**Projeto Peppa Educações**

Integrantes: Taís Bruno, Rafael Carrilho, Natália Bruno, Antônio Vinícius e Pedro Xavier

1. **Introdução**
   1. **Objetivo**

O Peppa Educações é uma plataforma de cursos com o objetivo de facilitar o acesso ao conhecimento através da internet. Através dela, pessoas podem criar e compartilhar cursos de diversas áreas do conhecimento, que poderão ser acessados por outros usuários que tiverem interesse em adquirir novas habilidades.

* 1. **Escopo**

O Peppa Educações é uma plataforma em que os usuários podem publicar e acessar cursos de diversas áreas de conhecimento. Um usuário pode se cadastrar no sistema com seu email, nome e senha protegida. O sistema apresenta módulos para cadastro, alteração, consulta e remoção de cursos e aulas para os cursos.

O cadastro de um curso exige somente o nome e as categorias relacionadas ao curso. Após a criação do curso, o usuário deve estipular aulas a serem publicadas junto ao curso. Um curso está relacionado a um instrutor e este tem permissão para editar o curso e suas aulas, além de poder anexar material à aula. O instrutor pode publicar múltiplos cursos e pode visualizar a lista dos cursos publicados.

No sistema, um usuário pode buscar cursos por nome, filtrá-los por avaliações e categorias. Ao selecionar um curso na lista, o mesmo visualiza um resumo do curso com a lista de aulas e pode se inscrever no curso. Uma vez inscrito, o usuário pode assistir às aulas do curso, comentar no curso, comentar nas aulas do curso e avaliar o curso.

A plataforma salva o progresso das aulas para que, caso interrompidas, os usuários possam retomar de onde pararam. O usuário pode se inscrever em diversos cursos e visualizar a lista de cursos nas quais está inscrito. O sistema mantém um histórico de atividades realizadas pelos usuários, tal qual criar curso, criar aula, permitindo um acompanhamento eficiente de suas contribuições à plataforma.

1. **Requisitos Arquiteturais**
   1. **Restrições**

As seguintes restrições de requisito e de sistema possuem uma relação significativa com a arquitetura:

* Acesso via internet: A aplicação poderá ser acessada por meio da internet através de requisições web.
* Linguagem e framework: A linguagem utilizada no desenvolvimento do sistema será o Ruby, adotando o framework Ruby on Rails.
* Somente usuários cadastrados e autenticados no sistema têm acesso a suas funcionalidades.
* Sendo um sistema Web, ele deverá funcionar com os navegadores Internet Explorer 8 ou superior, Firefox 20 ou superior e Chrome 24 ou superior.
  1. **Requisitos e Soluções**

| **Requisito** | **Solução Arquitetural** |
| --- | --- |
| Autenticação e Autorização | Utilização de autenticação baseada em Basic Token com JWT, e autenticação baseada em papéis, garantindo maior segurança e controle de acesso aos recursos da plataforma. |
| Paginação de Dados | Ao carregar listas de conteúdo com mais de 20 itens, a resposta será paginada através do parâmetro ‘page’, proporcionando melhor desempenho e experiência do usuário. |
| Armazenamento de Arquivos | O armazenamento de arquivos na aplicação poderá ser realizado em sistema de arquivos ou banco de dados. A estratégia de armazenamento deverá ser determinada de acordo com o volume de dados, as características dos arquivos armazenados e o ciclo de vida dos arquivos no sistema. |
| Segurança do Sistema | As senhas dos usuários serão criptografadas ao completar o cadastro. Para acessar os registros da aplicação, os usuários deverão informar um token de autorização que só será gerado após um login bem-sucedido com o email e senha correspondentes. |
| Exclusão lógica de dados | Ao excluir um recurso da aplicação, o mesmo não será apagado do banco de dados. Ao invés disso, será atribuída uma flag de ‘deleted\_at’, indicando que o recurso foi apagado e está inativo. |
| Utilização do padrão MVC | A arquitetura do sistema será dividida em camadas, cada uma com funções específicas, seguindo o padrão Model-View-Controller(MVC) para maior organização e manutenibilidade do código. |
| Portabilidade de SGBDs | Devido ao uso do framework Ruby on Rails, é possível conectar qualquer SGBD à plataforma através de sua interface de queries dinâmicas, garantindo flexibilidade e adaptabilidade. |
| Interfaces de usuário  amigáveis e dinâmicas | A interface de usuário será desenvolvida considerando a usabilidade da aplicação. |

