Autores: Amós Santana, Clara Costa, Luiza Cavalcante e Saunay Coutinho
Professor: Francisco Magalhães
Disciplina: Estrutura de Dados Orientadas a Objetos

Sistema de Gerenciamento de Eventos

1. Contexto do projeto

O projeto EventManagement é uma aplicação web desenvolvida que une a gestão de eventos com a inscrição de participantes. O grande intuito desse sistema é servir como uma interface facilitadora, otimizando e automatizando as interações entre organizadores e público. Este sistema, desenvolvido em C++ com base em conceitos de Programação Orientada a Objetos, permite que administradores gerenciem e participantes se inscrevam em eventos de forma intuitiva.

2. Funcionalidades

Para Participantes:

- **Visualização de Eventos**: Consultar informações detalhadas de eventos disponíveis, incluindo data, hora e local.
- **Inscrição Simplificada**: Realizar a inscrição em eventos de interesse de forma direta.

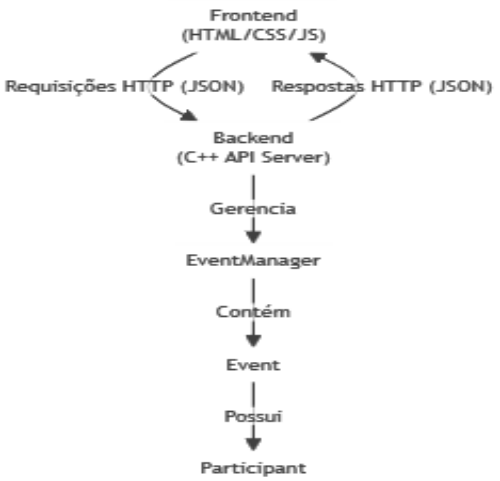
Para administradores:

- **Gestão Completa de Eventos**: Criar, editar e excluir eventos conforme a necessidade.
- **Controle de Participantes**: Visualizar a lista de inscritos para cada evento.
- **Análise de Dados**: Acessar um painel (dashboard) com estatísticas sobre os eventos e inscrições.

3. Linguagens Utilizadas

O sistema é dividido em duas grandes partes:

- Backend (C++): Responsável pela lógica de negócio, manipulação de dados e API.
- Frontend (Web): Interface gráfica para administradores e participantes.



4. Diagrama de Classes

4.1 Explicação das Classes e Herança

- Person é a classe base, com atributos comuns a qualquer pessoa.
- Participant herda de Person, adicionando um identificador único.
- Event contém uma lista de participantes e métodos para gerenciá-los.
- EventManager centraliza a gestão dos eventos.
- ApiServer faz a ponte entre o frontend e o backend, recebendo e respondendo requisições HTTP.

Classe: Person

É a **classe base** que representa uma entidade genérica de pessoa. Ela encapsula atributos comuns a qualquer indivíduo que possa interagir com o sistema, como nome, e-mail e contato.

- **Funcionalidades**:
  - Armazena e gerencia o **nome**, **e-mail** e **contato** de uma pessoa.
  - Fornece métodos para **validar** o formato do nome e do e-mail (garantindo que não sejam vazios e que o e-mail siga um padrão válido).
  - Oferece funcionalidade para **formatar números de telefone** (DDD + número) para padrões brasileiros.
  - Permite acessar e modificar seus atributos através de **getters e setters**.
  - Gera uma **representação em string** dos dados da pessoa (toString()).

Classe: Participant

Representa um participante específico de um evento. Ele estende a classe Person, adicionando um identificador único para o participante dentro do sistema.

- **Funcionalidades**:
  - Herda todas as funcionalidades de Person (nome, email, contato, validação).
  - Possui um ID único para identificação do participante.
  - Sobrescreve o método toString() para incluir o ID do participante em sua representação.
  - Métodos getId() e setId() para manipular seu identificador.

Classe: Event

Representa um evento individual no sistema. Ele armazena todos os detalhes do evento e gerencia os participantes inscritos nele.

