ChatGPT

Plano Pedagógico – Projeto de Microserviços em Sistemas Corporativos (16 semanas)

Disciplina: Desenvolvimento de Sistemas Corporativos (6º Período)
Projeto Prático: Plataforma de Gestão de Eventos Corporativos (arquitetura de microserviços)
Tecnologias-Chave: Spring Boot (Java), FastAPI (Python), React (JavaScript), PostgreSQL, MongoDB, Docker, Kubernetes (conceitos), CI/CD, RabbitMQ, JWT, etc.

Cronograma Semanal Resumido (16 Semanas)

Semana	Tema Principal	Entregáveis/Atividades Avaliativas
1	Introdução a Sistemas Corporativos e Microserviços	*Definição do tema do projeto* (descrição inicial do sistema)
2	Definição de Escopo e Requisitos	Documento de Visão e Escopo (problema, requisitos, user stories) [1]
3	Modelagem de Domínio e Divisão de Microserviços	Rascunho do Modelo Conceitual (diagrama de domínio) e lista preliminar de microserviços
4	Design Arquitetural de Microserviços (Tecnologias e Padrões)	Modelo Conceitual & Arquitetura (diagramas finais, decisões) + *Esqueleto dos projetos* (repos, CI, Docker Compose) [2]
5	Protótipo de Interface de Usuário (Frontend)	Protótipo navegável da UI (wireframes ou React com dados mock) [3]
6	Definição de Contratos de API (API Gateway)	Documentação das APIs (endpoints REST de cada serviço – OpenAPI/Swagger) [4]
7	Iteração 1 – Implementação do Backend (Parte 1)	**Entrega parcial:** Serviço de **Usuários/Autenticação** implementado (cadastro/login via JWT)
8	Iteração 1 – Implementação do Backend (Parte 2)	**Entrega parcial:** Serviços de **Eventos** e **Inscrições** implementados (listar eventos, registrar participação)
9	Iteração 1 – Integração e MVP Funcional	**Entrega:** MVP Integrado (login, lista de eventos, inscrição) funcionando [5] [6]; *Demonstração em aula* + Feedback do professor
10	Iteração 2 – Novas Funcionalidades (Pagamentos, Notificações)	Serviços de **Pagamentos** (integração sandbox) e **Notificações** (e-mail) implementados e integrados ao fluxo

Semana	Tema Principal	Entregáveis/Atividades Avaliativas
11	Iteração 2 – Aprimoramentos e Serviço de Feedback	Serviço de **Feedback** (comentários/pesquisas pós-evento) implementado; melhorias de desempenho (cache Redis) e robustez (tratamento de erros, segurança) aplicadas [7] [8]
12	Iteração 2 – Consolidação e Deploy em Staging	**Entrega:** Sistema completo em ambiente de *staging* (homologação) [9] rodando via Docker/K8s; relatório de progresso (novas features e ajustes)
13	Testes Finais e Garantia de Qualidade	**Entrega:** Relatório de Testes Finais (resultados de testes end-to-end, performance básica, checklist de requisitos atendidos) [10] [11]
14	Documentação Final do Projeto	Documentação completa consolidada: Manual do Usuário, Guia do Desenvolvedor (setup, código), Documentação de Arquitetura (diagramas finais)
15	Preparação de Deploy e Revisão Geral	Deploy final em produção (ou ambiente equivalente) reproduzível (Docker Compose/K8s) [12] ; **Checkpoint:** revisão integrativa de todo conteúdo e ensaio de apresentação
16	Apresentação Final e Retrospectiva	**Entrega Final:** Apresentação do projeto (slides + demonstração ao vivo) [13] ; Retrospectiva das lições aprendidas [14]

Detalhamento Semanal

Semana 1: Introdução a Sistemas Corporativos e Arquitetura de Microserviços

- **Objetivo pedagógico:** Apresentar o contexto de sistemas corporativos e motivar a arquitetura de microserviços como solução moderna. Os alunos deverão compreender as diferenças entre sistemas monolíticos e distribuídos, e conhecer o projeto prático (plataforma de eventos corporativos) que será desenvolvido ao longo do semestre.

- **Conteúdos teóricos abordados:**

- Características de sistemas corporativos (escala, modularidade, manutenção em longo prazo).
- Monolito vs. Microserviços: conceitos, vantagens (escalabilidade independente, resiliência) e desafios (complexidade, comunicação inter-serviços) [15] [16] .
- Estudos de caso de empresas que adotaram microserviços (breves exemplos).
- Visão geral do projeto de **Gestão de Eventos Corporativos**: descrição do domínio (eventos, participantes, inscrições, etc.) e requisitos gerais.

- Introdução ao método **C.E.R.T.O.** para uso de IA generativa (ChatGPT) no apoio ao desenvolvimento [17] – apresentar a estrutura: **Contexto, Exigências, Referências, Tarefas, Observações** – e como será integrado às atividades da disciplina.

- **Práticas propostas:**

- **Formação de equipes** e discussão inicial do projeto: confirmar o tema “plataforma de eventos corporativos” e levantar em conjunto exemplos de funcionalidades (ex.: criar eventos, inscrever-se, efetuar pagamento, dar feedback).
- Configuração do ambiente de desenvolvimento: instalar/confirmar acesso às ferramentas necessárias (JDK/Spring Boot initializer, Python/FastAPI, Node/npm para React, Docker, IDEs).
- Atividade orientadora: elaboração de um *canvas* simples ou brainstorming para delimitar o problema a ser resolvido (qual necessidade dos eventos corporativos o sistema atenderá).

- **Demonstração CERTO:** O professor estrutura ao vivo uma pergunta ao ChatGPT sobre conceitos de microserviços, por exemplo: fornecer *Contexto* (explicando que a turma está iniciando microserviços), *Exigências* (resposta concisa, em linguagem simples, com exemplos reais), *Referências* (mencionar termos como “monolito”, “escalabilidade”), *Tarefas* (pedir um resumo das diferenças e benefícios), *Observações* (solicitar resposta em português). A resposta da IA complementa a introdução teórica, ilustrando o uso da técnica CERTO para obter explicações claras.

- **Aplicação do método C.E.R.T.O. na aula:**
  Exemplo prático: Ao final da aula, os alunos podem usar o ChatGPT para explorar ideias de projeto. Cada equipe formula um prompt estruturado:

- **Contexto:** “Somos estudantes desenvolvendo um sistema corporativo de eventos (inscrições, pagamentos, etc.) para um projeto semestral.”
- **Exigências:** “Precisamos definir as principais funcionalidades e escopo inicial. O sistema será desenvolvido em 4 meses por 5 alunos usando microserviços.”
- **Referências:** “Já identificamos alguns módulos: Usuários, Eventos, Inscrições, Pagamentos, Notificações, Feedback.”
- **Tarefas:** “Sugira 5 funcionalidades essenciais para uma plataforma de gestão de eventos corporativos e possíveis microserviços correspondentes.”

- **Observações:** “Responder em português de forma objetiva.”
  Essa consulta, seguindo o método CERTO, pode ajudar a validar e ampliar as ideias de escopo levantadas pela equipe, funcionando como um *brainstorming* assistido [18] [19].

- **Atividade avaliativa/entregável sugerido:** Cada equipe entrega uma breve **descrição do projeto** escolhido (tema confirmado) e um parágrafo de visão geral do problema que será resolvido. Esse documento inicial serve para alinhar entendimento com o professor. *(Avaliação formativa: feedback sobre clareza do escopo antes de prosseguir.)*

Semana 2: Definição do Escopo e Levantamento de Requisitos

- **Objetivo pedagógico:** Orientar os alunos na definição detalhada do escopo do sistema de eventos corporativos, coletando requisitos funcionais e não-funcionais e identificando atores e histórias de usuário. Ao final da semana, os alunos terão produzido um documento de visão do projeto, que servirá de referência para todo o desenvolvimento.

- **Conteúdos teóricos abordados:**

- Técnicas de elicitação de requisitos: brainstorming, entrevistas simuladas, análise de sistemas similares.
- Tipos de requisitos: **Funcionais** (funcionalidades que o sistema deve ter) vs. **Não-funcionais** (desempenho, segurança, usabilidade, etc.).

- Especificação de requisitos em formato de **user stories** (“Como [ator], eu quero [ação] para [benefício]”).
- Identificação de **atores** (tipos de usuários: ex. participante, organizador/admin do evento).
- Descrição do *escopo* do projeto na forma de Documento de Visão: problema a ser resolvido, objetivos do sistema, funcionalidades principais.
- Ferramentas de modelagem de requisitos: diagramas de caso de uso (UML) para visualizar interação atores x funcionalidades (opcional).
- Definição inicial de **domínios de negócio** e mapeamento preliminar para microserviços a partir dos requisitos (introdução à noção de *bounded contexts*).

- Decisão das tecnologias por camada (frontend, backend, banco) conforme recomendação do plano – ex.: React no front, Spring Boot/Java e FastAPI/Python no back, PostgreSQL e Mongo nos dados – já alinhando preferências da turma.

- **Práticas propostas:**

- Em equipes, **brainstorm e levantamento**: listar as principais funcionalidades esperadas no sistema de eventos (ex.: “criar evento”, “inscrever participante”, “processar pagamento”, “enviar confirmação”, “coletar feedback” etc.), junto com requisitos não-funcionais relevantes (ex.: “suportar 1000 usuários simultâneos”, “autenticação segura com JWT”).
- Elaborar uma **lista estruturada de requisitos**: pelo menos 8–12 requisitos funcionais e os principais requisitos não-funcionais. Identificar, se possível, quais módulos do sistema cada requisito afeta.
- Desenvolver **user stories** exemplares para ilustrar funcionalidades chave (ex.: “Como organizador, quero criar um evento definindo título, data, capacidade...”; “Como participante, quero me inscrever em um evento e receber confirmação”).
- Construir (opcionalmente) um **diagrama de caso de uso** mostrando atores (usuário, administrador) e casos de uso do sistema (cadastra evento, faz inscrição, efetua pagamento, etc.) para ter uma visão visual do escopo.
- Delinear uma **divisão preliminar de microserviços** baseada nos domínios identificados: por exemplo, associar requisitos aos serviços **Usuários**, **Eventos**, **Inscrições**, **Pagamentos**, **Notificações**, **Feedback**. Essa divisão inicial orientará o design posterior [20] [21].

- **Aplicar CERTO no planejamento do escopo:** as equipes refinam ou validam suas listas de requisitos consultando o ChatGPT. Por exemplo, ao listar funcionalidades, podem perguntar: *“Estamos desenvolvendo um sistema de eventos com microserviços (Contexto). Precisamos garantir que não esqueçamos requisitos importantes e atender às exigências do curso (Exigências: microserviços, 4 meses, stack definida). Já listamos X, Y, Z (Referências: mencionar funcionalidades já pensadas). Tarefa: Que requisitos ou funcionalidades essenciais talvez estejam faltando na nossa lista? Observações: responder em português considerando sistemas de eventos corporativos.”* A resposta da IA pode apontar funcionalidades não lembradas (como emissão de certificados, por exemplo), enriquecendo o escopo inicial [18].

- **Atividade avaliativa/entregável sugerido: Documento de Visão e Escopo** do projeto. Este documento inclui a descrição do problema e objetivo do sistema, atores envolvidos, lista de requisitos funcionais e não-funcionais levantados, algumas user stories de exemplo e uma proposta de divisão inicial em módulos/microserviços [22] [21]. *(Avaliação*: qualidade e completude do escopo definido, coerência com o tema proposto.)

Semana 3: Modelagem de Domínio e Identificação de Microserviços

- **Objetivo pedagógico:** Ensinar os alunos a traduzir os requisitos em um modelo conceitual e a definir claramente os limites de cada microserviço (separação de domínios). Ao final da semana, os estudantes terão um diagrama de domínio com as principais entidades e relações, e saberão quais microserviços serão implementados e suas responsabilidades.

- **Conteúdos teóricos abordados:**

- **Modelagem de domínio**: conceitos de entidades, atributos e relacionamentos; como derivar entidades a partir dos requisitos (por exemplo, Evento, Usuário, Inscrição, Pagamento, Notificação, Feedback).
- UML básico para **Diagramas de Classes/Domínio**: representação das entidades do sistema de eventos e seus relacionamentos (associações, cardinalidades).
- **Divisão em microserviços**: princípios de coesão de domínio e baixo acoplamento. Introdução a *Domain-Driven Design (DDD)* (contextos delimitados) aplicado de forma simples – cada microserviço corresponde a um *bounded context* de negócio.
- Heurísticas para identificar microserviços: separar por funcionalidade de negócio (ex.: não misturar lógica de pagamentos com lógica de eventos), avaliar se uma entidade/função pode evoluir independentemente.
- **Padrões de design** relevantes: *Database per Service* (cada microserviço com seu banco) e *API Gateway* (padrão apresentado brevemente, detalhado na semana 6).
- **Tecnologias de persistência**: decidir quais bancos usar para quais serviços (ex.: dados transacionais em PostgreSQL; dados não estruturados ou analíticos possivelmente em MongoDB ou outros – e.g. Feedback pode usar Mongo para respostas de formulário). Conceito de **polyglot persistence**.

- (Revisão rápida) Notação ER vs UML de classes para modelos de dados, conforme preferência.

- **Práticas propostas:**

- Construir em equipe o **Diagrama de Domínio** do sistema: identificar as principais entidades (por exemplo: *Usuário, Evento, Inscrição, Pagamento, Notificação, Feedback*), seus atributos básicos e os relacionamentos entre elas. Definir cardinalidades (ex.: um Evento pode ter muitos Participantes/Inscrições; uma Inscrição pertence a um Evento e a um Usuário; etc.).
- Delinear quais entidades pertencem a qual **microserviço**: por exemplo, *Usuário* no serviço de Usuários, *Evento* no serviço de Eventos, *Inscrição* no serviço de Inscrições, etc. Fazer uma tabela ou anotações ligando entidades -> serviço responsável.
- Atualizar/refinar a lista de microserviços a implementar, garantindo que cada requisito identificado na semana 2 está coberto por algum serviço. Resultará numa lista final de serviços: **Auth/Usuários**, **Eventos**, **Inscrições**, **Pagamentos**, **Notificações**, **Feedback**, + **API Gateway/ BFF**.
- Documentar a **responsabilidade de cada microserviço** (breve descrição): ex.: “Serviço de Eventos – gerencia criação/edição de eventos, listagem de eventos disponíveis; Serviço de Inscrições – gerencia registros de participação em eventos, integrando com Pagamentos para eventos pagos”, etc.
- Revisar a **escolha de tecnologias** para cada serviço conforme o plano e conhecimentos da turma: por exemplo, decidir que serviços X e Y serão em Java Spring Boot, enquanto Z em Python FastAPI (demonstrando flexibilidade de stack), ou optar por todos em uma mesma stack para facilitar (decisão justificada).

- **Aplicar CERTO na validação do modelo**: os alunos podem usar o ChatGPT para revisar sua modelagem. Ex: *"Contexto:* temos entidades Evento, Usuário, Inscrição etc para um sistema de eventos; *Exigências:* modelo deve estar consistente e normalizado; *Referências:* descrevemos relações (um usuário pode ter várias inscrições, etc.); *Tarefa:* verificar se essa modelagem faz sentido ou sugerir ajustes; *Observações:* explicar brevemente razões de possíveis ajustes."* A IA, bem instruída, pode apontar se alguma entidade está faltando ou se alguma relação parece incorreta, ajudando a refinar o modelo antes de congelá-lo.