1. **Padrões arquiteturais adotados**

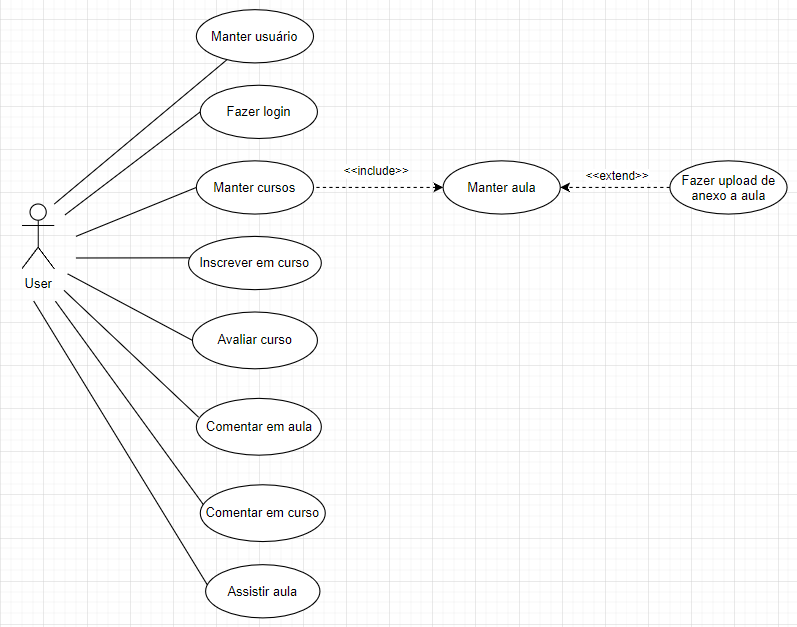
A arquitetura do sistema será fundamentada nos padrões MVC (Model-View-Controller) e Cliente-Servidor. A adoção do Cliente-Servidor envolverá a implementação de uma API REST, que irá dispor objetos JSON e status HTTP como resposta às requisições web realizadas pelos usuários.

Optamos pelo padrão MVC devido a sua ampla utilização em sistemas de informação e no mercado, bem como pela familiaridade e experiência prévia dos integrantes do grupo com este tipo de arquitetura. Além disso, ao dividir as responsabilidades entre Models, Controllers e Views, a arquitetura proposta promove escalabilidade, manutenibilidade, testabilidade e modularidade do sistema, facilitando seu desenvolvimento e evolução ao longo do tempo.