- **Funcionalidades**:
  - Armazena e gerencia atributos como ID, nome, data, hora, local, descrição, capacidade máxima, preço, categoria e status de atividade.
  - Valida seus próprios dados (nome, local, descrição, capacidade, preço, formato de data/hora, e se a data/hora do evento é futura para novas criações).
  - Permite adicionar, buscar, atualizar e remover participantes associados a este evento. Mantém uma lista interna de ponteiros para objetos Participant.
  - Verifica a disponibilidade de vagas e se o evento está ativo para novas inscrições (canRegister()).
  - Fornece métodos para salvar seus dados e os de seus participantes associados em um arquivo.
  - Gera uma representação em string e exibe os detalhes do evento.

Classe: EventManager

É a classe controladora central do sistema. Ela é responsável por gerenciar a coleção de todos os eventos, orquestrar suas operações e persistir os dados.

- **Funcionalidades**:
  - Gerencia uma coleção de todos os objetos **Event** no sistema.
  - Fornece métodos de alto nível para adicionar, atualizar e excluir eventos.
  - Permite adicionar, buscar, atualizar e remover participantes em eventos específicos.
  - Lida com a persistência dos dados, carregando eventos de um arquivo (eventos.txt) na inicialização e salvando-os ao adicionar/modificar/remover.
  - Fornece estatísticas do dashboard (total de eventos, participantes, eventos futuros e de hoje) em formato JSON.
  - Utiliza a biblioteca json/json.hpp para formatar retornos de dados como JSON, facilitando a integração com outras partes do sistema ou interfaces.

5. Análise Geral dos Conceitos de POO Aplicados

a. Encapsulamento

Este é um conceito central e visível em todas as classes. Os atributos de cada classe são privados (**private**) ou protegidos (**protected**), o que significa que o acesso direto a eles é restrito. A manipulação desses dados ocorre exclusivamente através de métodos públicos (getters e setters), protegendo a integridade das informações. Isso garante que os dados sejam modificados apenas de maneiras válidas e controladas pelos métodos da própria classe, tornando o código mais seguro e fácil de depurar.

- **Alguns exemplos:**
  - Em Person e Participant, name, email, contact e id são privados. Você não pode simplesmente mudar o nome de uma pessoa de fora da classe; você deve usar setName().
  - Na classe Event, todos os detalhes do evento (ID, nome, data, etc.) e a lista de participantes (std::vector<Participant*> participants) são privados. Isso impede que outras partes do código alterem o estado do evento de forma descontrolada.
  - EventManager mantém sua lista de eventos (std::vector<Event*> events) e o nextEventId como privados, controlando o acesso e a modificação através de seus próprios métodos.

b. Herança

A herança permite que novas classes ("filhas") reutilizem e estendam o comportamento de classes existentes ("mães"). No EventManagement, a relação "É UM" é fundamental: ela significa que uma classe derivada é uma versão mais específica da sua classe base. Isso promove a reutilização de código e estabelece uma hierarquia clara, tornando o modelo de dados mais intuitivo e a base de código mais fácil de manter.

- **Exemplo:**
  - A classe Participant herda de **Person**. Isso significa que um participante *é uma* pessoa, e automaticamente "ganha" todos os atributos (nome, email, contato) e funcionalidades (getters, setters, validação) de Person sem precisar reimplementá-los. Participant então adiciona sua própria característica específica: um id.

c. Polimorfismo

O polimorfismo, que significa "muitas formas", permite que objetos de diferentes classes sejam tratados de maneira uniforme. Essa flexibilidade e extensibilidade são alcançadas principalmente com funções virtuais e ponteiros/referências para a classe base. O polimorfismo permite escrever código mais genérico que pode operar em objetos de diferentes tipos, desde que compartilhem uma interface comum (definida pela classe base).

- **Alguns exemplos:**

- O destrutor em Person é virtual (virtual ~Person()). Isso é crucial para garantir que, ao deletar um objeto Participant através de um ponteiro Person* (por exemplo, em uma coleção genérica de Persons), o destrutor correto de Participant seja chamado, evitando vazamentos de memória.
- O método toString() na classe Person é virtual, e é sobrescrito (**override**) na classe Participant. Isso permite que, se você tivesse uma coleção de ponteiros Person* (que poderiam apontar para objetos Participant), chamar toString() em cada um deles executaria a versão específica da classe real do objeto.