- **Atividade avaliativa/entregável sugerido: Modelo Conceitual** inicial do sistema. Entrega do diagrama de domínio (entidades e relações) acompanhado de uma lista final dos microserviços previstos e suas responsabilidades. Esse material será parte do design final. *(Avaliação*: verificar se o modelo cobre os requisitos e se a divisão de serviços está lógica e equilibrada.)

Semana 4: Design Arquitetural de Microserviços e Decisões Tecnológicas

- **Objetivo pedagógico:** Capacitar os alunos a elaborar a arquitetura geral do sistema de microserviços e tomar decisões de design e tecnológicas informadas (padrões de comunicação, autenticação, infraestrutura). Ao final da semana, os estudantes terão diagramas de arquitetura e um projeto-base criado (repositórios, pipelining) para dar início à implementação.

- **Conteúdos teóricos abordados:**

- **Arquitetura de referência de microserviços:** apresentação de um diagrama arquitetural geral: clientes (frontend) se comunicando com um **API Gateway**, que encaminha para diversos serviços de domínio independentes, cada qual com seu banco de dados [23] [24]. Comunicação síncrona via REST e assíncrona via fila (RabbitMQ ou Kafka) para integração de eventos [25] [26].
- **Padrões de integração**: Request-Response (REST/HTTP) vs. Event-Driven (mensageria). Quando usar cada um (consultas imediatas vs. processamento assíncrono).
- **Gateway API**: responsabilidades (roteamento centralizado, autenticação central, rate limiting, etc.) – possiblidades de implementação (ex.: *Spring Cloud Gateway*, *Express.js*, *Kong/Nginx* ingress) [27]. Decisão no projeto: implementar um gateway simples customizado (ex.: um serviço Node.js ou Spring Boot leve) vs. utilizar Next.js como BFF do front.
- **Comunicação entre microserviços:** REST (sincronismo) e introdução a **mensageria** para eventos (*eventual consistency*). No contexto do projeto: ex.: enviar evento "InscriçãoRealizada" para serviço de Notificações enviar e-mail, ou "PagamentoConfirmado" para atualizar inscrição. Visão geral do RabbitMQ/Kafka (topologia básica de filas/tópicos) [28] [29].
- **Banco de dados por serviço:** reforçar a prática de cada serviço ter seu repositório de dados isolado [30]. Decisões do projeto: PostgreSQL será usado para serviços transacionais (Usuários, Eventos, Inscrições, Pagamentos), MongoDB para dados não relacionais (Feedback), Redis possivelmente para cache. Ferramentas de **migração de esquema** (Flyway, Prisma) citadas como boas práticas para versionar o DB.
- **Autenticação e Autorização:** estratégia JWT (JSON Web Token) para sessões stateless entre front e serviços [31] [32]. Desenhar como será o fluxo: o serviço de Usuários/Auth valida login e emite JWT; o gateway e/ou serviços verificam o token nas requisições subsequentes. Conceito de autorização por papel (p.ex.: admin vs usuário comum) se aplicável para funcionalidades como criar eventos.
- **Infraestrutura DevOps:** uso de Docker para conteinerizar serviços e bancos [33]. Conceito de orquestração com Kubernetes (apresentar ideia de pods, cluster, scaling) – dado o tempo, possivelmente apenas teórico ou demonstrativo. CI/CD pipelines: breves conceitos (integração contínua rodando testes, entrega contínua fazendo deploy automático) [34] [35] – será aplicado na prática básica na configuração inicial.

- **Boas práticas de código e colaboração:** controle de versão (GitHub/GitLab) – decidir se usar repositórios separados por serviço ou monorepo; convenção de branch e pull requests para colaboração [36] [37]. Padronização de estilo de código, uso de README para cada serviço.

- **Práticas propostas:**

- Elaboração do **Diagrama de Arquitetura de Microserviços** final: representando todos os serviços identificados e suas interações principais (setas indicando chamadas REST do Gateway para cada serviço, e setas pontilhadas ou eventos para integrações via mensageria). Incluir no diagrama componentes de infraestrutura: API Gateway, Message Broker, Bancos de Dados para cada serviço, e componentes como serviço de autenticação separado se aplicável [38] [39].
- Documentação das **decisões arquiteturais**: produzir um breve documento (ou seção) textual descrevendo as escolhas feitas e por quê – ex.: "Optamos por Spring Boot no serviço X devido à robustez e familiaridade da equipe, e FastAPI no serviço Y pela rapidez em prototipação, ilustrando poliglotismo. Utilizaremos JWT para autenticação por ser stateless e adequado a microserviços. RabbitMQ será usado para notificações assíncronas, garantindo baixo acoplamento entre Inscrições e Notificações" [40] [41]. Incluir como serão tratados requisitos não-funcionais como escalabilidade (ex.: conteinerização e possibilidade futura de Kubernetes), segurança (JWT, sanitização) etc.
- **Criação dos repositórios e esqueleto dos serviços:** inicializar cada microserviço com um projeto básico:
    - Serviço de Usuários/Auth: criar projeto Spring Boot (ou NestJS .***) com dependências Web, JPA, Security (ou FastAPI base, etc.).
    - Repetir para Eventos, Inscrições, Pagamentos, Notificações, Feedback (pode-se usar um *template* para agilizar se disponível).
    - Configurar repositório Git para cada (ou um monorepo estruturado por pastas de serviço), já com .gitignore adequado, licença, README inicial.
    - Definir estrutura de pacotes/módulos de cada serviço (MVC separação, etc.) mesmo que ainda vazio.
- **Pipeline CI inicial:** adicionar configuração de *Continuous Integration* (ex.: GitHub Actions workflow ou GitLab CI) para build e testes automáticos de pelo menos um serviço. Exemplo: um workflow YAML que faça build do projeto e execute testes em cada push, para inculcar a prática de CI [34]. *(Mesmo que ainda não haja muitos testes, configurar para detectar build quebrado.)*
- **Dockerização:** criar um **Dockerfile** para cada microserviço, permitindo empacotá-lo em container [41]. Exemplo: Dockerfile do serviço Usuários baseado em openjdk-alpine para Spring Boot jar, outro para FastAPI com Python base image. Fazer o mesmo para bancos (pode usar imagens oficiais de Postgres, Mongo, RabbitMQ).
- Montar um **docker-compose.yml** que orquestre todos os contêineres do sistema (serviços + banco de dados + broker), configurando redes e variáveis mínimas [41]. Nesta fase inicial, pode-se colocar cada serviço para rodar em porte fixas e apenas retornar algo simples (ex.: endpoint "/health" ou "/hello").
- **Hello World distribuído:** executar o docker-compose e verificar se todos os contêineres sobem corretamente e se os serviços respondem em seus endpoints básicos (ex.: chamar GET /hello de cada serviço via API Gateway ou diretamente) [42]. Isso valida a viabilidade do setup e da arquitetura proposta antes de avançar.

- **Aplicar CERTO na revisão do design:** antes de finalizar, os alunos podem pedir ao ChatGPT uma validação final da arquitetura. Por exemplo: *"Contexto:* Descrevemos nossa arquitetura de microserviços para eventos (listando serviços e interações); *Exigências:* deve ser escalável e seguro; *Referências:* mencionamos tecnologias escolhidas (Spring Boot, FastAPI, JWT, RabbitMQ); *Tarefa:* avaliar se a arquitetura cobre bem os requisitos e sugerir melhorias ou apontar riscos;

Observações: focar em segurança de dados e manutenibilidade.”* A IA pode responder com recomendações (por ex., sugerir adicionar serviço de configuração centralizada ou atenção à observabilidade), as quais a equipe discute se devem ser incorporadas [43].

- **Atividade avaliativa/entregável sugerido: Modelo Conceitual e Design Arquitetural** – entrega do conjunto completo de artefatos de design:

- Diagrama de Domínio (refinado se houve ajustes).
- Diagrama de Arquitetura de Microserviços.
- Documento textual de decisões arquiteturais e tecnológicas (incluindo tratamento de requisitos não-funcionais).
- **Comprovante de setup:** link dos repositórios criados e captura de tela ou log do Docker Compose executando todos os serviços com sucesso (ex.: mensagens “Hello World” de cada serviço).
  (Avaliação: coerência e integridade da arquitetura proposta, aderência às boas práticas recomendadas [44] [41]; bônus pela iniciativa em CI/CD e dockerização já demonstrada.)

Semana 5: Protótipo de Interface e Experiência do Usuário (Frontend)

- **Objetivo pedagógico:** Abordar aspectos de front-end e usabilidade, permitindo que os alunos visualizem e validem a experiência do usuário antes da implementação completa. O objetivo é que projetem as telas principais da aplicação de eventos corporativos e entendam como o frontend interagirá com os microserviços.

- **Conteúdos teóricos abordados:**

- **UX/UI em aplicações corporativas:** princípios de design de interface (consistência, feedback ao usuário, acessibilidade básica).
- Ferramentas de prototipação rápida: apresentação do Figma (ou similar) para wireframes e protótipos clicáveis *low-fidelity* vs. *high-fidelity*.
- **Bibliotecas e frameworks frontend:** visão geral do React e seu uso em SPAs; citar Next.js caso haja interesse em SSR, mas o foco será React CRA ou Vite para uma SPA. Design system básico – talvez utilização de biblioteca de UI (Material-UI, Ant Design, Bootstrap ou Tailwind CSS) para agilizar estilização.
- **Componentização e estado** no React (breve – suposição de base prévia dos alunos, senão, relembrar conceitos fundamentais).
- Integração Frontend–Backend: como o front consumirá as APIs definidas – uso de fetch/Axios para chamadas HTTP, manuseio de JSON, e importância de alinhar com contratos definidos.
- Roteamento no front-end (React Router) para navegação entre páginas (ex.: tela de login, tela de lista de eventos, tela de detalhes do evento/inscrição).

- Importância de protótipos para **validação precoce**: identificar falhas de usabilidade antes de codificar.

- **Práticas propostas:**

- **Protótipo de Telas:** Em equipes, delinear as telas principais do sistema de eventos. Recomenda-se incluir: Tela de Login/Cadastro de usuário, Tela de Lista de Eventos disponíveis, Tela de

Detalhe de Evento (com opção de se inscrever), Tela de Confirmação de Inscrição/Pagamento, e talvez Tela de Dashboard/Admin (para organizadores criarem eventos ou visualizarem inscritos).
    - Primeiro, rascunhar **wireframes em papel ou quadro** para discutir layout e elementos necessários.
    - Em seguida, usar uma ferramenta (ex.: Figma) para criar um **protótipo navegável** dessas telas, estabelecendo links de navegação (ex.: do login bem-sucedido para lista de eventos, da lista para detalhe ao clicar em um evento).
    - Alternativamente (ou em complemento), implementar um **protótipo de frontend em React**: criar componentes estáticos para as páginas acima, utilizando dados mock (ex.: array estático de eventos no código) [45] . O foco é na estrutura visual e navegação, não em dados reais ainda.
- Aplicar conceitos de design: adicionar logos/título, elementos de formulário (campos de texto para login), listas ou cards para eventos, botões de ação ("Inscrever-se", "Pagar", etc.). Garantir que o protótipo reflita o fluxo de usuário de forma lógica.
- **Apresentação interna do protótipo:** cada equipe apresenta para outro grupo ou para o professor o fluxo navegável, coletando feedback sobre usabilidade e completude (ex.: "Falta uma confirmação aqui", ou "E se o usuário quiser cancelar inscrição?" – anotar essas observações).
- Ajustar o protótipo conforme feedback recebido.
- **Aplicação do método CERTO:** Os alunos podem envolver a IA para obter sugestões de design ou fluxo. Exemplo: *"Contexto:* Estamos desenhando a interface de um sistema de inscrição em eventos corporativos; *Exigências:* a interface deve ser amigável para usuários não técnicos e responsiva; *Referências:* já criamos telas de login, lista de eventos, detalhe do evento; *Tarefa:* sugerir melhorias de UI/UX ou elementos que não devemos esquecer (ex.: botão de logout, indicação de vagas restantes); *Observações:* resposta em tópicos objetivos."* A IA pode listar ideias (como incluir busca de eventos, filtro por categoria, etc.), das quais os alunos avaliam a pertinência.

- Se o tempo permitir e a turma já tiver familiaridade básica, iniciar a **configuração do projeto React** real: criar um novo app React, instalar bibliotecas de UI necessárias, configurar roteamento básico, e talvez implementar 1–2 componentes estáticos do protótipo no código.

- **Atividades avaliativas/entregáveis sugeridos: Protótipo navegável da interface (UI)** – pode ser um link Figma ou screenshots das telas chave, ou a aplicação React rodando com telas estáticas [46] . Os critérios de avaliação envolvem: completude das telas principais, alinhamento com os requisitos (todas funcionalidades chave representadas), clareza e usabilidade do fluxo proposto. *(Avaliação formativa:* feedback do professor sobre o design da interface, requerendo ajustes caso algo não atenda aos requisitos do sistema.)

Semana 6: Definição de Contratos de API e Arquitetura de Comunicação

- **Objetivo pedagógico:** Garantir que front-end e back-end estejam em sintonia por meio de contratos de API bem definidos. Os alunos aprenderão a especificar endpoints REST de forma padronizada e preparar o terreno para que implementação de front e back aconteçam em paralelo sem ambiguidades. Ao final da semana, haverá uma documentação clara das APIs de cada microserviço (contrato) e possivelmente um API Gateway básico configurado.

- **Conteúdos teóricos abordados:**

- **Design de APIs RESTful:** boas práticas para definição de endpoints (uso de substantivos no plural, verbos HTTP adequados – GET/POST/PUT/DELETE, códigos de status apropriados para cada situação).

- Estrutura geral de endpoints para nosso domínio, exemplo: */api/eventos* (GET lista todos eventos, POST cria novo evento), */api/eventos/{id}* (GET detalhe, PUT atualizar, DELETE remover), */api/inscricoes* (POST inscrever um usuário num evento), etc. Incluindo formato JSON de request/response esperado.
- **Documentação de API:** introdução ao OpenAPI/Swagger – escrever especificações YAML/JSON para descrever endpoints, parâmetros, schemas de dados e possíveis respostas. Benefícios: gerar client/server stubs, UI de teste (Swagger UI).
- Ferramentas práticas: Swagger Editor, libraries como Springdoc (Spring Boot) para gerar docs automaticamente via anotations, FastAPI docs (automatic Swagger UI).
- **API Gateway vs. comunicação direta:** discutir se o front-end chamará um único endpoint do Gateway ou cada serviço. Provavelmente, definiremos que o front se comunica via um **API Gateway** central (por simplicidade, e para aplicar padrão gateway). Então no contrato deve-se considerar endpoints do gateway e seu roteamento para serviços internos.
- **Segurança nas APIs:** definição de headers de autorização (ex.: envio de token JWT no Authorization header), proteção de certos endpoints (ex.: */api/eventos/criar* só acessível se role=admin). Também políticas de CORS caso front esteja em domínio diferente (conceito).