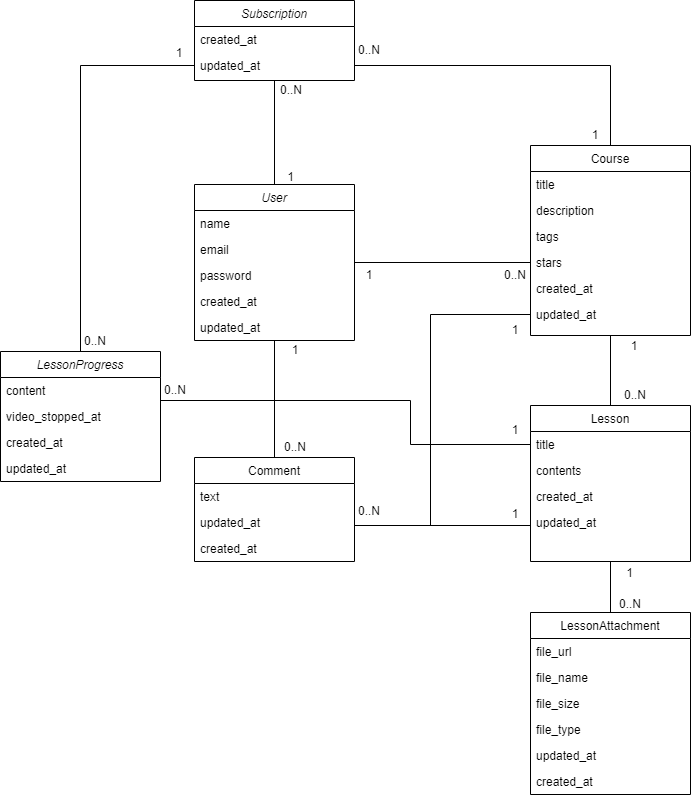
1. **Visão arquitetural**

Os diagramas de análise do projeto proporcionam uma visão simplificada das classes a serem implementadas, abrangendo o diagrama de casos de uso, modelo conceitual e o diagrama de sequência do sistema. Esses diagramas serão anexados como imagens ao repositório do projeto, para fácil acesso de todos os desenvolvedores envolvidos.

**4.1 Diagrama de casos de uso**

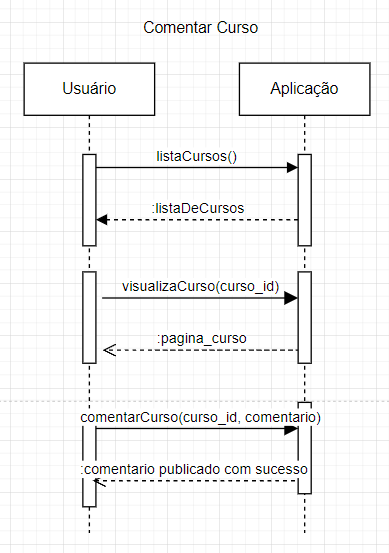


**4.2 Modelo conceitual**

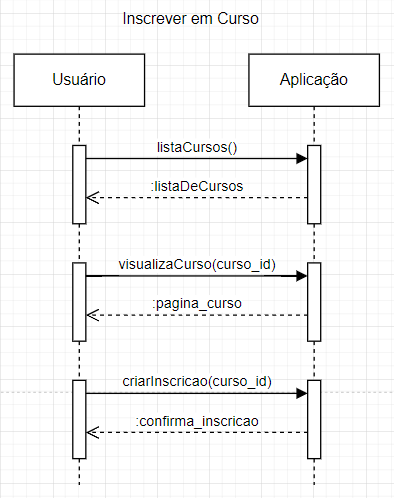
****

**4.3 Diagrama de sequência do sistema**

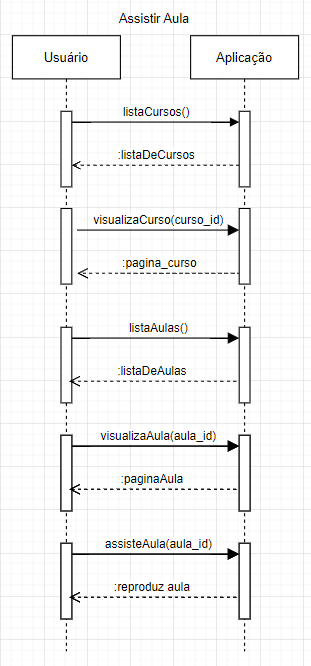
Comentar em curso:

****

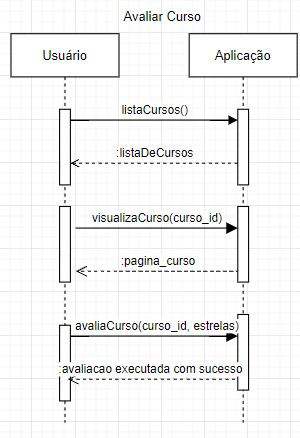
Inscrever em curso:

****

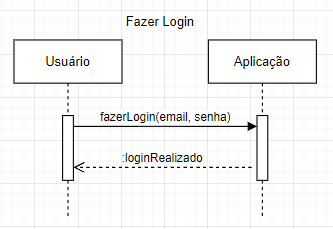
Assistir aula:

****

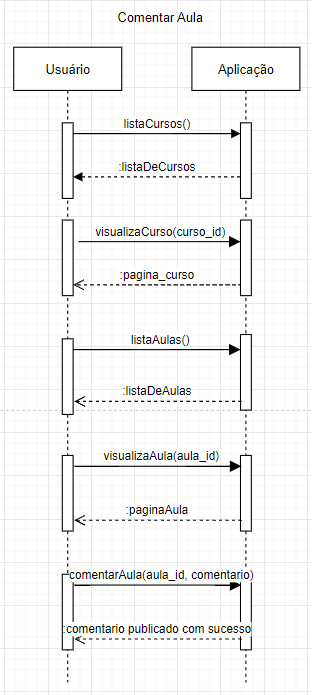
Avaliar curso:

****

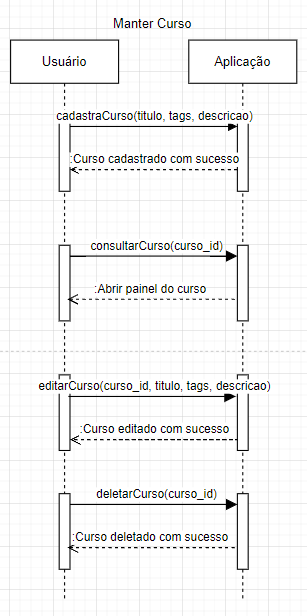
Fazer login:

****

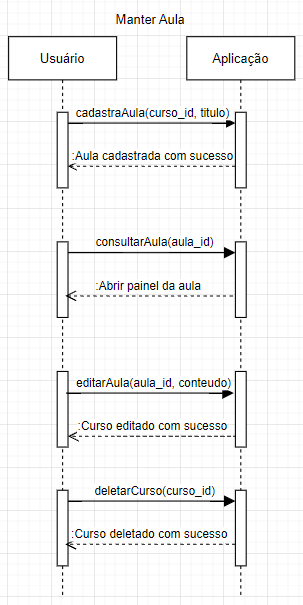
Comentar em aula:

****

Manter curso:

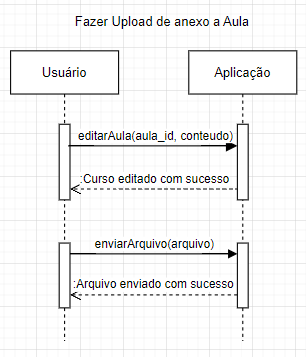
****

Manter aula:

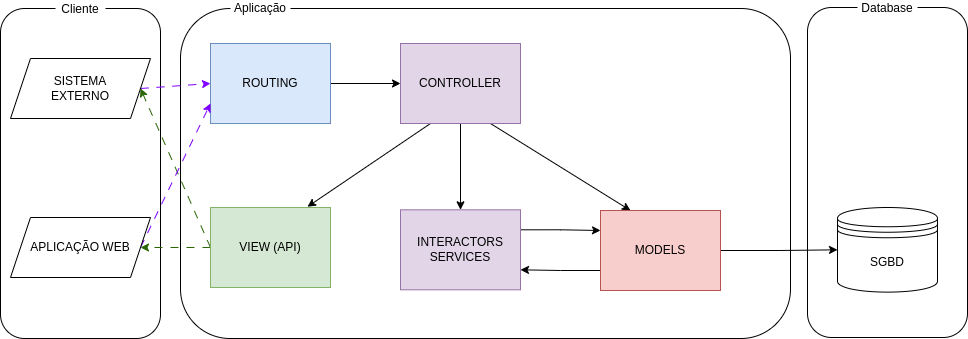
****

Manter usuário: Análogo a Manter Curso

Fazer upload de anexo a aula:

****

**4.4 Diagrama de arquitetura**



A aplicação Web representa a interface do front end desenvolvido em … que será conectado a API através de requisições HTTP. A linha tracejada verde representa a resposta da API, enquanto a linha tracejada roxa representa a requisição HTTP em si. A representação do SGBD explicita o banco de dados da nossa aplicação, que pode ser o MySQL, Postgres, etc. Models representa a classe que lidará com ações referentes ao banco de dados e que fará a interface com o SGBD. A classe Routing irá mapear os endpoints da aplicação para uma ação em um controller. O Controller é uma classe que recebe um chamado através do Routing, e funcionará como um intermediador entre o usuário e as lógicas de serviço. Os interactor services são as classes que vão extrair as lógicas de serviço dos controllers, tornando-os mais modularizados. Ao finalizar a execução das lógicas de serviço, os Controllers irão retornar uma resposta em formato JSON (View da API) para o serviço externo que fez a requisição.