6. Associação e Composição (Relacionamentos entre Classes)

Além dos pilares formais, os relacionamentos entre as classes são cruciais para a arquitetura do projeto.

- **Exemplos:**
  - Associação (**Event** tem **Participant**s): A classe Event contém um std::vector<Participant*> participants. Isso indica que um evento gerencia múltiplos participantes. A forma como os participantes são gerenciados (ponteiros e delete no destrutor do Event) sugere uma composição forte, onde o ciclo de vida do Participant está intimamente ligado ao Event ao qual ele pertence.
  - Composição (**EventManager** tem **Event**s): A classe EventManager contém um std::vector<Event*> events. O EventManager é responsável por criar e destruir os objetos Event que ele gerencia, indicando uma relação de composição.
- **Benefício:** Esses relacionamentos modelam o mundo real de forma eficaz, dividindo as responsabilidades e permitindo que as classes colaborem para realizar as funcionalidades do sistema. A composição garante que, quando um objeto "contêiner" é destruído, seus "componentes" também o sejam (se gerenciados pelo contêiner).

7. Diagrama Complementar

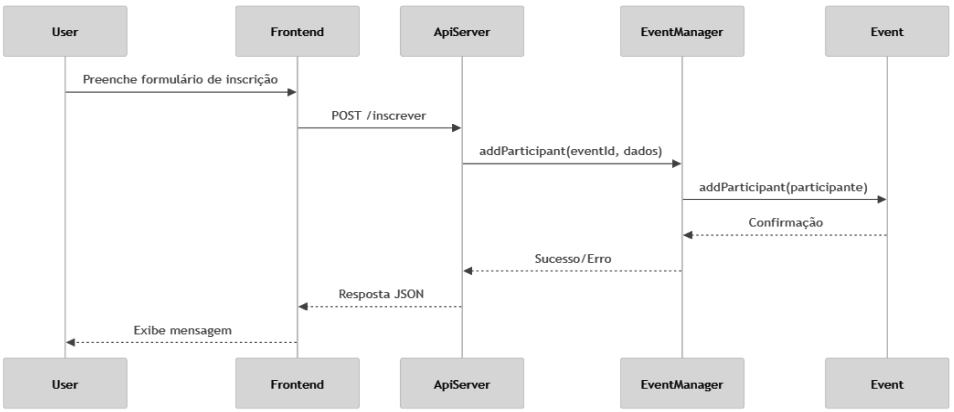


Diagrama de Sequência: Inscrição em Eventos

7. Contribuições

Nome	Contribuição Principal
Saunay	Modelagem das classes principais (Person, Participant, Event) e definição da arquitetura do backend.
Amós	Implementação das funcionalidades de manipulação de eventos e participantes, incluindo métodos de cadastro e remoção.
Luiza	Desenvolvimento da API REST em C++, integração entre as classes e tratamento das requisições HTTP.
Clara	Testes das funcionalidades do backend, documentação do código, elaboração dos diagramas e revisão técnica.

8. Considerações finais

O projeto representa uma aplicação web robusta, desenvolvida por nós estudantes do curso de Sistemas de Informação da UFPE para otimizar e automatizar a gestão e inscrição em eventos. No backend, a implementamos em C++ para usar a aplicação dos princípios de Programação Orientada a Objetos, e trabalhar com o encapsulamento para proteger a integridade dos dados, herança com a relação entre `Person` e `Participant` para reuso de código, e composição/associação entre `EventManager`, `Event` e `Participant` para uma modelagem do domínio. Além disso, trabalhamos com polimorfismo, através de destrutores virtuais e sobrescrita de métodos como `toString()`, também foi fundamental para a flexibilidade do sistema.

No frontend, a utilização de HTML, CSS e JavaScript, com uma estrutura modularizada por componentes e serviços, garantiu uma interface intuitiva e de fácil manutenção. Essa arquitetura proporciona um sistema extensível, capaz de ser adaptado e escalado para gerenciar diversos tipos de eventos e necessidades futuras.