- **Sequência de chamadas em um fluxo:** exemplo de um diagrama de sequência integrando serviços: "Usuário faz inscrição" – Frontend -> Gateway -> Serviço Inscrição -> Serviço Pagamento (se evento pago) -> Serviço Notificação. Mostrar como representar essa interação para entender orquestração [47] [48].

- **Práticas propostas:**

- **Definição dos endpoints de cada microserviço:** Em equipes, listar para cada serviço os endpoints REST que serão necessários, com métodos e descrições. Por exemplo:
  - *Serviço Usuários/Auth:* `POST /auth/signup` (cadastra usuário), `POST /auth/login` (retorna JWT), `GET /auth/me` (dados do perfil logado), etc.
  - *Serviço Eventos:* `GET /eventos` (lista eventos), `GET /eventos/{id}` (detalhe), `POST /eventos` (criar novo – protegido), etc.
  - *Serviço Inscrições:* `POST /inscricoes` (realizar inscrição do usuário logado em um evento), `GET /inscricoes?usuario={id}` (listar inscrições do usuário ou de um evento).
  - *Serviço Pagamentos:* `POST /pagamentos` (processar pagamento para uma inscrição, com dados de cartão fictícios ou referência à inscrição).
  - *Serviço Notificações:* talvez `POST /notificacoes/email` (enviar e-mail – mas se for exclusivamente reativo via fila, pode não expor endpoint público).
  - *Serviço Feedback:* `POST /feedback` (enviar feedback para evento), `GET /feedback?evento={id}` (listar feedbacks de um evento para admin).
  - **API Gateway:** definir roteamentos agregados se necessários – por ex, o front faz `POST /api/inscricaoCompleta` para um endpoint do Gateway que internamente orquestra inscrição + pagamento, mas isso pode complicar. Alternativamente, gateway apenas expõe caminhos diretos para cada serviço (prefixos /auth, /eventos, /inscricoes etc.). Decidir e documentar a abordagem.
- **Documentar contratos:** Criar um documento (ou coleção de arquivos YAML) **OpenAPI** listando todos os serviços e seus endpoints, com:
  - Path, método, descrição.
  - Estrutura dos dados JSON esperados e retornados (ex.: definir esquema *Evento* com campos id, nome, data, etc.).
  - Códigos de status padrão (200 OK, 201 Created, 400 Bad Request nas validações, 401 Unauthorized se faltar token, 404 Not Found, etc.).

    - Se não usar OpenAPI formal, pode ser uma tabela para cada serviço com essa informação. O importante é ter o contrato claro [4].
- Validar se **front-end** e **back-end** estão alinhados: usar as telas do protótipo para verificar: "Na tela X precisaremos de um endpoint que forneça dados Y". Exemplo: tela de lista de eventos consome GET /eventos; tela de detalhe consome GET /eventos/{id}; ação de inscrição usa POST /inscricoes com corpo {eventoId, talvez userId se não deduzir do token, etc.}.
- Criar eventualmente **stubs**: por exemplo, implementar nos serviços controladores vazios ou métodos que retornem dados dummy com o formato correto. Ou configurar no API Gateway ou frontend um *mock server* para permitir front-end testar integração antes do back real ficar pronto.
- **Integrar Swagger:** se a opção for gerar automaticamente, já habilitar em cada microserviço a página Swagger UI (ex.: Springdoc ou FastAPI docs).
- Elaborar um **Diagrama de Sequência** (opcional, se houver tempo) para um fluxo complexo, confirmando o contrato e interação: por exemplo, fluxo de inscrição paga – mostra como front chama Inscrição (passando token), Inscrição chama Pagamento, Pagamento retorna status, Inscrição salva e retorna confirmação, Notificação é acionada assincronamente via mensagem. Isso ajuda a equipe a entender a *coreografia* ou orquestração necessária.
- **Aplicação do CERTO:** Os alunos podem usar o ChatGPT para revisar seus contratos de API. Por exemplo: *"Contexto:* definimos endpoints REST para serviços de eventos e inscrições; *Exigências:* seguir padrões REST e cobrir todos casos de uso; *Referências:* listar brevemente alguns endpoints definidos; *Tarefa:* verificar se a API está coerente e completa para as funcionalidades planejadas, e sugerir melhorias (como inclusão de paginação, filtros, etc.); *Observações:* —."* A IA, conhecedora de padrões REST, pode apontar endpoints faltantes ou não consistentes, ajudando a aprimorar a especificação.

- Em paralelo, se a turma for dividida em times front/back, o **time de back-end pode começar codificação básica**: por exemplo, implementar os modelos (classes ou schemas) e controladores com métodos vazios retornando resposta dummy (ex.: lista de eventos hardcoded). O time de front-end pode configurar chamadas fetch/Axios conforme o contrato (mesmo que apontando para mocks). Assim, iniciam integração gradual.

- **Atividade avaliativa/entregável sugerido: Especificação de APIs e Contratos** – documento (ou coleção Swagger/OpenAPI) descrevendo todas as APIs do sistema, servindo de contrato entre frontend e backend [4] [49]. *(Avaliação:* verificar se todos os requisitos identificados estão cobertos por algum endpoint, consistência de nomenclatura e uso correto de verbos HTTP/ status; documentação clara o suficiente para que um desenvolvedor externo pudesse entender como usar as APIs.)

Semana 7: Iteração 1 – Implementação do Backend (Parte 1: Usuários e Autenticação)

- **Objetivo pedagógico:** Iniciar a implementação do projeto focando nos serviços fundamentais de segurança e identidade. Nesta semana os alunos irão desenvolver o serviço de Usuários/ Autenticação, estabelecendo o mecanismo de login via JWT e integrando-o ao API Gateway, bem como configurar o acesso ao banco de dados para persistir usuários. É dada ênfase a práticas de implementação (estrutura de código, teste básico) e ao uso de IA para apoio em dúvidas de codificação.

- **Conteúdos teóricos abordados:**

- **Revisão rápida de Spring Boot e/ou FastAPI**: estrutura de um projeto, controllers (endpoints), services (lógica), repositories/DAOs (acesso a dados). Anotações comuns (Spring) ou decorators (FastAPI) para criar rotas.
- **ORM e Acesso a Dados:** no Spring, usar Spring Data JPA com entidade User mapeada para tabela (campos id, nome, email, senha, etc.); no FastAPI, usar SQLModel/SQLAlchemy ou drivers para interagir com PostgreSQL. Discutir migrations (Flyway, Alembic) se aplicável, ou deixar scripts SQL.
- **Autenticação JWT na prática:** geração de token JWT (header, payload, signature), bibliotecas para isso (ex.: jjwt no Spring, PyJWT no Python). Configurando expiracão do token, secret key.
- No Spring Security: configuração de filtro JWT (ou usar Spring Security com OAuth2 JWT if experienced) – provavelmente optar por manual/simple approach: criar endpoint login que retorna token; criar um filtro que verifica Authorization header nas demais requisições.
- **Hash de senhas:** importância de armazenar senha hash (BCrypt) ao cadastrar usuário – integrar BCrypt library no Java ou relevant lib em Python.
- **Boas práticas de implementação:** logs básicos (registrar eventos importantes, mas evitar expor dados sensíveis), tratamento de erros (ex.: retornar 400 se email já cadastrado, 401 se credenciais inválidas).

- Escrever **testes unitários simples**: exemplificar criando um teste para um método de serviço (ex.: validar que criação de usuário salva no repo, senhas são encriptadas).

- **Práticas propostas:**

- **Implementar Serviço de Usuários/Auth:**
    - Criar entidade/Model **User** (campos id, nome, email, senhaHash, role etc.).
    - No banco (PostgreSQL), criar tabela `users`. Aplicar migração ou script SQL inicial.
    - Implementar repositório (Spring Data JpaRepository ou equivalente) para User.
    - **Endpoint SignUp:** `POST /auth/signup` – receber dados do usuário, validar (email não usado, força de senha mínima), fazer hash da senha, salvar no banco, retornar sucesso ou dados básicos do novo usuário.
    - **Endpoint Login:** `POST /auth/login` – receber email e senha, verificar no banco, comparar hash, se ok gerar JWT (incluir claims como userId, role), retornar token. Se falhar, retornar 401.
    - **JWT generation e validation:** escrever método para gerar token (usar secret do config) e configurar middleware/filtro:
    - No Spring: configurar filtro que intercepta requests, lê header Authorization “Bearer token”, valida token (usando secret), e popula contexto de segurança (Authentication) ou anexa userId no request.
    - No FastAPI: usar dependency (Depends) ou middleware JWT verification for protected routes.
    - **Endpoint Me (opcional):** `GET /auth/me` – retorna info do usuário logado (usar userId do token para buscar e retornar nome/email).
- **Integração com API Gateway:**
    - Se o Gateway for um serviço separado, configurá-lo para rotear as chamadas /auth/* para o serviço de Usuários. Isso pode ser feito com um simples proxy (ex.: API Gateway em Node usando http-proxy-middleware, ou no Spring Cloud Gateway definindo rotas).
    - Assegurar que o Gateway permite o login e signup públicos, mas exige token JWT nas rotas protegidas (pode delegar verificação aos serviços internos ou verificar no próprio gateway).
    - Testar manualmente: executar o serviço Auth e Gateway (Docker ou IDE) – fazer um POST signup e login via Gateway e obter token; em seguida chamar (via Gateway) um endpoint

dummy protegido de outro serviço com o token para ver se bloqueia/permite conforme esperado.

- **Feedback rápido:** testar no front-end (ou via Postman) a chamada de login. Talvez adaptar a tela de login do protótipo React para realmente chamar o endpoint e autenticar, exibindo erro ou sucesso. Isso dá visibilidade imediata do progresso.
- **Uso de IA no desenvolvimento:** incentivar alunos a consultar ChatGPT para dúvidas de implementação específicas. Exemplos:
  - Se travarem na configuração do Spring Security JWT, podem perguntar: *"Contexto:* projeto Spring Boot, preciso autenticar via JWT; *Exigências:* usar filtro no Spring Security; *Referências:* erro X que está dando ou trecho de código atual; *Tarefa:* identificar o erro e corrigir configuração; *Observações:* ..."*. A IA pode ajudar a encontrar o erro de configuração ou indicar passos (como registrar o filtro na chain) [50] .
  - Para FastAPI, se dúvida em usar OAuth2PasswordBearer, solicitar exemplo.
  - Para hashing, pedir exemplo de uso do BCrypt no Spring, etc.
- **Testes unitários básicos:** escrever pelo menos um teste no serviço de usuários:
  - Ex.: testar que ao chamar função de cadastro com senha, o usuário salvo tem senha encriptada (não igual à original).
  - Ou teste de validação de login com senha correta/incorreta.
  - Configurar esses testes para rodar no pipeline CI (garantir que passam).

- **Controle de versão:** fazer commits significativos e abrir merge request se usando branches, para revisão de código pelo professor/colegas.

- **Atividade avaliativa/entregável sugerido: Código do Serviço de Usuários/Auth funcional**, integrado com JWT. Evidências:

- Endpoint de *SignUp* e *Login* operantes (por exemplo, captura de requisição/resposta no Postman mostrando login gerando token).
- JWT emitido e verificado nos demais serviços (pelo menos confirmando que outros serviços rejeitam requisição sem token ou com token inválido).
- Código fonte versionado no repositório, pipeline CI passando.

Avaliação: corretude da implementação (ex.: consegue autenticar e proteger rotas conforme esperado), qualidade do código (organização em camadas, uso de hashing de senha, tratamento de erros adequadamente), e adoção de boas práticas (variáveis de ambiente para secrets, logs, etc.).

Semana 8: Iteração 1 – Implementação do Backend (Parte 2: Eventos e Inscrições)

- **Objetivo pedagógico:** Desenvolver os microserviços principais de negócio – Eventos e Inscrições – conectando-os ao banco de dados e integrando suas operações. Os alunos aprenderão sobre comunicação interna entre serviços (sincrona e/ou assíncrona) e consolidarão o uso de ORMs, validação de dados e lógica de negócio básica. Ao final da semana, deverá ser possível criar e listar eventos e realizar inscrições através dos serviços, com autenticação aplicada.

- **Conteúdos teóricos abordados:**

- **Implementação de API REST** no contexto do domínio:
  - Serviço de **Eventos**: Create, Read, Update, Delete eventos (CRUD básico) – mas talvez limitar a criação/edição a um usuário admin.

    - Serviço de **Inscrições**: criação de inscrição vinculada a evento e usuário; regras de negócio (ex.: não permitir duplicatas, respeitar capacidade do evento).
- **Comunicação síncrona entre microserviços:** necessidade de um serviço consultar outro: por exemplo, Inscrições precisar confirmar via Eventos se há vaga disponível em determinado evento, ou obter detalhes do evento. Abordagens:
    - Simplificada: o front-end já fornece as info necessárias (eventId) e Inscrições confia que eventId exista – validação mínima chamando Evento.
    - Chamada REST interna: Inscrições faz GET em Eventos/{id} (como se fosse um cliente) para validar e talvez reduzir contagem de vagas.
    - **Client HTTP**: usar WebClient/RestTemplate (Java) ou requests (Python) para chamadas internas; tratar timeout, erro (ex.: se serviço Eventos off).
- **Comunicação assíncrona (introdução prática):** conceito de publicar evento de domínio. Ex.: quando uma inscrição é confirmada, publicar mensagem "InscriçãoRealizada" em uma fila. Vantagem: outros serviços (Notificações, ou o próprio Eventos para contagem) podem reagir sem acoplamento direto.
    - Introduzir rapidamente RabbitMQ: exchanges, queues, routing keys.
    - Libraries: Spring AMQP, Pika (Python) etc.
    - Decidir um pequeno caso para aplicar: e.g., Notificações usará mensagem para envio de email, então Inscrições ao salvar envio publica mensagem com email do usuário e detalhes do evento.
- **Transações e consistência eventual:** discutir que em microserviços distribuídos não dá pra garantir transação ACID abrangendo dois serviços (inscrição + atualização de evento), então ou se opta por simplicidade (Inscrição chama Eventos dentro de sua transação – lock eventual, não ideal) ou aceita consistência eventual (inscrição salva e separadamente evento decrementa vaga).
    - Mencionar padrões como Saga (não implementar, apenas conscienciar).
- **Validações de negócio:** exemplificar como verificar capacidade do evento:
    - Versão simples: campo `vagas` em Evento; ao criar inscrição, checar quantas inscrições já existem para aquele evento (consulta contagem no DB Inscrições ou flag no Evento).
    - Ou manter `vagasRestantes` no evento e decrementá-lo – cuidado com concorrência se multiple signups concurrently (mas não aprofundar demais).
    - Decisão do projeto: pode optar por simplificar e não implementar controle de vagas na MVP, focando no fluxo principal.
- **Relaciones entre entidades cross-service:** como armazenar referências:
    - Inscrição armazena `eventoId` e `usuarioId`. Não usar foreign key real (pois bancos separados), mas garantir integridade via lógica da aplicação (ex.: verificar ids existentes via API).

- **Teste de integração simples:** explicar noções de testes integrados multi-serviço (ex.: usando docker-compose de teste ou banco em memória) – possivelmente não implementado agora, mas mencionar que após implementar Inscrições, seria bom testar criando evento e inscrição juntando serviços.

