Laboratório de Desenvolvimento de Software
2025_2026

Tema 1 —Distribuição de Serviço Docente (versão 02/11/2025)

Objetivo geral

Desenvolver uma aplicação modular e distribuída que apoie a resolução e gestão da distribuição de serviço docente num contexto académico.
A solução deverá incluir: Base de dados relacional, API REST, GraphQL e microserviços gRPC; Mecanismos de autenticação e controlo de permissões;
Interface Web desenvolvida em Flutter; Garantias de concorrência, segurança e auditabilidade.

Etapa 0 — Análise e Modelação

Levantamento de requisitos, modelação entidade-relação, conversão para modelo lógico PostgreSQL e validação da normalização e integridade:

- Identificar as necessidades funcionais do sistema (ex.: gestão de docentes, áreas científicas, cursos, unidades curriculares, horas de contacto e distribuição do serviço docente).
- Recolher os requisitos não funcionais (ex.: desempenho, persistência, integridade dos dados, modularidade).
- Definir casos de utilização representativos (ex.: “atribuir docente a UC”, “consultar serviço de um docente”, “atualizar carga horária”).
- Especificar os principais atores do sistema (coordenador de curso, docente, administrador, etc.) e as suas permissões.
- Elaborar o modelo conceptual (entidade–relação), identificando as entidades e os relacionamentos entre: Departamentos, Áreas Científicas, Docentes, Cursos, Unidades Curriculares, Horas de Contacto.
- Converter o modelo conceptual num modelo lógico relacional compatível com PostgreSQL, aplicando normalização e definindo chaves primárias e estrangeiras.
- Validar o modelo com base nos requisitos funcionais e assegurar que suporta as operações pretendidas nas etapas seguintes.

Etapa 1 — Setup e Infraestrutura

Criação de ambiente Docker com PostgreSQL, Next.js e GraphQL. Configuração da base de dados, scripts e testes de conectividade:

- Criar uma máquina virtual com Docker e Docker Compose, configurando os serviços:
- -PostgreSQL (base de dados principal);
- -Next.js (API inicial).
- Criar a base de dados a partir das scripts SQL e inserir registos de teste.

Etapa 2 — API REST Base

Desenvolvimento de endpoints REST para departamentos, áreas, docentes, cursos e UCs, com validações e testes Postman (ver anexo I). Garantir consistência entre chaves estrangeiras e integridade referencial da base de dados.

Etapa 3 — API GraphQL

Implementação de servidor GraphQL (Apollo, ou outro) com schemas e resolvers. Suporte a queries aninhadas e testes em Apollo Studio (ou outro):

- -Criar um serviço Docker (graphql) ligado à mesma base de dados PostgreSQL.
- -Implementar um servidor GraphQL com:
- -Schemas e resolvers para departamentos, áreas, docentes, ucs e para as entidades da distribuição de serviço.
- -Permitir queries aninhadas (por exemplo, UC → docentes → área científica).
- Testar e documentar consultas exemplificativas.

Etapa 4 — Arquitetura de Microserviços gRPC

- Dividir a aplicação em dois serviços:
  - Serviço A (interno, gRPC) → gere docentes (ou outra entidade principal) e comunica diretamente com a base de dados.
  - Serviço B (gateway HTTP) → expõe endpoints REST e comunica com o serviço A via gRPC.
- Definir o contrato de comunicação (ficheiro .proto) com mensagens e métodos.
- -Implementar a comunicação entre serviços usando as bibliotecas adequadas (@grpc/grpc-js, @grpc/proto-loader).
- -Garantir que o serviço B não acede diretamente à base de dados.
- -Testar a cadeia: Postman → Gateway HTTP → gRPC → BD.

Etapa 5 — Módulo de Gestão da Distribuição de Serviço Docente

Integração das APIs REST, GraphQL e gRPC. Implementação da lógica de atribuição de docentes a UCs, validação de horas e regras de integridade.

