.

MORAN, José Manuel. Aperfeiçoando os modelos de EAD existentes na formação de professores. **Educação**, v. 32, n. 03, p. 286-290, 2009.

MORAES, Maria Cândida. Educar na biologia do amor e da solidariedade. 2003.

MORIN, Edgar et al. O**s setes saberes necessários à educação do futuro**. Cortez Editora, 2014.

DA SILVA COQUEIRO, Naiara Porto; SOUSA, Erivan Coqueiro. A educação a distância (EAD) e o ensino remoto emergencial (ERE) em tempos de Pandemia da covid 19 Distance education (Ed) and emergency remote education (ERE) in times of Pandemic covid 19. **Brazilian Journal of Development**, v. 7, n. 7, p. 66061-66075, 2021.

PUCCI, Bruno; CERASOLI, Josianne Francia. As novas tecnologias e a intensificação do trabalho docente na universidade. **Educação e Filosofia**, v. 24, n. 47, p. 171-190, 2010.

PENTEADO, Regina Zanella; COSTA, Belarmino Cesar Guimarães Da. Trabalho docente com videoaulas em EAD: dificuldades de professores e desafios para a formação e a profissão docente. **Educação em Revista**, v. 37, p. e236284, 2021.

SCHLEMMER, Eliane; SACCOL, Amarolinda Zanela; GARRIDO, Susane. Um modelo sistêmico de avaliação de softwares para educação a distância como apoio à gestão de EaD. **REGE Revista de Gestão**, v. 14, n. 1, p. 77-91, 2007.