- **Práticas propostas:**

- **Implementar Serviço de Eventos:**
    - Entidade **Evento** (campos: id, nome, descrição, data/hora, local, capacidade/vagas totais, talvez flag se pago e valor).
    - Criar repositório para Evento (tabela events no PostgreSQL).
    - **Endpoint GET /eventos:** retorna lista (possivelmente paginada) de eventos. No MVP, pode retornar todos.

    - **Endpoint GET /eventos/{id}:** retorna detalhes do evento (ou 404 se não encontrado).
    - **Endpoint POST /eventos:** (se aplicável para admins) cria novo evento – valida dados (ex.: data futura, campos obrigatórios).
    - **Endpoint PUT /eventos/{id}:** atualiza evento (ou usar PATCH para status de vagas, etc.).
    - **Endpoint DELETE /eventos/{id}:** remove evento (se preciso).
    - Proteger criação/edição/remoção com autorização (JWT role=admin).
    - Se aplicável, criar alguns eventos iniciais no DB (se não tiver interface de criação para popularem algo para teste).
- **Implementar Serviço de Inscrições:**
    - Entidade **Inscrição** (id, eventoId, userId, timestamp, talvez status – pendente/confirmada).
    - Repositório Inscrição (tabela registrations).
    - **Endpoint POST /inscricoes:** realiza inscrição do usuário autenticado em um evento:
    - Extrair userId do JWT (no header via Gateway ou repassado).
    - Receber eventoId (no corpo ou rota).
    - Verificar se já existe inscrição daquele user para aquele evento (evitar duplicar) – se sim, retornar erro 409 ou 400.
    - **Regra de vagas:** opcional – checar se evento ainda tem vaga:
        - Simplificado: contagem de inscrições existentes para eventoId e comparar com evento.capacidade (requere chamar serviço de Eventos para obter capacidade ou manter contagem).
        - Poderiam aqui demonstrar chamada interna: do serviço Inscrições fazer um GET no serviço Eventos/{id}:
        - Se resposta indica vagas ou capacidade, decidir se pode inscrever.
        - Se evento não encontrado ou lotado, retornar erro apropriado.
    - Se tudo ok: salvar nova inscrição no banco (associando userId e eventoId).
    - Se evento era pago: talvez marcar inscrição com status “PENDENTE PAGAMENTO” inicialmente ou já iniciar processo de pagamento (ver abaixo, possivelmente delegar para serviço Pagamentos).
    - Retornar sucesso (201 Created) ou detalhes da inscrição criada.
    - **Endpoint GET /inscricoes?usuario={id}:** (ou `/minhas`) retorna inscrições do usuário logado (para ele ver em quais eventos está inscrito).
    - **Endpoint GET /inscricoes?evento={id}:** retorna inscrições de um evento (para admin ver lista de participantes).
    - (Esses endpoints GET podem ser implementados depois se necessário; foco principal é conseguir inserir inscrição).
- **Integração Pagamento (esboço):** se evento for marcado como pago, definir como sinalizar:
    - Poderia ser: no momento da inscrição, se event.pago, então:
    - ou Inscrições chama sincronicamente o serviço de Pagamento para processar (passando user e event info).
    - ou Inscrições cria registro com status pendente e retorna indicando que requer pagamento, e front-end então chamaria endpoint do Pagamento.
    - Para MVP, talvez não processar pagamento real ainda, só simular:
    - Poderíamos ignorar pagamento na iteração 1 (tratar todos eventos como gratuitos para focar no fluxo principal). Deixar pagamento para iteração 2.
    - Decisão: para simplificar MVP, consideraremos eventos gratuitos inicialmente, permitindo fluxo contínuo.
- **Publicar evento de inscrição para Notificações:**
    - Após criar inscrição com sucesso, implementar publicação em RabbitMQ:
    - Configurar RabbitMQ container (se não já, no docker-compose).

    - No serviço Inscrições, produzir mensagem com informações mínimas (ex.: eventoId, user email ou userId).
    - No serviço Notificações (a ser implementado na iteração 2), isso será consumido.
    - Teste: por enquanto, pode-se colocar um *consumer temporário* que simplesmente loga a mensagem (ou notificação service como stub que loga).
    - Se alunos não conseguirem a tempo, deixar apenas preparado (ex.: uma função publishMessage que pode estar ociosa).
- **Testes e validações:** testar via Postman ou integrando com front:
    - Criar 1-2 eventos via endpoint (ou direto no DB) para ter dados.
    - Usar um usuário logado para chamar POST /inscricoes no evento e verificar resposta.
    - Tentar cenários: inscrever duas vezes mesmo user (ver se bloqueia), inscrever em evento inexistente (deve dar 404/erro).
    - Verificar no banco se registros foram criados adequadamente.
    - Se front-end já consegue consumir, tentar integrar na tela de lista de eventos um botão inscrever que chama a API (pode mostrar alert de "Inscrição realizada").
- **Uso de ChatGPT durante implementação:** exemplos:
    - Consultar como realizar requisições HTTP internas: *"Como chamar uma API REST externa dentro de um serviço Spring Boot (Contexto), Exigências: usar RestTemplate ou WebClient, passando token JWT, etc."*
    - Para RabbitMQ: *"Exemplo de código para publicar mensagem RabbitMQ em Java/Python"*.
    - Depuração: se Inscrições retorna erro, usar ChatGPT fornecendo o stack trace (nas Observações) para ajuda em identificar causa.
- **Commitar e integrar**: push do código desenvolvido; atualizar CI if needed (novos tests).

- **Atualizar Docker Compose:** incluir serviços Eventos e Inscrições na configuração, linká-los ao banco. Verificar se todos sobem.

- **Atividade avaliativa/entregável sugerido: Código dos Serviços de Eventos e Inscrições implementados**:

- Deve ser possível listar eventos e criar novas inscrições via APIs.
- Apresentar uma **demonstração interna**: por exemplo, usar o front-end (ou Postman) para exibir um usuário navegando pelos eventos e se inscrevendo com sucesso.
- *(Avaliação:* funcionalidade dos serviços conforme requisitos – ex.: consegue-se cadastrar evento e inscrever usuário; integridade mantida (sem duplicatas); aplicação correta de autenticação – somente usuários logados inscrevem, somente admins criam eventos; e qualidade do código – uso de validações e padrões adequados.)

Semana 9: Iteração 1 – Integração Total e MVP (Produto Viável Mínimo)

- **Objetivo pedagógico:** Consolidar o trabalho da primeira iteração integrando todos os componentes desenvolvidos (frontend, gateway e microserviços) em um sistema mínimo funcional (MVP). Os alunos irão testar o fluxo completo do aplicativo e corrigir problemas de integração, vivenciando a importância de *end-to-end testing*. Ao final da semana, deverão apresentar o MVP rodando e refletir sobre os ajustes necessários, recebendo feedback formal (checkpoint do projeto).

- **Conteúdos teóricos abordados:**

- **Integração de componentes:** desafios comuns ao integrar front-end com múltiplos serviços (CORS, formatos JSON incompatíveis, versões de API divergentes).

- **Testes de sistema (end-to-end):** definição e importância. Exemplos de casos de teste end-to-end para nosso MVP: *"Usuário se registra, faz login, visualiza eventos e realiza inscrição; sistema responde confirmando"*. Conceitos de teste manual vs. automatizado (introduzir ferramentas como Selenium/Cypress para front ou Postman collections).
- **Detecção e tratamento de erros em integração:** leitura de logs de múltiplos serviços para rastrear uma falha (ex.: se inscrição não aparece, verificar logs do serviço de Inscrição e do front).
- **Feedback do usuário final:** garantia de que o MVP atende ao objetivo básico. Discutir possivelmente o *mínimo necessário* para um sistema ser utilizável – priorização de funcionalidades.

- **Planejamento da próxima iteração:** como usar o feedback do MVP para guiar melhorias (introdução ao conceito de retrospectiva ágil).

- **Práticas propostas:**

- **Teste completo do MVP:** Colocar toda a aplicação para rodar:
    - Backend: subir todos os microserviços (Auth, Eventos, Inscrições, Gateway; Pagamentos/Notificações podem estar ausentes ou stubs por enquanto) – usar Docker Compose ou executar via IDE.
    - Frontend: rodar a aplicação React (em dev server ou deploy simples) e configurá-la para apontar ao gateway para API (definir base URL).
    - Executar *passo a passo* o cenário principal:
    - Cadastro de um novo usuário (via tela de SignUp ou Postman) – verificar usuário criado no banco.
    - Login com esse usuário – front recebe JWT, armazena (ex.: localStorage).
    - Tela de listagem de eventos – consumir GET /eventos do gateway – verificar se lista aparece (precisar ter eventos no DB; se não, criar alguns manualmente ou via chamada API).
    - Tela de detalhe/inscrição – ao acionar "Inscrever-se", realizar POST /inscricoes – verificar resposta e feedback na interface (ex.: mostrar mensagem de sucesso).
    - Opcional: Ver no banco de dados que a inscrição foi gravada; ou se implementado, ver log de mensagem enviada para notificação.
    - Testar variações: inscrição duplicada (deve retornar erro), chamada sem token (front deve redirecionar para login ou API retornar 401).
    - Ajustar configurações de CORS se necessário para o React conseguir chamar (possivelmente habilitar no Gateway ou nos serviços).
    - Monitorar logs de todos serviços durante teste para identificar quaisquer exceptions ou comportamentos inesperados.
- **Correção de bugs de integração:** É comum que nesta fase apareçam problemas (ex.: formato de data diferente, um campo faltando na resposta da API que o front esperava). Os alunos devem colaborar para localizar o problema e corrigi-lo:
    - Se front quebrando, usar DevTools do navegador para inspecionar requests/respostas.
    - Se um serviço retorna erro, ler stacktrace no console do serviço para entender causa (ex.: NullPointer, erro SQL, etc.).
    - Priorizar correções que impeçam o fluxo principal.
    - Documentar temporariamente problemas não críticos que serão resolvidos depois (não perder de vista).
- **Teste de performance básico (sanity):** opcionalmente, simular 5-10 inscrições quase simultâneas (via script ou pedindo colegas para tentar ao mesmo tempo) para observar se o sistema comporta (talvez útil se problemas de concorrência).

- **Demonstração do MVP:** cada equipe apresenta para o professor (e possivelmente colegas) o sistema em funcionamento end-to-end, mesmo que simples, comprovando que já é possível realizar o objetivo principal (inscrição em evento) [5] [6]. A demonstração pode ser informal, mas deve cobrir as funcionalidades implementadas. O professor toma notas de pontos de melhoria.
- **Aplicação do CERTO na depuração:** se encontrarem um bug persistente, os alunos podem usar ChatGPT com a estrutura CERTO: *Contexto:* explicar a arquitetura e onde ocorre o erro; *Exigências:* precisam resolver sem quebrar outra funcionalidade; *Referências:* incluir mensagem de erro/log; *Tarefa:* pedir identificação da causa raiz e sugestão de correção; *Observações:* mencionar o que já tentaram [51]. Isso pode agilizar a resolução de problemas obscuros.

- **Checkpoint de retrospectiva:** promover uma breve sessão de retrospectiva dentro de cada equipe (ou aberta na turma) focada na Iteração 1:

    - O que funcionou bem? (ex.: divisão de tarefas front/back em paralelo, uso do método CERTO para resolver dúvida X, etc.)
    - Que dificuldades enfrentaram? (ex.: integrar JWT no front, configurar Docker, etc.)
    - O que poderia ser melhorado na próxima iteração? (ex.: escrever mais testes antecipadamente, melhorar comunicação entre subequipes, etc.)
    - Registrar 2-3 ações de melhoria para Iteração 2.
    - **Feedback do Professor:** O professor fornece retorno formal sobre o MVP: se o escopo mínimo foi atingido, se há problemas de arquitetura ou qualidade que precisam ser corrigidos, e sugestões de prioridades para a próxima fase. Esse feedback deve ser incorporado no planejamento da Iteração 2.

- **Atividade avaliativa/entregável sugerido: Demonstração do MVP Integrado** – os grupos devem entregar ou apresentar evidências do sistema funcionando como um todo, cobrindo autenticação, fluxo de evento e inscrição [6] [52]. Podem ser vídeos curtos da tela ou a própria apresentação ao vivo. Adicionalmente, um **relatório sucinto de status** da Iteração 1: listando funcionalidades implementadas, testes realizados, problemas encontrados e soluções/adaptações feitas (incluindo menção se usaram ChatGPT e como). *(Avaliação:* atendimento do critério de pronto do MVP – se permite cumprir o caso de uso principal; qualidade da integração – ex.: ausência de erros não tratados; e reflexão crítica no relatório – se identificaram bem pontos falhos e acertos.)

Semana 10: Iteração 2 – Novas Funcionalidades (Pagamentos e Notificações)

- **Objetivo pedagógico:** Expandir o sistema com funcionalidades complementares de maior complexidade e começar a abordar requisitos não-funcionais importantes. Especificamente, nesta semana os alunos integrarão um serviço de Pagamentos (simulado ou sandbox) e o serviço de Notificações (envio de e-mails), aprimorando o fluxo de inscrição. Aprenderão sobre integração com APIs externas (pagamento) e serviços de infraestrutura (SMTP para e-mail, filas) e reforçarão o conceito de *event-driven architecture*. Ao final da semana, o sistema suportará inscrições pagas e envio de notificações de confirmação.

- **Conteúdos teóricos abordados:**

- **Integração com serviços externos (pagamentos):** discutir como funcionam gateways de pagamento (ex.: PayPal, Stripe) – normalmente via APIs REST ou SDKs. Conceitos de transação financeira simulada, segurança (não guardar dados de cartão em texto plano), ambiente sandbox vs produção.

- **Serviço de Pagamentos no projeto:** delinear como será o fluxo: ao inscrever-se num evento pago, chamar o microserviço de Pagamentos que simula ou delega a cobrança do cartão. Decidir se será sincrono (espera aprovação imediata) ou assincrono (inscrição pendente até confirmação).
- **Envio de e-mails e notificações:** protocolos (SMTP básico) ou APIs de e-mail (SendGrid, etc.). No nosso caso, podemos usar um servidor SMTP simples (MailHog para dev) ou simular envio logando a saída. Estrutura de um e-mail de confirmação (assunto, corpo com detalhes do evento).
- **Mensageria aprofundamento:** rever RabbitMQ em prática:
    - Criar exchange/topic para eventos de domínio (ex.: "eventos.inscricao" para novas inscrições).
    - **Consumer**: implementar um consumidor no serviço de Notificações que fique escutando a fila de inscrições e, ao receber, envia email.
    - Idempotência e reenvio: mencionar superficialmente que idealmente o consumo deve ser robusto a falhas (ex.: se envio de email falha, mensagem reentra na fila).
    - Notar como esse padrão melhora desacoplamento: Inscrições não precisa conhecer Notificações.
- **Poliglotismo nos serviços:** abrir discussão para se algum novo serviço poderia ser em linguagem diferente. Por exemplo, propor que **Notificações** seja implementado em Python (FastAPI) em vez de Java, ilustrando a vantagem de microserviços em permitir escolha da melhor ferramenta para cada propósito. (Python tem facilidade com SMTP/email).
    - Também porque menor overhead para um serviço relativamente simples.
- **Considerações de infraestrutura:** envio de e-mail real requer chave SMTP ou API key – para fins didáticos, usaremos uma conta teste ou um catcher (ou só log). Mas explicar cuidados (se fosse real, usar cofres para credenciais).
- **Logging distribuído básico:** agora que sistema se torna mais distribuído com eventos, destacar a importância de correlacionar logs. Ex.: incluir em logs de Inscrições um ID da inscrição, e no log de Notificações usar o mesmo ID para rastrear. (Introduzir conceito de *trace ID*, mas não implementar se complexo).
- **Segurança adicional:** pagamento sendo crítico, assegurar endpoints de Pagamento só acessíveis via serviço de Inscrição (pode restringir por token ou secret interno). Notificações possivelmente nem expõe endpoint público (só responde a eventos).