Foi adotado o estilo arquitetural em camadas e Cliente-Servidor com o propósito de dividir as responsabilidades, facilitar a manutenção e evolução do sistema e aumentar a flexibilidade.

1. **Qualidade**

Com a elaboração desse documento de arquitetura, podemos garantir que o sistema Peppa Educações atenderá aos atributos de qualidades desejados. Ao adotar o padrão MVC, garantimos o atributo de Manutenibilidade por deixarmos as classes com responsabilidades divididas e reduzidas, facilitando a compreensão e a modificação do código.

A arquitetura de conexão com o banco de dados, garantida pelo uso do Ruby on Rails, nos dá a flexibilidade de conectar com qualquer SGBD que seja de interesse do desenvolvedor. Isso nos proporciona Portabilidade, permitindo a migração para outros sistemas de banco de dados, se necessário, sem grandes impactos para o sistema.

A implementação de criptografia de senha e role based authentication são medidas essenciais para garantir os atributos de Segurança e Adequação Funcional do sistema. Essas estratégias evitam o acesso não autorizado a informações sensíveis, garantindo que apenas usuários com as devidas permissões possam acessar determinadas partes do sistema.

**6. Diagrama de classe detalhado: (Imagem do diagrama está no Github)**

Escolhemos a arquitetura MVC para o nosso projeto. Fizemos um diagrama de classe com todas as classes do sistema incluindo as classes de Services, Models, Controllers, mas para compactar o planejamento do diagrama de classe, tiramos das classes que implementam interfaces os métodos que já estão declarados na interface. Para os modelos, no diagrama, temos em verde a classe base concedida pela ferramenta chamada 'ActiveRecord', que é responsável pela implementação e entrega de métodos para a manipulação do banco de dados. Essa classe especial concede para todos os modelos que herdam dela alguns métodos utilizados para consultar registros, criar novos registros na base de dados, editá-los, apagá-los entre outros procedimentos bastante utilizados. Os controllers terão a responsabilidade de receber as demandas das requisições HTTP e desempenharão funções como: autenticação da requisição; Entendimento da solicitação do usuário; Delegação para a classe de serviço responsável, quando necessário; Formatação da resposta; Entrega da resposta formatada. Para realizar a formatação da View que será entregue como resposta, o controller irá instanciar uma classe de serializer, que será implementada somente com os atributos que desejamos entregar para o solicitante.

**7. Padrões GoF (Imagem dos diagramas está no Github):**

**7.1 - Strategy:**

Para garantir a utilização do padrão GoF **Strategy**, utilizaremos um módulo de entrega de conteúdos baseado no tipo desse conteúdo. Dessa forma, quando uma lição precisar entregar ao solicitante sua lista de conteúdos, ela fará isso com base na implementação do método 'Format' de cada tipo de conteúdo permitido, entre eles: documento (imagens, pdf's), texto e vídeo.

**7.2 e 7.3 - Singleton e Factory:**

Para garantir a utilização dos padrões GoF **Singleton** e **Factory** em nosso projeto, iremos aplicá-los às classes de serviço, que são responsáveis pela implementação das lógicas de negócio e pelos permissionamentos necessários para manipular os dados da plataforma. A implementação do Singleton permitirá que uma única instância do serviço seja inicializada para tratar todas as requisições, evitando a necessidade de criar novas instâncias. Para a implementação do Factory, as classes de serviço trabalharão diretamente com os dados a serem criados ou manipulados, receberão as informações necessárias a partir dos controllers e irão aplicar as lógicas de criação de cada tipo de classe. Isso inclui uma variedade de operações, como a criação de recursos. Por exemplo, no caso específico da criação de um comentário, utilizaremos uma fábrica capaz de criar comentários tanto para um curso quanto para uma aula, baseando-se na informação 'commentable' que receberá.