Categoria	Exemplo de validação	Onde / Como implementar
Identidade e existência	Docente e UC existem e estão ativos	Trigger ou verificação no serviço
Coerência relacional	Docente pertence à área da UC	Serviço (política de negócio)
Unicidade	Não duplicar atribuição (docente + UC + tipo + ano)	UNIQUE na BD

Categoria	Exemplo de validação	Onde / Como implementar
Limites de horas	Soma de horas atribuídas ≤ horas de contacto da UC	Trigger BD
Horas válidas	Horas ≥ 0	CHECK BD
Docente ativo	Bloquear atribuição a docentes inativos	Trigger BD ou Serviço
Elegibilidade	Docente tem grau/área adequados	Serviço
Limites individuais	Total de horas ≤ carga máxima por docente	Serviço
Conflito de horário	Não sobrepor horários do mesmo docente	EXCLUSION CONSTRAINT (BD)
Relatórios	Horas atribuídas vs. horas disponíveis	Views / Materialized Views

Erros e respostas padronizadas:

- 400 Bad Request → dados mal formatados.
- 404 Not Found → entidade inexistente.
- 409 Conflict → duplicação ou ultrapassagem de limite.
- 412 Precondition Failed → quebra de política (ex.: docente inelegível).
- 422 Unprocessable Entity → conflito lógico (ex.: sobreposição de horários).

Etapa 6 — Autenticação e Permissões

Objetivo: Implementação de autenticação JWT (stateless) e RBAC. Gestão de tokens (login, refresh, logout, revogação). Criação de APIs públicas e controlo de acessos com perfis de utilizador.

Arquitetura: Access JWT (10–15 min) com claims {sub, sid, tv, role, exp, courseIds}; sessions na BD por login; refresh tokens como strings aleatórias (NÃO JWT) guardadas com hash e associadas à sessão; revogação imediata por sessão (revoked_at) e logout global por utilizador (token_version).

Esquema (sugerido) para BD:

- users(id, email, password_hash, role, token_version DEFAULT 1,...)
- sessions(id UUID PK, user_id FK, family_id UUID, revoked_at TIMESTAMPTZ NULL, expires_at TIMESTAMPTZ NOT NULL,...)

- refresh_tokens(id UUID PK, session_id FK, token_hash TEXT, is_revoked BOOLEAN, expires_at TIMESTAMPTZ,...)
- Middleware (pseudocódigo): verificar assinatura; ler {sub,sid,tv}; BD→ sessão por sid (existente, ativa, não expirada); BD→ utilizador por sub (token_version==tv); anexar claims e aplicar RBAC por rota.

Endpoints obrigatórios:

- POST /auth/login → cria sessão e refresh; devolve access (com sid,tv) + refresh.
- POST /auth/refresh → verifica e roda refresh (revoga antigo; cria novo na mesma família); devolve access+refresh.
- POST /auth/logout → revoga sessão atual (sessions.revoked_at=now); revoga refresh da sessão.
- POST /auth/logout-all → incrementa users.token_version e revoga sessões ativas do utilizador.
- APIs públicas (/api/public/**) → API Key + rate limiting (sem dados sensíveis).
- RBAC: perfis Administrador, Coordenador, Docente e Convidado; guardas por rota (requireRole). Regras finas validadas no serviço com base nas claims (ex.: courseIds).

Segurança mínima: bcrypt/argon2; HTTPS/TLS; CORS restrito; access 10–15 min; refresh 7–30 dias (hash na BD); auditoria de login/refresh/logout/401/403/revogações; seeds de utilizadores (um por perfil).

Testes/aceitação: login válido/ inválido; rotas protegidas com/sem token; RBAC por perfil (200/403); refresh com rotação; revogação por sessão (401 imediato); logout-all (401 imediato); APIs públicas com/sem API Key (200/401/429).

Etapa 7 — Interface Web e Mobile em Flutter

Desenvolvimento de interface Flutter Web responsiva e segura. Consumo das APIs protegidas, autenticação, refresh automático e gestão de estado com Provider/Riverpod.

Etapa 8 — Relatórios e Demonstração Final

Relatório 1 (até 23 de novembro)

Documentar a estrutura técnica do sistema até este ponto, garantindo que o modelo de dados, as APIs básicas e os mecanismos de integridade estão corretamente implementados e coerentes com os requisitos do projeto.