- **Testes de integração multi-serviço:** ao introduzir mais serviços, a complexidade de teste cresce. Mencionar que podem usar testes integrados compondo vários serviços (ex.: simular inscrição completa incluindo pagamento e ver se notificação gerou email). Deixar implementação para semana de testes.

- **Práticas propostas:**

- **Implementar Serviço de Pagamentos:**
    - Escolher linguagem (pode ser Spring Boot novamente ou aproveitar para mostrar FastAPI if time; mas Java ok).
    - Este serviço terá possivelmente uma integração externa fake. Simplificar:
    - Criar um endpoint `POST /pagamentos` que recebe algo como {inscricaoId, dadosCartao ou tokenPagamento}. Para segurança, não manusearemos cartão real – pode simular aprovação automática.
    - Ao receber requisição, o serviço Pagamentos cria um registro de pagamento (opcional, se quisermos guardar histórico – Payment entity com id, inscricaoId, status, valor).
    - Em seguida, responde sucesso (200) se simulado ok, ou 402 Payment Required se simular falha, etc.

    - Poderia chamar API de sandbox externa: *Se* houver tempo/recursos, integrar com PayPal sandbox via SDK ou API simplesmente para demonstrar, mas isso pode ser muito detalhado. Em vez disso, talvez gerar aleatoriamente success/fail para demonstrar lógica de erro.
    - **Integração Inscrição-Pagamento:** ajustar o serviço de Inscrições:
    - Se o evento inscrito tiver flag pago:
        - Opção 1: Inscrições chama imediatamente o serviço Pagamento (síncrono). Se pagamento ok, marca inscrição como CONFIRMADA; se falhar, marca inscrição CANCELADA/ não cria inscrição.
        - Opção 2: Inscrições cria registro com status PENDENTE e retorna ao front pedindo redirecionamento para processo de pagamento. Front então chamaria Pagamentos (via gateway ou diretamente).
        - Decisão para simplificar: utilizar *Opção 1 sincrona*: no backend, ao receber inscrição, ele próprio chama Pagamentos internamente (como internal API call, similar a eventos), e só finaliza inscrição se pagamento retornar ok. Isso simplifica fluxo para usuário (um clique faz tudo), embora não reflita alguns flows reais mas é suficiente.
    - Implementar essa chamada interna: Inscrições -> Pagamentos (REST call, similar a Inscrições->Eventos).
    - Tratar a resposta: se falha, talvez retornar erro ao front (“pagamento recusado”) e não gravar inscrição; se sucesso, salvar inscrição com status confirmada.
    - Logar resultado.
    - Proteger serviço de Pagamentos: por segurança, somente serviços internos (Inscrições) deveriam chamá-lo, mas não implementaremos auth complexa – confiar no isolamento de rede (docker-compose) ou simples validação de origem se quisesse.
- **Implementar Serviço de Notificações:**
    - Linguagem: **Python FastAPI** (por exemplo) para variedade. Configurar minimal.
    - Este serviço pode ter dois componentes:
    - Um *consumer* rodando em segundo plano ouvindo RabbitMQ.
    - Opcionalmente, um pequeno API para testes/admin (não estritamente necessário).
    - Instalar lib de RabbitMQ (pika) e de e-mail (smtplib ou use FastAPI background tasks).
    - **Consumidor de mensagens:** conectar no RabbitMQ, declarar fila `inscricoes` (ou via exchange) – depende de configuração. Aguardar mensagens.
    - Ao receber mensagem de nova inscrição: extrair email do usuário (a mensagem deve conter ou permitir pegar via userId -> poderia então chamar serviço Usuários pra pegar email se só veio userId).
    - Montar conteúdo do e-mail (ex.: "Olá {nome}, sua inscrição no evento X em {data} foi confirmada.").
    - Enviar email: se temos SMTP disponível, usar. Caso não: simular enviando (ex.: imprimir no log "Email para fulano: ...").
    - Se envio falhar, realizar tentativa (pode logar erro).
    - **Integração para envio real (opcional):** configurar um servidor SMTP dummy (como MailHog container) para capturar emails e visualizar. Ou usar conta Gmail de teste (menos seguro). Se configurado, armazenar credencial externamente.
    - Testar o consumidor manualmente: publicar uma mensagem de exemplo via RabbitMQ admin ou code e ver se serviço pega.
    - Integrar com Inscrições: já provavelmente implementamos Inscrições publicando mensagem. Se não, agora fazê-lo: após salvar inscrição (e pagamento ok se houver), publicar msg com info: e.g., {"userId":..., "eventId":..., maybe "email": "...", "eventName":"..."}.
    - Garantir o serviço Notificações está rodando em paralelo (via docker-compose).

    - **Atualizar docker-compose:** adicionar serviço de pagamentos e notificações, além de RabbitMQ.
    - **Ajustar front-end:** agora após inscrição (se pagamento envolvido), front pode precisar coletar dados de pagamento:
    - Se implementarmos sincrono interno, do ponto de vista do front, nada muda muito – ele clica “inscrever”, e se for pago, talvez deveria aparecer um formulário de cartão.
    - Simplificar: podemos decidir que todos eventos até agora eram gratuitos; para mostrar pagamento, marcamos um evento como pago a partir de agora:
        - Na tela de detalhes do evento, se event.pago = true, mostrar campos de cartão (fake) e incluir esses dados na chamada de inscrição.
        - Ou mais simples: colocar botão "Inscrever e Pagar" que apenas chama inscrição – internamente Inscrições lida. Talvez só para simular, podemos enviar junto um dummy card info.
    - Implementar feedback: se inscrição+pagemento for ok, mostrar "Inscrição confirmada, email enviado"; se falha, mostrar "Pagamento recusado".
    - Isso pode ser refinado, mas o essencial é que o sistema agora trate ambos os casos.
    - **Considerar casos de erro para testes:** e.g., simular pagamento falhou (talvez se valor > X, recusar, ou random fail) – verificar se inscrição não foi salva, e front recebe erro.
- **Uso de ChatGPT para novas tecnologias:**
    - Perguntar exemplos de envio de e-mail em Python, ou configuração de RabbitMQ consumer.
    - Por ex.: *“Contexto:* microserviço em Python precisa enviar emails; *Exigências:* usar smtplib com conta Gmail; *Referências:* forneça talvez credenciais fictícias; *Tarefa:* exemplo de código para enviar email com assunto e corpo; *Observações:* —.”*
    - Ou *“Como implementar um consumer RabbitMQ em Python que consome fila X e processa mensagens JSON”.*

- **Testes da nova funcionalidade:**

    - Escolher um evento no sistema e marcá-lo como pago (no DB ou via API).
    - Tentar inscrição nesse evento pelo front:
    - Ver comportamento: se pagamento simulado corretamente e email logado.
    - Verificar logs:
    - Inscrições deve mostrar chamada ao serviço Pagamento e resultado.
    - Pagamentos deve mostrar recebimento de pedido.
    - Notificações deve logar envio email (ou ver email no MailHog).
    - Tentar casos: pagamento falha (ex.: usar número de cartão "fail" se codificamos isso), observar se front avisa e nenhum email enviado.
    - Testar eventos grátis ainda funcionam (não chama pagamento).
    - Multi-thread: inscrever 2 usuários em eventos diferentes quase simultâneo – ver se ambos recebem email.

- **Atividade avaliativa/entregável sugerido:** Atualização do sistema com **Novas Funcionalidades de Pagamento e Notificação**:

- **Demonstração:** mostrar em aula o fluxo completo para um evento pago: usuário se inscreve, pagamento é processado (simulado) e email de confirmação é enviado (mostrar evidência, ex.: tela do MailHog ou log) [53] [54] .
- O entregável pode ser um **mini-relatório** explicando a integração realizada: como funciona o pagamento simulado, como as notificações estão implementadas, incluindo configurações (ex.: “usamos MailHog na porta tal, credencial X”).

- *(Avaliação:* funcionalidade das novas features – inscrição paga resultando em atualização correta e notificação; correta utilização de mensageria para desacoplar (Inscrições não chamando Notificações diretamente); robustez básica – ex.: sistema lida com falha de pagamento; e aspectos de código – solução criativa para simular pagamento, uso adequado de libs e event-driven.)

Semana 11: Iteração 2 – Aprimoramentos de Robustez e Serviço de Feedback

- **Objetivo pedagógico:** Completar o escopo funcional previsto (implementando a coleta de feedback dos eventos) e direcionar esforços para melhorar a qualidade sistêmica: desempenho, confiabilidade e segurança. Os alunos aprenderão sobre otimizações como caching, tratamento abrangente de erros e reforço de segurança. Ao final da semana, todos os requisitos funcionais iniciais estarão implementados e o sistema estará mais resiliente, preparando-o para os testes finais.

- **Conteúdos teóricos abordados:**

- **Serviço de Feedback:** propósito (coletar avaliações dos participantes após eventos). Diferenças de requisitos: possivelmente grande volume de dados de texto (feedbacks), menor necessidade de transações fortes, ideal para NoSQL (MongoDB).
    - Vantagem de usar **MongoDB** aqui: esquemas flexíveis para respostas, facilidade de agregar dados (ex.: médias de nota).
    - Breve introdução ao MongoDB e comparativo com SQL: documentos JSON, coleções, não há joins diretos, uso de _id etc.
    - Biblioteca: usar Spring Data Mongo ou PyMongo, dependendo da implementação (Java ou Python).
- **Cache e Performance:** introduzir **Redis** como ferramenta de caching in-memory. Discussar cenários de uso:
    - Cache de leitura pesada: por ex., lista de eventos pode ser cacheada, em vez de consultar DB a cada vez (especialmente se houver muitos eventos ou acessos).
    - Implementação: Spring Cache Abstraction ou manualmente interagir com Redis. TTL (expiração de cache).
    - Também citar uso de Redis para armazenar sessões se JWT não fosse stateless, ou cache de token blacklist se implementássemos logout.
    - Para nosso projeto, escolher um endpoint para demonstrar caching – ex.: GET /eventos.
- **Melhoria de Resiliência:** introduzir o padrão **Circuit Breaker** e **Retry**:
    - Explicar conceito: se um serviço dependente estiver fora, evitar chamadas repetidas que falham – Circuit Breaker abre e impede chamadas temporariamente.
    - Ferramentas: libs como Resilience4j, Istio (em K8s).
    - Talvez não implementar plenamente, mas conscientizar. Um mini-implementação: ex.: no serviço Inscrições ao chamar Pagamentos, implementar uma lógica de 3 tentativas antes de desistir; ou usar Resilience4j annotations (if library can be added easily).
    - **Timeouts:** assegurar que chamadas HTTP internas tenham timeout configurado para não ficar travados indefinidamente.
    - **Fallbacks:** ex.: se serviço de Notificação indisponível, Inscrições ainda conclui e apenas loga “notificação pendente”.
- **Segurança avançada:**
    - **Validação de inputs:** lembrar de validar do lado servidor todos os dados recebidos (mesmo já validados no front). Ex.: tamanho de strings, formatos (email válido), evitar SQL injection (usando queries parametrizadas ou ORM já está ok), prevenir script injection em campos de texto (escapar outputs em front).

    - **Sanitização:** se algum serviço injeta dados do usuário em HTML/email, sanitizar para evitar XSS.
    - **Criptografia:** garantir que senhas já estão seguras (hash, sal); se houver dados sensíveis (talvez não no nosso escopo), discutir criptografia de dados em repouso.
    - **Autorização fina:** revalidar que cada endpoint protege o que deve (ex.: usuário comum não pode deletar eventos, etc.). Implementar checks de autorização nas funções (no Spring, usar @PreAuthorize roles; no Python, manual).
    - **Logs seguros:** não registrar informações sensíveis (ex.: não logar conteúdo de senhas ou tokens).
- **Monitoramento básico:** (Teórico leve) – introduzir conceitos de métricas (requests per second, CPU, memory), uso de Prometheus e Grafana. Talvez não implementar, mas mencionar que poderiam instrumentar endpoints para coletar métricas (ex.: Spring Actuator metrics).
- **Refatoração e clean-up:** enfatizar limpar *code smells*: remover código morto, adicionar comentários onde necessário, padronizar nomenclaturas e formatações, etc., para entregar produto polido.

- **Preparação para testes finais:** planejar que com todas features implementadas, próximo passo é testar a fundo – isso justifica arrumar tudo agora para que testes não fiquem bloqueados por bugs triviais.

- **Práticas propostas:**

- **Implementar Serviço de Feedback:**
    - Escolher implementação – provavelmente Java Spring Boot com Spring Data Mongo (já que alunos estão com Spring fresco) ou Python with FastAPI + PyMongo.
    - **Configurar MongoDB**: adicionar container Mongo no docker-compose.
    - Criar modelo **Feedback**: campos como id, eventId, userId, nota (rating de 1-5), comentários (texto), data/hora.
    - **Endpoint POST /feedback:** um participante envia feedback de um evento:
    - Requer autenticação (usuário logado).
    - Validar que o user participou do evento (pode consultar inscrições para ver se userId+eventId existe, ou incluir no JWT a lista de eventos? Mais simples: consultar Inscrições microserviço – poderia ser interna, ou skip se confia).
    - Validar conteúdo: nota dentro de 1-5, comentário tamanho limite.
    - Salvar no Mongo (coleção feedbacks) o documento.
    - Retornar 201 Created.
    - **Endpoint GET /feedback?evento=ID:** retornar lista de feedbacks (ou estatísticas) de um evento – provavelmente para organizador ver avaliações:
    - Proteger para admin (ou talvez disponíveis publicamente se quiserem mostrar rating médio).
    - Consulta Mongo por eventId, retornar lista de feedbacks ou média.
    - Se for público, poderia retornar só média e contagem em vez de todos comentários. Mas a critério do escopo – definam se feedback é privado ou público.
    - Testar feedback:
    - Simular envio de feedback via Postman para um evento e ver no Mongo (use Mongo client or simple GET to verify).
    - Integrar minimal no front: talvez uma seção "avaliar evento" aparece após inscrito? Se sem UI, não tem problema, pode testar via API only e apresentar via console.
    - Polir: se tempo, implementar que usuário só pode enviar feedback se evento já ocorreu (ex.: comparar data evento <= hoje).