Em caso de ações de consulta, o controller faz o tratamento, formata a resposta e a devolve. Quando a ação demanda um serviço, o controller chama a classe de serviço Singleton apropriada, que executa o serviço e retorna a resposta para o usuário.

**7.4 - State**

Para garantir a utilização do padrão GoF **State**, teremos as classes ExpiresState, CompletedState e InProgressState que implementam uma interface SubscriptionState, que está associada a uma Subscription. Dessa forma, um objeto de Subscription poderá ter seu comportamento alterado de acordo com o seu estado interno. Por exemplo, se a Subscription estiver em 'InProgressState', a chamada ao método 'start' teria um comportamento diferente em relação a outra Subscription que estivesse num estado de 'ExpiresState'. Isso também irá refletir em permissionamento de acessos do usuário, porcentagem de completude do curso entre outras.

**7.5 - Composite**

Para garantir a utilização do padrão GoF Composite, aplicamos este padrão para gerir a relação e operações entre cursos, aulas e conteúdos. Um 'Curso' é composto por uma estrutura com diversas "Aulas", e cada "Aula" por sua vez contém diversos 'Conteudos'. Aqui, a 'Subscription' funciona como um componente de acompanhamento do progresso do aluno, marcando a completude total no curso.

O atributo "completion\_percentage" do objeto 'Subscription' criado quando o aluno se inscreve em um curso será atualizado toda vez que uma nova instância de LessonProgress for adicionada para marcar que um aluno já viu determinada aula. No serviço de 'LessonMarkProgress', ele irá chamar o método de calculate\_percentage do objeto de Subscription, que olha todas as aulas completas e faltantes, para calcular essa porcentagem.

**8. Diagramas de sequência detalhados:**

**Imagem do diagrama está no Github**

Casos de uso:

* Comentar em curso
* Inscrever em curso
* Manter usuário
* Manter curso
* Fazer upload de anexo a aula
* Manter aula
* Comentar em aula
* Fazer login
* Avaliar curso
* Assistir aula
* Completar inscrição do curso
* Expirar inscrição do curso

**9. Padrões GRASP:**

**9.1 e 9.2 - Coesão alta (High Cohesion) e Acoplamento fraco (Low Coupling):**

A implementação da arquitetura MVC garante parcialmente o aumento da coesão de código e diminuição do acoplamento, pois traz um maior entendimento e organização sobre o contexto dos métodos em cada classe e evita que uma classe que será modificada force uma modificação em outra classe. Nessa arquitetura, vemos os modelos responsáveis pelo gerenciamento dos dados e os controllers responsáveis pela interface com o solicitante, regras de negócio e retornos. Além disso, para tornar a implementação dos controllers mais isoladas, foram adicionadas as classes de utilidade, os serializers, e os próprios serviços, que servem para extrair lógicas que estariam nos mesmos.

Exemplo: Quando um usuário se cadastra, ele deve também ser autenticado para o seu primeiro acesso à plataforma. Em vista disso, para o desenvolvimento da ação de SignUp, teríamos que fazer um método do controller com diversas responsabilidades, o que faria com o que essa classe tivesse baixa coesão. Como solução, fizemos a implementação das classes de utilidade, dos serviços e dos serializers, para separar essas responsabilidades em outras classes menores e mais especializadas.

**9.3 - Criador (Creator):**

Uma vez que as classes de serviços foram criadas para receber os parâmetros dos controllers, e manipular os dados para melhor adequar ao formato do modelo, eles se tornam a classe mais adequada para criação de instâncias, tendo também uma aplicação do padrão GoF de Factory embutido.

Exemplo: Dessa forma, para realizar a implementação da ação de criação de Cursos, o controller recebe os parâmetros através da API, e passa para o serviço CourseCreate, para realizar a criação, modificação e retorno do objeto Course.