- **Implementar Caching (Redis):**
    - Adicionar container Redis ao docker-compose.
    - Escolher um lugar para aplicar:
    - Por exemplo, no serviço Eventos: cachear resultado de GET /eventos (lista de eventos), para evitar acesso repetido ao DB quando a lista não muda com frequência.
    - Configurar Spring Cache (enable caching + use @Cacheable on getAllEvents method, which by default uses a ConcurrentMap or we can config Redis as cache manager).
    - Ou manual: for Python, perhaps skip due to overhead; likely do in Spring if using Spring Boot.
    - Testar: chamar /eventos duas vezes, ver no log que segunda vez não acessou DB (if logging cache hits).
    - Ensinar invalidation: se um novo evento é adicionado via POST, invalidar cache (Spring Cache can evict on save).
    - Alternativamente, cache outros heavy operations if exist (maybe not many heavy ones in our small project).
- **Hardening inter-serviço:**
    - Configurar timeouts e retries:
    - Ex.: In Inscrições calling Pagamentos: set a short timeout (e.g., 3s). If fail, maybe try again once.
    - Could integrate Resilience4j: add dependency and annotate method with @Retry (attempts=3) and @CircuitBreaker (to open if continuous fails).
    - Or implement try-catch with loop manually.
    - Simulate scenario: stop Payment service and see Inscrições handling:
    - With retry: it will try and fail 3 times quickly, then respond with error message like "Pagamento indisponível, tente mais tarde".
    - With circuit breaker (if implemented): subsequent attempts might immediately fail if open, which is fine.
    - Document how to reset (like after X seconds).
    - Implement fallback for Notificações:
    - If Notificações down, our current design with RabbitMQ means message will queue until service up – which is inherently a nice fallback (durable queue).
    - Check that we set queue durable and messages persistent if needed.
    - Maybe simulate by stopping Notificações, doing an inscrição, then starting Notificações to see it consume backlog.
    - Log observation.
    - **Input validations:**
    - Sweep through all controllers and add validation annotations or code:
        - Spring: use javax.validation (@NotBlank, @Size, etc.) on DTOs, and global exception handler to return 400 if validation fails.
        - Python: Pydantic in FastAPI automatically does some validation if using BaseModel.
    - Test couple: try to create event with missing name (expect 400 with message).
    - Ensure consistent error responses (maybe define an error JSON format).
    - **Authorization checks:**
    - Confirm all admin-only endpoints check role (if not done, add checks).
    - For example, ensure only admin can create event or view all feedback: in Spring, possibly use @PreAuthorize or manual check using JWT claims.
    - Test by calling as normal user and expecting 403.
    - **Polish security:** ensure password hashing is in place (if not, implement now).
    - If any token secrets or SMTP credentials, ensure they are not hardcoded – maybe use env vars via docker-compose (teaching how to supply secrets).

    - Perhaps implement **logout** endpoint: not strictly needed with JWT (stateless), but could implement token blacklist if desired. Possibly skip due to complexity.
- **Code refactoring & docs update:**
    - Clean up any TODOs in code, add comments explaining complex sections (future maintainers).
    - Update API documentation (Swagger) to include new endpoints (Pagamentos, Notificações maybe none, Feedback).
    - Update architecture docs if any changes (ex.: new services).
    - Check that README or developer guide notes new services and how to run RabbitMQ, Redis, Mongo etc.
    - Consider creating **Architecture Decision Records (ADR)** for key changes (just mention if practiced).
- **Test run:**
    - Execute a *full system test* now with all features:
    - Use-case: Organizador cria evento pago, vários usuários se inscrevem, sistema processa pagamento e envia emails, um usuário envia feedback depois, organizador visualiza feedback.
    - Try to observe logs and DBs to ensure each part working.
    - Performance test small: maybe measure how quickly can it handle, but not critical.

- **Uso de IA durante otimizações:**

    - Consultar ChatGPT se encontrarem gargalo: *"Como otimizar consulta X do MongoDB?"* ou *"Melhores práticas de configurar Resilience4j no Spring Boot"*.
    - Validate security: *"Checklist de segurança para API REST"* (como sanity check if nothing missed).
    - Possibly use ChatGPT to do a mini code review: provide a snippet and ask for improvement suggestions (Observações: snippet; Tarefa: "sugerir melhorias de robustez ou legibilidade").

- **Atividade avaliativa/entregável sugerido: Sistema Completo Implementado com Melhorias**:

- Código do **Serviço de Feedback** funcionando e integrado (com MongoDB).
- **Demonstração** das melhorias: por exemplo, mostrar que listar eventos está mais rápido após cache (difícil visualmente, mas talvez logs), ou simular indisponibilidade de um serviço e mostrar sistema degradando graciosamente (ex.: desligar Pagamentos e demonstrar mensagem de erro amigável ao tentar pagar, em vez de travar).
- **Relatório breve de progresso (Iteração 2):** indicar todas as funcionalidades adicionadas nesta iteração e as melhorias técnicas feitas, incluindo quaisquer alterações arquiteturais (ex.: "Introduzido Redis para cache, implementação de feedback com MongoDB, ampliado diagrama de arquitetura para incluir esses componentes"). 9 55
- *(Avaliação:* cumprimento do escopo completo – todos os módulos funcionais implementados; qualidade técnica – presença de mecanismos de cache, validações robustas, segurança reforçada; documentação e código coerentes – projeto pronto para fase de testes finais.)

Semana 12: Iteração 2 – Consolidação Final e Deploy em Ambiente de Homologação (Staging)

- **Objetivo pedagógico:** Finalizar a fase de desenvolvimento colocando o sistema completo em um ambiente controlado para testes integrados (staging) e garantindo que está pronto para uso/presentação. Os alunos consolidarão toda a aplicação, unificando versões, preparando

contêineres e implantação. Ao final da semana, o sistema estará rodando de forma reproduzível em um servidor ou ambiente isolado, marcando o *feature freeze* (congelamento de funcionalidades) e abrindo caminho para a fase de testes e avaliação.

- **Conteúdos teóricos abordados:**

- **Ambientes de Deploy:** diferenças entre desenvolvimento, homologação (staging) e produção. Importância de testar em ambiente o mais próximo possível do real antes da entrega final.
- **Configurações por ambiente:** uso de perfis (ex.: Spring Profiles, arquivos .env diferentes) para ajustar parâmetros (ex.: credenciais de banco) entre dev e prod. Gestão de variáveis de ambiente no Docker Compose ou Kubernetes secrets.
- **Container Registry e Imagens Docker:** possivelmente como publicar as imagens Docker em um repositório (Docker Hub, GitHub Container Registry) para facilitar deploy fora do ambiente local.
- **Kubernetes (introdução prática):** se houver recursos, demonstrar rapidamente como seria implantar um dos serviços no Kubernetes:
    - Conceitos: Pod, Deployment, Service, Ingress.
    - Mostrar um YAML de Deployment do serviço Eventos e um Service + Ingress, só como exemplo.
    - (Não necessariamente executar, apenas para vislumbrar o profissional).
- **Continuous Deployment (CD):** como seria possível automatizar o deploy no staging a cada push no main branch – ex.: GitHub Actions pipeline que constrói imagens e atualiza container. Talvez mencionar se pipeline foi parcialmente implementado para Docker builds.
- **Versão final do produto:** adotar *versionamento semântico* (v1.0.0 para a entrega final). Taggear repositório. Congelar adição de novas features – apenas correções de bugs daí em diante.

- **Checklist de Pre-Release:** listar tudo que deve ser verificado antes de considerar "pronto para teste/entrega": todos endpoints funcionando, todos serviços no compose up, testes unitários passando, documentação atualizada, nenhuma credencial sensível vazando, etc.

- **Práticas propostas:**

- **Configurar ambiente de *staging*:**
    - Ideal: provisionar uma VM ou servidor (pode ser na infra da faculdade ou cloud gratuita) para rodar o sistema. Alternativa: usar a própria máquina do grupo mas simular ambiente limpo (subir em container).
    - Transferir os arquivos necessários: Docker Compose + .env (com senhas seguras), imagens Docker (ou construir lá).
    - Subir o docker-compose no servidor e resolver quaisquer ajustes (ex.: abrir portas adequadas, configurar domain/localhost, etc.).
    - Se não for possível servidor externo, considerar **staging local**: rodar tudo em Docker na própria máquina mas usando imagens e rede simulada para emular proximidade de real.
    - Verificar conectividade: front-end talvez implantado como container Nginx servindo build estático React, ou simplesmente rodar front local apontando para IP do compose.
    - **Testar no staging:** repetir testes end-to-end básicos neste ambiente para garantir nada difere (ex.: as URLs de callback, etc.).
    - Monitorar performance básica: usar comandos docker stats ou outros para ver uso de CPU/RAM com tudo rodando.
    - Ajustar configs se serviço consome muito recurso.

- **Finalizar *Docker Compose* de produção:**
    - Garantir que no compose de produção estejam todos serviços, cada um com restart policy (always) para robustez, e talvez escala de instâncias (pode deixar 2 replicas de algum serviço, mas não trivial sem K8s – provavelmente skip).
    - Colocar volumes se necessário (ex.: se quiséssemos persistir dados do DB – mas para simplicidade, DBs podem persistir em volume local).
    - Documentar no README como rodar este compose.
- **(Opcional) Deploy no Kubernetes (demonstração):**
    - Se a faculdade tiver minikube ou similar, tentar empacotar 1 ou 2 serviços em K8s:
    - Escrever manifest YAML de Deployment do gateway + service NodePort, etc.
    - Não precisar fazer todos, só para ilustrar.
    - Caso sem K8s real, explicar como seria (teórico).
- **Continuous Deployment Simulation:**
    - Atualizar pipeline CI (GitHub Actions) para construir imagens Docker e enviar a um registry (opcional, se tempo).
    - Ou script manual: docker build de todos e docker push para hub sob tags.
    - Documentar imagem e tag de cada serviço.
    - Assim, entregar junto imagens para facilitar correção do professor, por ex.
- **Taggear versão 1.0:** nos repositórios, criar tag `v1.0.0` e liberar *release* no Git (se for GitHub, gerar Release).
- **Lockdown de funcionalidades:** comunicar equipe que agora apenas bugfixes entram – introduzir noções de *change freeze* antes de release para evitar instabilidade tardia.
- **Smoke Test final:** com o sistema rodando em staging, executar um último ciclo de testes superficiais de tudo:
    - Login, inscrição, pagamento, email, feedback – tudo uma vez.
    - Acompanhar logs – se algo crítico aparece, corrigir agora.
    - Se nenhum showstopper, declarar MVP final pronto para testes rigorosos semana seguinte.
- **Aplicação do CERTO no deploy:** se encontrarem dificuldades de deploy:
    - Ex.: *"Contexto:* tentando deployar microserviços no AWS EC2 usando Docker Compose; *Exigências:* precisa rodar em background e reiniciar em crash; *Referências:* descrevemos brevemente erro ou comportamento (ex.: container X não sobe, porta Y inacessível); *Tarefa:* pedir sugestões de configuração ou troubleshooting; *Observações:* —."*
    - ChatGPT pode auxiliar indicando verificar firewall, usar docker-compose detach, etc.
- **Atualização da documentação técnica:** se algo mudou em config (ex.: URLs, porta do front, credenciais), refletir nos documentos (Dev Guide).
    - Incluir instruções claras de deploy para o avaliador (ex.: "executar `docker-compose up -d` e acessar http://localhost:3000 para front, etc.").
    - Capturar *screenshots* do sistema rodando (UI) para incluir no manual do usuário.

- **Preparar mente para testes:** concluir com equipe revisando critérios de aceitação iniciais listados no doc de visão e marcando todos como "Atendido? Sim/Não" – se algum "Não", discutir se foi decidido escopo out ou se é bug. Isso vira insumo para testes semana 13.

- **Atividade avaliativa/entregável sugerido: Sistema implantado em Staging + Relato de Implantação**:

- Entregar evidências de que o sistema está rodando integradamente em ambiente de homologação. Pode ser um IP/URL de acesso para o professor (se disponível), ou um relatório com prints do Docker Compose up e da aplicação acessível naquele ambiente.

- **Relatório de implantação:** documentando passo-a-passo como subir o sistema (para que avaliadores possam reproduzir facilmente) e quaisquer diferenças de configuração entre dev e prod (ex.: credenciais dummy usadas).
- *(Avaliação:* completude técnica – todos componentes configurados corretamente no ambiente, capacidade de reproduzir a execução; qualidade do deploy – uso de containers, scripts ou pipeline, etc., evidenciando preparação profissional; e entrega no prazo do sistema funcional para testes, marcando cumprimento do cronograma [56] [12].)

Semana 13: Testes Finais, Verificação da Qualidade e Ajustes

- **Objetivo pedagógico:** Validar exaustivamente o sistema contra os requisitos definidos, garantindo qualidade e estabilidade. Os alunos realizarão testes abrangentes (funcionais, usabilidade, desempenho básico, tolerância a falhas) e produzirão um relatório de testes. Também farão os ajustes finais identificados (bugfixes). Ao final da semana, espera-se ter um sistema estável, testado e pronto para a entrega final, com evidências formais de qualidade.

- **Conteúdos teóricos abordados:**

- **Estratégia de Testes:** recapitular pirâmide de testes (unidade, integração, E2E). Aqui foco nos **Testes de Sistema**:
    - **Testes Funcionais**: validar cada requisito funcional (cada funcionalidade implementada) – criar casos de teste para cada user story. Cobrir também cenários de exceção (erros).
    - **Testes de Usabilidade**: avaliar se a interface é intuitiva (feedback qualitativo; possivelmente pedir a alguém de fora para usar e relatar dificuldade).
    - **Testes de Desempenho**: planejar pequenos testes de carga – ex.: usando JMeter ou k6, simular 50 requisições simultâneas de listagem de eventos ou inscrições e medir tempo médio de resposta. Ou avaliar uso de recursos com N usuários.
    - **Testes de Segurança**: verificar se sistema resiste a inputs maliciosos (XSS, SQLi – provavelmente mitigado por ORM, mas tentar), verificar se autenticação não pode ser burlada (ex.: chamar inscrição sem token – deve dar 401), tokens expirados não funcionam, etc.
    - **Testes de Resiliência**: simular falhas de serviços (derrubar container X e ver se o restante continua funcional graciosamente).
    - **Teste de compatibilidade**: se front web, testar em outros browsers ou dispositivos (básico).
- **Ferramentas de teste:** apresentação de alguns:
    - **Postman/Newman**: para roteirizar chamadas API (pode criar uma coleção de requests para todo o fluxo e rodar automaticamente, verificar respostas).
    - **Cypress or Selenium**: para testes automatizados de front-end (talvez escrever 1 ou 2 cenários simples, se houver conhecimento).
    - **JMeter/k6**: para carga – demonstrar script ou GUI JMeter fazendo 100 requests.
- **Métricas de teste:**
    - Cobertura de testes unitários (%) – gerar relatório (ex.: Jacoco no Java) [11].
    - Métricas de performance: tempo médio login, etc., comparado com requisitos (ex.: se havia NFR “responder em < 2s”).
    - Resultado de testes de carga: throughput alcançado, uso CPU.
    - Tabela de casos de teste vs resultado (Pass/Fail).
- **Critérios de aceitação finais:** revisar cada requisito do Documento de Visão (semana 1-2) e marcar se foi cumprido (traçabilidade). Itens não atendidos devem ser justificados ou considerados *débitos/out-of-scope*.

- **Bug triage e gerenciamento:** se encontrar muitos bugs, priorizar por severidade (crítico – impede função principal; médio – contornável; cosmético).
- **Importância de não "corrigir durante teste" descontroladamente:** manter disciplina – reproduzir bug, corrigir, rerodar testes relacionados. Documentar alterações.

- **Go/No-Go:** conceito de definir se produto está apto para release após testes ou se precisa de mais trabalho (em ambiente profissional, aqui será apto se nenhum bug crítico aberto).

- **Práticas propostas:**

- **Plano de Testes:** antes de sair testando, cada equipe elabora uma lista de casos de teste para cobrir:
    - Para cada funcionalidade (CRUD de eventos, inscrição, pagamento, notificação, feedback, login, etc.), definir cenários feliz e de erro.
    - Exemplo: "Realizar inscrição - caso feliz" (pré-condições: usuário logado, evento existe e vagas >0; passos; resultado esperado: inscrição confirmada, e-mail enviado).
    - "Realizar inscrição - evento lotado" (esperado: mensagem de erro "vagas esgotadas").
    - "Login - senha incorreta" (esperado: erro 401).
    - "Acessar endpoint admin como user normal" (esperado: 403).
    - "Enviar feedback sem ter participado" (esperado: erro).
    - Organizar casos de teste em uma matriz ou planilha para marcar status.
- **Execução de Testes Funcionais/Manuais:**
    - Realizar cada caso conforme planejado. Registrar resultado (Pass/Fail) e evidências (screenshot, logs).
    - Registrar bugs encontrados: descrever passo para reproduzir, comportamento atual vs esperado.
    - Atenção especial a integrações: verificar se ações cascata ocorreram (ex.: inscrição gerou notificação? confirmar no log ou email).
    - Testar *criterios de aceitação originais*: cada requisito do escopo - por ex., "sistema deve permitir cancelamento de inscrição?" se estava nos req. Se sim e não implementamos, anotar como falta (ou identificar como requisição não funcional? Clarificar).
- **Testes automatizados complementares:**
    - Rodar suíte de testes unitários novamente – garantir tudo verde.
    - Talvez criar **teste de integração automatizado**: ex.: usando Postman with Newman CLI to run a sequence: Signup, Login, List events, Inscribe, Check email stub. Configurar isso como part of CI.
    - **Teste de carga leve:** com JMeter:
    - Script: 10 threads, cada um faz login e 5 inscrições (maybe simulate 10 users concurrently).
    - Observar se sistema responde sem erros 5xx, e medir tempo médio das requests.
    - Se possível, incrementar até notar degradação.
    - Não exagerar (lab environment).
    - Anotar se atendeu possíveis metas (ex.: suportou 20 req/seg sem erro).
    - **Teste de falha de serviços:**
    - Derrubar serviço Pagamentos e tentar inscrever em evento pago: ver se front mostra erro amigável e sistema não quebra globalmente.
    - Derrubar Notificações: fazer inscrições e depois subir Notificações, verificar que emails saem (mensagens ficaram na fila).
    - Derrubar DB de eventos enquanto faz listagem: ver se erro propagado tratado (talvez não feito, mas ver comportamento e anotá-lo).
    - **Teste de segurança:**

    - Tentar SQL injection em algum param (ex.: email as `' OR 1=1--`). Esperado: tratado como string normal (ORM previne).
    - XSS: tentar inserir `<script>` num campo feedback e depois ver se ao exibir feedback em HTML (se tivesse) executaria. Provavelmente só testável se front mostra sem escape – se front never displays, ok.
    - Verificar cookies, tokens – ensure we use HTTPs in production (maybe not in dev), mention as a note.
    - Pentest básico: usar ferramenta como OWASP ZAP in passive mode to see any glaring issue.
- **Registro de resultados:** compilar um **Relatório de Testes**:
    - Resumo da abordagem.
    - Tabela de casos de teste com resultados (pass/fail).
    - Lista de bugs encontrados e seu status (resolvido/não resolvido). Marcar críticos x menores.
    - Cobertura de testes unitários (%). Ex.: "Cobertura 75% classes, 60% linhas no serviço X, etc." [11] .
    - Resultados de teste de carga (ex.: "Aguentou 20 req/seg sem erros, tempo médio 300ms em listar eventos").
    - Revisão de requisitos: tabela de requisitos vs implementado/testado (para demonstrar traçabilidade).
    - Aceite do sistema: declaração se o sistema atendeu todos requisitos (justificar faltantes se algum).
    - Observações: por ex., “notamos que sob carga de 50 req/seg sistema ficou lento, mas isso está fora do escopo do curso melhorar agora”.
- **Correção de bugs finais:** para cada bug relevante encontrado:
    - Se trivial ou crítico, corrigir imediatamente no código e reexecutar os testes relacionados.
    - Se muito complexo e pouco tempo, documentar workaround ou aceitar como limitação conhecida.
    - Garantir que nenhum bug crítico (que impede funcionalidade essencial) permaneça.
    - Regressão test: se mexeu em algo, rodar rapidamente os testes correlatos para garantir não quebrou outro ponto.
    - Atualizar documentação se bug fix altera comportamento ou requer nota (ex.: "limite x alterado").
- **Re-run test cases:** após correções, repetir os casos falhados para ver se agora passam.
    - Atualizar relatório (versão final).
- **Aprovação formal:** professor (ou líder) revê o relatório e talvez re-testa pontos críticos para validar. Em ambiente acadêmico, isso se confunde com a própria avaliação do trabalho. Internamente, alunos podem fazer um *checklist final* de “pronto para apresentar”: todos semáforos verdes.

- **Uso do CERTO para testes/documentação:**

    - Alunos podem usar ChatGPT para ajuda em escrever casos de teste ou documentar algo:
    - *“Contexto:* sistema de microserviços X, queremos casos de teste para funcionalidade Y; *Exigências:* cobrir casos normal e erro; *Referências:* descrição da funcionalidade; *Tarefa:* sugerir possíveis casos de teste; *Observações:* —.”* – IA sugere cenários que talvez não pensaram.
    - *“Contexto:* precisamos escrever um resumo do teste de carga realizado; *Exigências:* incluir números e conclusão; *Referências:* throughput e latência medidos; *Tarefa:* elaborar um parágrafo de conclusão sobre performance; *Observações:* —.”* – IA pode ajudar a formular bem.

    - Para documentação final (próxima semana), já podem pedir IA para revisar texto de algum manual, mas cuidado para não introduzir erros – usar para melhorar linguagem se preciso.

- **Atividade avaliativa/entregável sugerido: Relatório de Testes Finais e Qualidade** [57] [11] :

- Documento consolidado contendo:
    - Sumário dos testes realizados (funcionais, não-funcionais).
    - Tabela de casos de teste com resultados.
    - Métricas (cobertura, desempenho).
    - Lista de bugs corrigidos com breve descrição das correções.
    - Confirmação de atendimento de requisitos (todos critérios de aceitação marcados).
    - Observações finais (limitações conhecidas, lições sobre qualidade).
- *(Avaliação:* abrangência e profundidade dos testes (cobriram todos aspectos?); rigor e clareza do relatório – se apresenta evidências e conclusões objetivas; qualidade do produto final – idealmente poucos ou nenhum bug restante, demonstrando que a equipe garantiu qualidade. O relatório serve também como evidência para a nota, documentando o estado do projeto.)

Semana 14: Documentação Final e Preparação da Entrega

- **Objetivo pedagógico:** Reunir e produzir toda a documentação final do projeto, tanto para usuários quanto técnica, consolidando o aprendizado em forma escrita. Os alunos irão criar manuais, atualizar diagramas finais e preparar o *pacote final* para entrega. Também iniciam os preparativos da apresentação final (slides, demo). Ao final da semana, todo material escrito do projeto estará pronto e revisado.

- **Conteúdos teóricos abordados:**

- **Documentação de Software:** importância e tipos:
    - **Manual do Usuário:** orientado a usuários finais, explica como usar o sistema (passo a passo para realizar tarefas, screenshots). Foca em funcionalidades, não em detalhes técnicos.
    - **Manual do Desenvolvedor/Implantação:** orientado a futuros mantenedores. Contém instruções de como configurar e rodar o ambiente, estrutura do código (resumo de cada microserviço, tecnologias usadas, onde estão configurados), decisões arquiteturais tomadas (pode anexar ADRs se houver), e instruções de implantação (Docker/K8s usage). Inclui detalhes de endpoints (pode referenciar documentação OpenAPI).
    - **Documentação de Arquitetura:** poderia ser parte do dev manual, mas enfatiza visão arquitetural: diagramas atualizados (de componentes, implantação, sequência de fluxo), explicação de como componentes interagem, como escalaria, etc.
    - **Documentação de API:** provavelmente já atendido pelo Swagger, mas caso necessário, incluir endpoints principais e exemplos de uso.
    - **Registro de decisões (ADR):** se a equipe usou isso, anexar ou resumir as principais (por ex., “Decidimos usar JWT para auth por X motivo”).
- **Ferramentas de documentação:** mention possibilmente MkDocs, GitHub Wiki, etc., mas como é entrega acadêmica, um documento Word/PDF bem formatado ou Markdown serve. Sugerir usar diagramas criados no Draw.io ou Lucidchart embutidos.
- **Revisão da escrita:** importância de clareza, concisão, sem erros gramaticais. Revisão cruzada entre membros.
- **Packaging do projeto:** listar todos itens a entregar: código-fonte, documentação, relatórios, slides, etc. Como organizar (talvez um link para repositório, mais docs em anexo).

- **Slide Deck (Apresentação):** estrutura recomendada:
    - Título, equipe, objetivo do projeto.
    - Motivação e escopo (o problema que resolve).
    - Arquitetura (diagrama e principais decisões).
    - Demonstração (talvez capturas ou flow).
    - Destaques técnicos (dificuldades superadas, inovações).
    - Resultados (talvez mencionar testes, se atingiu objetivos).
    - Lições aprendidas e conclusões.
    - Backup slides (se esperar perguntas sobre tech details).
- **Storytelling:** lembrete de tornar a apresentação compreensível, contar uma história de como evoluíram (mas isso mais para apresentação, porém pode tocar aqui).

- **Preparação para banca/perguntas:** apontar que documentar bem ajuda a responder questionamentos (ex.: porque escolheram tal tecnologia – já está justificado no doc, etc.).

- **Práticas propostas:**

- **Atualizar Diagrama e Arquitetura final:** Caso alguma modificação tenha ocorrido durante a implementação (ex.: adicionaram serviço, mudaram comunicação), atualizar o diagrama de arquitetura global. Refinar desenhos, garantindo que estejam legíveis e representem a versão final (incluindo Pagamentos, Notificações, Feedback, Gateway, DBs, etc.).
    - Fazer o mesmo com diagrama de domínio se houve mudança em entidades.
    - Colocar todos diagramas com legenda, títulos.
- **Compilar Manual do Usuário:**
    - Escrever instruções passo a passo para usar o sistema:
    - Como acessar (URL, credenciais de teste).
    - Como se registrar, logar.
    - Como navegar pela lista de eventos, inscrever-se.
    - Como realizar pagamento (informar que é simulado e qualquer número funciona, por ex.).
    - Como verificar email (se usar MailHog, explicar como acessar interface do MailHog ou se simplesmente ver log).
    - Como fornecer feedback.
    - Incluir *screenshots* de cada etapa principal para facilitar entendimento visual.
    - FAQ curto se útil (ex.: "E se esquecer senha?" – talvez não implementado, mas pode citar como limitação).
    - Linguagem acessível, evitar jargão técnico.
    - Revisar completude: o usuário consegue usar 100% do sistema com esse guia?
- **Compilar Manual do Desenvolvedor/Implantação:**
    - Estrutura sugerida:
    - **Introdução:** resumo do sistema, contexto (um para cada doc ou repetido, ok).
    - **Arquitetura:** incluir diagrama principal e descrição textual de cada componente (microserviço X faz Y, comunica via Z).
    - **Tecnologias:** listar todas com versões (ex.: Java 17/Spring Boot 3, Python 3.10/FastAPI 0.8, React 18, Postgres 14, etc.).
    - **Estrutura de pastas/repositórios:** se monorepo, explicar layout; se multi-repo, listar repos URLs.
    - **Configuração do ambiente de dev:** requisitos (Docker, Java, etc.), variáveis de ambiente, comandos para rodar cada serviço local (ex.: mvn spring-boot:run, uvicorn main:app etc.) – embora provavelmente rodamos via docker, mas bom explicar ambos.

    - **Build e Deploy:** instruções de uso do Docker Compose (ex.: comando `docker-compose up -d` e quais portas será exposto). Se K8s: explicar aplicar YAMLs, etc.
    - **Pipeline CI/CD:** se configurado, descrever (ex.: “GitHub Actions build on push, runs tests, builds images”).
    - **Detalhes de implementação:** por serviço, destacar particularidades:
        - Banco de dados: estrutura (pode incluir modelo relacional simplificado ou exemplo de documentos Mongo).
        - Integrations: ex.: "Serviço Inscrições chama Pagamentos via REST no endpoint ..., Notificações via RabbitMQ queue 'X'." – essas coisas para compreensão futura.
        - Config de segurança: ex.: JWT secret em env, expirações.
        - Dependências e como atualizar (ex.: libs usadas).
    - **Execução de Testes:** como rodar testes unitários (mvn test, etc.), onde estão relatórios (if any).
    - **Como contribuir:** (se fosse open source, mas no contexto, talvez pular).
    - **Possíveis melhorias futuras:** curta lista de features não implementadas ou melhorias identificadas (útil se alguém for continuar projeto).
    - Garantir tom técnico porém claro.
    - Incluir referências a arquivos (ex.: "ver arquivo docker-compose.yml para configuração de serviços.").
    - **Anexar documentação de API:** talvez incluir um apêndice com a tabela de endpoints e descrições (ou referenciar swagger link).
    - **Anexar ou incluir** diagramas UML (domínio, sequência se fizeram) refinados.
- **Revisão da Documentação:** dividir para revisão cruzada: um membro revisa manual usuário, outro dev, etc., para eliminar erros e verificar se alguém de fora entenderia.
- **Criar Pacote Final:** organizar todos artefatos:
    - Código-fonte: talvez já no repositório, mas entregar zip se exigido.
    - Contêineres: se pediu, entregar imagens ou Dockerfiles.
    - Documentação: PDF/Doc dos manuais, relatório de testes, etc., tudo nomeado claramente.
    - Slides de apresentação (rascunho se já tiver).
    - Uma capa/README resumindo conteúdos e agradecimentos/dados da equipe.
- **Planejamento da apresentação final:**
    - Começar a montar os **slides** (se ainda não): distribuir tópicos entre membros (quem fala sobre arquitetura, quem narra demo, etc.).
    - Ensaiar breve (pode ser no início da semana 15) mas se der, um ensaio agora para ajustar tempo.
    - Checar se algum demo vídeo é necessário (às vezes grupos gravam um vídeo de tela como backup se ao vivo falhar – considerar).
    - Refinar storytelling: introdução impactante (problema), depois solução, etc.
    - (Detalhes de apresentação mais na semana 15 se for o foco lá).