**9.4 - Controlador (Controller):** Existe uma classe (Controlador) que manipula os eventos do usuário e faz a ligação entre a Visão (o que o usuário vê) e o Modelo (os dados). Nosso sistema em Ruby on Rails já implementa um Controller para cada recurso, tanto é que temos as classes 'CoursesController', 'LessonsController', UsersController e etc. Cada um desses controladores é responsável por manipular as solicitações do usuário para seu recurso correspondente e atualizar o modelo e a visão de acordo.

**9.5 - Especialista (Expert):**

O padrão especialista foi utilizado na construção da estrutura de serviços para determinar qual classe seria responsável por determinada alteração de dados. Dessa forma, dividimos responsabilidades entre os controllers, modelos e serviços, para garantir que cada um teria seu determinado único e exclusivo. Os modelos lidarão com as validações de dados e manipulações com a base de dados, os controllers serão responsáveis pela organização do fluxo de determinada ação na plataforma, e os serviços, como a classe mais especialista e que recebe a maior parte das informações, serão responsáveis pelas lógicas de negócio e tratamento dos casos de uso.

Exemplo: Para a manipulação do progresso do curso, baseado nas aulas vistas, foi criado o serviço LessonMarkProgress, que recebe como parâmetro o usuário autor daquela ação e a aula que está sendo manipulada, realiza as verificações necessárias e retorna a Lesson com seus novos estados.

**10. Princípios SOLID:**

**Single Responsibility Principle (Princípio da Responsabilidade Única):**

Teríamos a classe *User* na aplicação que, além de lidar com atributos e comportamentos do usuário, poderia estar manipulando a autenticação e autorização. Para evitar isso, fizemos as classes de serviço UserLogin e UserSignUp, responsáveis pelas lógicas de implementação das ações respectivas. Dessa forma, caso precisemos utilizar a mesma lógica em outro ponto do código no futuro, ela não será implementada novamente.

**Open-Closed Principle (Princípio Aberto-Fechado):**

Atualmente, os conteúdos suportados no escopo de uma lição são textos e documentos, como vídeos, imagens e pdfs.A princípio, poderiamos ter tratado as diferentes formas de exibição e tratamento dos conteúdos de uma aula com um switch. Dessa forma, no futuro, quando fosse necessário adicionar outro tipo de conteúdo, seríamos obrigados a incrementar indefinidamente esses switches. Para resolver esse problema, desenvolvemos uma classe abstrata Content, a qual é herdada pelas classes ContentText e ContentDocument e ContentVideo, responsáveis pela implementação de suas individualidades de exibição e validações de dados.

**Liskov Substitution Principle (Princípio da Substituição de Liskov):**

Não há uma utilização para esse princípio no nosso sistema. Na maioria dos casos onde precisamos de herança de classes, estamos fazendo uma herança em uma classe abstrata, sem uma implementação muito elaborada. Os casos utilizados por herança, são os casos de herança do ActiveRecord, uma classe implementada pelo framework, onde todos métodos de manipulação de dados funcionam da mesma forma em todas as classes e subclasses.

**Interface Segregation Principle (Princípio da Segregação de Interfaces):**

Assim como em Python e Javascript, em Ruby não existe a palavra reservada e o conceito de interface. Para garantir a utilização desse princípio, criamos uma ideia de contrato nas classes que herdam de Content, para tratá-la como uma interface. Nesse caso, as subclasses ContentText, ContentDocument e ContentVideo são obrigadas a implementar o método “format”, ou o código irá levantar uma exceção de NotImplemented. Dessa forma, garantimos que todos as subclasses terão o mesmo contrato, independente da forma de utilização.

**Dependency Inversion Principle (Princípio da Inversão de Dependência):**

Este princípio está relacionado à remoção de dependências de alto nível sobre detalhes de implementação de baixo nível, tornando o código mais modular e mais fácil de alterar e manter. Também não há uma necessidade de grande utilização desse princípio no nosso projeto, mas um exemplo que podemos dar é a utilização dos métodos criados nos modelos pelo ActiveRecord para fazer a manipulação dos dados no banco, sem necessidade de manipulações de baixo nível. Além disso, mantivemos os modelos com pouca dependências entre si, tornando-os cientes apenas de suas próprias implementações, e não das implementações de outros modelos.