- **Uso de ChatGPT para documentação:**

    - Podem pedir para revisar linguagem: *“Leia este trecho do manual do usuário (Contexto) e sugira melhorias de clareza; Exigências: manter linguagem simples para leigo; Referências: [texto]; Tarefa: reescrever de forma mais clara; Observações: em português.”*
    - Podem pedir formatação de um trecho ou ver se algo técnico está bem explicado: *“Explique em termos simples como o microserviço X funciona”* para usar no doc de arquitetura se travar em simplificar.

    - Aproveitar que o método CERTO também se aplica a documentação – basicamente contextualizar a IA para obter parágrafos mais estruturados.
    - **Atenção:** revisar tudo que a IA produzir para ver se não inseriu algo incorreto do ponto de vista do projeto.

- **Atividade avaliativa/entregável sugerido: Pacote de Documentação Final do Projeto** [56] [12]:

- **Manual do Usuário** (em português, ilustrado).
- **Manual do Desenvolvedor/Arquitetura** (com diagramas, instruções de deploy).
- **Documentação das APIs** (pode ser incorporada ou separada, e/ou link para Swagger).
- **Diagramas finais** (Arquitetura, Domínio, etc. anexados/embutidos).
- Todos em formato apresentável (PDFs ou impressos encadernados se físico).
- *(Avaliação:* completude – se a documentação cobre os aspectos necessários para usar e dar manutenção no sistema; clareza e organização – linguagem apropriada para o público alvo de cada documento, boa estrutura, uso de diagramas; profissionalismo – aparenta documento de projeto real, com padronização e cuidados.)

Semana 15: Refinamento Final do Deploy e Ensaios de Apresentação

- **Objetivo pedagógico:** Garantir que o sistema final está adequadamente implantado e preparado para demonstração e que a equipe está apta a apresentar o projeto de forma coesa e confiante. Os alunos farão verificações finais no ambiente de produção (ou demo), corrigirão qualquer problema remanescente de última hora e ensaiarão a apresentação final, integrando conceitos e reflexões sobre o aprendizado. Ao final da semana, todos estarão prontos para a avaliação final.

- **Conteúdos teóricos abordados:**

- **Deploy final vs. staging:** ver se há diferenças – possivelmente não, se staging já era quase produção. Se houver, discutir *go live*: apontar DNS para server, etc. (Talvez teórico, se contexto permitir).
- **Plano de contingência de demo:** práticas para evitar imprevistos em apresentação:
    - Ter base de dados pré-populada com dados significativos (ex.: vários eventos, usuários de teste) para evitar ter que cadastrar tudo ao vivo.
    - Ter internet backup (se dependente de internet).
    - Ter vídeo demo offline pronto caso live demo falhe (não sempre exigido, mas é safe guard).
    - Ensaiar no equipamento que será usado, para checar resolução, etc.
- **Técnicas de apresentação eficaz:**
    - Falar com clareza, evitar textos longos em slides, preferir imagens (diagramas, screenshots).
    - Tempo: se têm 15 min, ensaiar para ficar dentro.
    - Divisão de fala: todos membros devem falar (se requerido), então coordenar transições suaves.
    - Antecipar perguntas típicas da banca: “Por que microserviços e não monolito?”, “Como garantir segurança de X?”, “O que fariam diferente?”, etc. Treinar respostas.
- **Retrospectiva final:** recolher do grupo as lições aprendidas no projeto e sobre trabalhar com microserviços. Consolidar isso para mencionar se couber na apresentação ou relatório final (às vezes pedem).

- **Autoavaliação e fechamento:** talvez o professor peça autoavaliação – discutir como medirão sucesso pessoal e da equipe (dentro do grupo).

- **Ética e uso de IA:** vale aqui uma discussão final (talvez breve) sobre a experiência de integrar IA (ChatGPT) no workflow: ajudou? Tomar cuidado p/ não vazar dados proprietários? Isso pode ser mencionado para valorizar a inovação do método CERTO usado [58] [59].

- **Práticas propostas:**

- **Verificação final do ambiente de produção:**
    - Se a apresentação será feita localmente, garantir que todos containers estão atualizados com últimas versões (refazer build/pull imagens se mudou algo).
    - Subir o sistema e deixar pré-carregado com dados:
    - Criar usuários de demonstração (um admin, vários normais) com senhas conhecidas.
    - Inserir eventos de exemplo (um lotado, um passado para feedback, etc.).
    - Talvez inserir algumas inscrições e feedbacks prévios para mostrar histórico.
    - Realizar um *smoke test* final: simular as ações que serão mostradas na demo para garantir estão funcionando hoje (as vezes algo quebra no último minuto – melhor descobrir agora).
    - Congelar esse estado de BD ou ter script para regenerar rápido caso precise reset.
    - Checar logs – limpar ou deixar nível em INFO (para demo não poluir muito terminal se mostrando).
    - Preparar telas/monitores: se mostrando logs ao vivo, usar um tail -f organizado; se mostrando MailHog UI, ter ela aberta.
- **Finalizar slides da apresentação:**
    - Integrar resultados dos testes no slide de resultados (ex.: “Testes: 0 bugs críticos, 95% casos aprovados, suportou 50 req/seg”).
    - Incluir gráficos se for impressionante (ex.: um gráfico simples de tempo de resposta vs usuários).
    - Inserir imagens do sistema (um screenshot da tela principal do app).
    - Revisar ortografia e tempo.
    - Ensaiar com equipe cronometrando:
    - Iniciar com apresentação do problema e solução.
    - Depois mostrar arquitetura (um membro explica diagrama).
    - Depois *demo ao vivo*: aqui dividir quem navega e quem narra o que acontece. Ensaiar a fala durante a demo (“Agora vou logar como João, um usuário comum... aqui vemos os eventos...” etc.).
    - Garantir demo não tome mais que metade do tempo, para sobrar tempo de conclusões.
    - Ensaiar respostas: simular professor perguntando “O que foi mais desafiador? Que melhorias fariam?”. Cada membro responde uma diferente para não ficar silêncio real.
- **Reflexão integradora:**
    - Dedicar alguns minutos a discutir em grupo os conceitos aprendidos:
    - Como foi projetar e implementar microserviços vs experiências anteriores (monolito): vantagens percebidas (deploy isolado, etc.) e desvantagens (complexidade infra).
    - Importância de DevOps (Docker, CI) que vivenciaram.
    - Uso da IA: cada um compartilhar uma ocasião que ajudou no projeto com CERTO – isso reforça internalização do método (e pode ser citado na apresenação).
    - Essa reflexão pode ser documentada ou apenas discutida para consolidar conhecimento (depende se professor pede um relatório reflexivo).
    - Pode também reforçar trabalho em equipe, comunicação, etc. (soft skills desenvolvidas).

- **Últimos ajustes e congelamento:**
    - Se qualquer bug menor ficou e se der tempo, pode arrumar agora (mas evitar mexidas grandes sem tempo de testar – risco).
    - Por segurança, tirar snapshot/backup do ambiente funcionando (ex.: exportar banco, salvar docker images) para recuperar se algo der problema no dia.
    - Combinar logística: quem leva notebook, quem apresenta tela, etc.
- **Aplicação do CERTO na preparação:**
    - Podem usar ChatGPT para polir a apresentação:
    - *"Contexto:* temos 10 minutos para apresentar projeto X; *Exigências:* queremos explicar em linguagem simples para público misto; *Referências:* forneça resumo do projeto e pontos-chave; *Tarefa:* sugerir um roteiro ou ordem de apresentação ideal; *Observações:* —."* – IA pode sugerir sequência ou analogias.
    - *"Contexto:* apresentação final de curso de eng. de software; *Exigências:* pode haver perguntas difíceis; *Tarefas:* que perguntas possivelmente serão feitas sobre microserviços e como responder;*Observações:* —."* – IA lista perguntas comuns (ex.: "Por que escolheram tecnologia tal? Como escalaria? O que fariam diferente?") para equipe treinar.

- **Motivacional:** professor pode dar última orientação de tranquilizar, lembrando-os de mostrar confiança e domínio, pois de fato percorreram todos os tópicos (é muita coisa, mas agora consolidada).

- **Atividade avaliativa/entregável sugerido:** *Checkpoint final* – **Revisão Geral e Ensaio**:

- Não há entrega formal nesta semana, mas o professor pode avaliar a dedicação da equipe nos preparativos. Poderia haver um *ensaio avaliativo* fechado, onde apresentam para o professor e recebem um último feedback para ajustar antes da apresentação oficial.
- Critérios observados: coordenação do time, narrativa clara, demonstração fluida, capacidade de responder perguntas sobre o projeto integrando conceitos vistos.
- *(Avaliação:* preparação e profissionalismo – a equipe demonstra estar pronta, com domínio do conteúdo e segurança para apresentar e implantar o sistema sem incidentes.)

Semana 16: Apresentação Final do Projeto e Retrospectiva

- **Objetivo pedagógico:** Permitir que os alunos exibam o resultado de seu trabalho, comunicando efetivamente as soluções implementadas e os conhecimentos adquiridos, e consolidar o aprendizado por meio de reflexão final. Nesta semana ocorre a apresentação final para a turma/banca avaliadora e uma retrospectiva pós-projeto. Ao final, espera-se que os alunos tenham demostrado o funcionamento do sistema, defendido as decisões tomadas e internalizado as lições aprendidas para futuros projetos.

- **Atividades principais:**

- **Apresentação Final:** Cada equipe realiza sua apresentação conforme planejado. Itens esperados na apresentação:
    - **Introdução:** contextualização do problema (gestão de eventos corporativos) e objetivos do projeto.
    - **Descrição da Solução:** funcionalidades implementadas e breve comparação com escopo inicial (se tudo feito ou algo ficou fora).
    - **Arquitetura:** mostrar diagrama de microserviços e explicar componentes e interações [13] . Destacar o uso de padrões (API Gateway, database per service, mensageria para notificações, etc.).

    - **Demonstração ao vivo:** executar o sistema real:
    - Mostrar tela de login, realizar login com usuário demo.
    - Navegar pela lista de eventos (destacar frontend React comunicando com backend via gateway).
    - Demonstrar inscrição num evento gratuito e num pago:
        - Para o pago, mostrar talvez logs ou mail confirmando pagamento e email (poderiam abrir o MailHog inbox se configurado).
    - Mostrar painel admin (se existente) para criar evento ou visualizar feedback.
    - Demonstrar envio de feedback (talvez previamente, ou se event passado, simular ao ajustar data).
    - **Resiliência demo (opcional):** alguns grupos gostam de fazer "teste de caos" na apresentação – ex.: *derrubar* um microserviço e mostrar que o sistema como um todo continua (como sugere o plano: derrubar serviço de pagamentos e mostrar que inscrição fica pendente) [60] . Se forem confiantes, podem tentar: por ex., derrubar Notificações container antes de uma inscrição, depois subi-lo e mostrar que email sai com atraso, etc. Isso impressiona banca se der certo.
    - Monitorar o tempo: a demo deve ser fluida; se algum passo falhar, ter backup (slides/vídeo) para explicar.
    - **Resultados e Qualidade:** apresentar dados dos testes:
    - “Nosso sistema atende X requisições/segundo”, “Cobertura de testes unitários em 80%”, “Sem bugs críticos abertos após testes”.
    - Talvez uma rápida menção de desafios superados (ex.: "tivemos dificuldades com autenticação no início, mas resolvemos implementando JWT e funcionou bem").
    - **Lições Aprendidas:** cada membro pode falar uma frase do que aprendeu (tecnologia nova, importância de planejamento, uso de IA, trabalho em equipe). Isso faz um fechamento pessoal e mostra reflexão.
    - **Agradecimentos e Q&A:** agradecer orientações, abrir para perguntas.
- **Sessão de Perguntas:** A banca/professor faz perguntas. Os alunos respondem demonstrando domínio:
    - Exemplos: “Como vocês lidariam com aumento de usuários (escalar)?” – Resposta: falar sobre replicar serviços, usar Kubernetes, auto scaling, etc., e que arquitetura de microserviços facilita escalar só os necessários [61] [15] .
    - “Se um serviço falhar, como o sistema se comporta?” – Responder com o que implementaram: ex.: “Usamos fila para notificação, então se esse serviço falhar, as inscrições ainda ocorrem e o email é enviado quando ele voltar – aumentamos resiliência” [26] .
    - “Por que escolheram tecnologia X para tal serviço?” – Justificar com base no que escreveram (ex.: Python no Notificações por simplicidade e boas libs de email, etc.).
    - “Que diferenças sentiu entre desenvolver monolítico vs microserviços?” – Esperam ver que entendeu trade-offs (complexidade vs escalabilidade).
    - Perguntas sobre CERTO/IA: “Usaram ChatGPT? Como?” – alunos podem relatar: “Sim, usamos o método CERTO em várias etapas – por exemplo, na definição de requisitos [18] , na solução de um bug complexo – e isso nos ajudou a ganhar tempo e ideias, mas sempre validamos os resultados” [58] .
- **Retrospectiva Final:** após apresentações (pode ser em aula seguinte informalmente se tempo), conduzir uma retrospectiva geral da disciplina:
    - O professor e alunos discutem o que funcionou bem ao longo do curso/projeto, o que poderia ser melhor (talvez ritmo, suporte, etc.).

- Alunos avaliam o uso de IA (método CERTO) no curso – foi útil? Atrapalhou algo? Como integrar no futuro? [62] [59]
- Destacar conquistas: equipes conseguiram entregar projetos complexos, aprendendo tecnologias de ponta (Docker, microserviços, CI) em curto tempo – isso deve ser celebrado.
- Coletar *feedback* dos alunos sobre a disciplina, que servirá para melhorar edições futuras.
- O professor faz um fechamento motivador, conectando a experiência com expectativas do mercado (ex.: "Hoje vocês basicamente vivenciaram como é desenvolver um sistema corporativo real, usando práticas modernas – isso os deixa mais preparados para desafios fora da universidade.").

• **Entregáveis da semana: Apresentação Final e Slides** [13] [14] :

• Os slides (arquivo PPT/PDF) são entregues ao professor.
• Possível entrega de um **vídeo demo** gravado (se exigido ou para registro).
• *Avaliação:* desempenho na apresentação (clareza, segurança, se cobriu os pontos solicitados, se demonstrou funcionamento real do sistema). Também a capacidade de respostas na Q&A – indicando domínio individual e coletivo.
• **Retrospectiva documentada:** não necessariamente avaliada, mas se o professor pedir um relatório reflexivo, pode ser entregue também.

(Observação: Este plano didático seguiu todas as etapas desde introdução, passando por arquitetura, design patterns, desenvolvimento full-stack (back e front), integração contínua, segurança, testes, até o deploy final e apresentação. A metodologia C.E.R.T.O. foi aplicada em diversas atividades para fomentar o uso orientado da IA como ferramenta de apoio ao longo do projeto [63] [64] . Os checkpoints e revisões inseridos garantem a integração contínua de conceitos e o acompanhamento do aprendizado, resultando em uma experiência completa de desenvolvimento de um sistema corporativo moderno.)

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] Plano de Desenvolvimento da Disciplina __Desenvolvimento de Sistemas Corporativos__ (6º Período).pdf
file://file-7QWTKooyskmr6TV2Kw9trM

[15] [16] Método CERTO em Desenvolvimento Corporativo_.pdf
file://file-W4xWVEKRuZWjeZuEoH5nRW