Conteúdo

1. Resumo da arquitetura (1 página máx.)

- Diagrama de componentes principais (PostgreSQL, API REST, Docker containers).

- Explicação breve de como os serviços comunicam (REST, portas, variáveis de ambiente).
- Descrição sumária dos objetivos do sistema (gestão de serviço docente, distribuição de horas, papéis de utilizador).

2. Modelo de dados

- Diagrama Entidade–Relação (E–R) completo e legível.
- Lista de tabelas com breve explicação (finalidade e principais campos).

Exemplo:

Tabela	Finalidade	Campos-chave
docente	Armazena dados dos docentes	id_doc, nome, area, grau
uc	Unidades curriculares	id_uc, nome, horas_contacto
atribuicao_docente_uc	Relação entre docente, UC e ano letivo	id_atribuicao, id_doc, id_uc, tipo, ano_letivo

Mapa de chaves primárias (PK) e estrangeiras (FK).

3. Endpoints REST implementados

Tabela resumo dos endpoints (CRUD) já disponíveis:

Método	Endpoint	Descrição	Códigos HTTP
GET	/docentes	Lista todos os docentes	200, 500
POST	/docentes	Cria novo docente	201, 400, 409
PUT	/docentes/:id	Atualiza docente existente	200, 404
DELETE	/docentes/:id	Elimina docente	204, 404

Sugestão: incluir captura de ecrã do Postman com 2–3 exemplos de pedidos e respostas.

4. Principais triggers e validações

Identificar e explicar as regras automáticas já implementadas na base de dados.
Exemplo de apresentação:

Categoria	Exemplo de validação	Implementação
Identidade	O docente e a UC devem existir antes da atribuição	FK id_doc, FK id_uc
Unicidade	Evitar duplicar combinação docente + UC + tipo + ano	Constraint UNIQUE(id_doc, id_uc, tipo, ano_letivo)

Categoria	Exemplo de validação	Implementação
Limites de horas	Soma de horas atribuídas ≤ horas da UC	Trigger valida_limites()
Horas válidas	Horas ≥ 0	Constraint CHECK(horas >= 0)
Coerência	Docente deve pertencer à mesma área da UC	Trigger check_area_docente()

5. Teste e validação

Descrever como foram testadas as regras: com scripts SQL ou Postman.
Registar códigos de erro e comportamentos esperados.
Indicar brevemente os casos de sucesso e erro validados.

6. Conclusão

Problemas encontrados e como foram resolvidos.
Próximos passos (autenticação e RBAC).

7. Formato e extensão

3 a 5 páginas (máx. 7 se incluir diagramas grandes).

Critério	Descrição	Peso	Tipo de feedback
1. Clareza do modelo de dados	Correção do diagrama E-R, normalização e coerência entre entidades e relações.	25 %	Comentários diretos no diagrama e texto explicativo.
2. Estrutura da base de dados	Uso adequado de PK/FK, constraints (CHECK, UNIQUE, triggers), normalização e integridade.	25 %	Identificação de melhorias e boas práticas SQL.
3. Implementação de APIs REST	Funcionalidade e coerência dos endpoints (GET, POST, PUT, DELETE), respostas e códigos HTTP.	25 %	Sugestões de uniformização e validação.
4. Testes e evidências	Apresentação de testes Postman, logs ou exemplos de sucesso/erro.	15 %	Feedback sobre clareza e completude dos testes.
5. Documentação	Estrutura, linguagem e formatação do relatório.	10 %	Observações de estilo e comunicação técnica.

Relatório 2 (até 07 de dezembro)

Laboratório de Desenvolvimento de Software
2025_2026

Este relatório deve descrever a implementação da **autenticação, autorização e controlo de acessos (RBAC)** na aplicação desenvolvida.
O objetivo é garantir que apenas utilizadores autenticados e com permissões adequadas possam aceder ou modificar recursos do sistema.

Critério	Descrição	Peso	Tipo de feedback
1. Correção do fluxo de autenticação	Implementação de login, refresh e logout de acordo com a arquitetura JWT + BD.	25 %	Indicação de incoerências ou vulnerabilidades.
2. Estrutura e segurança dos tokens	Definição clara de claims (sub, sid, tv, role, exp), expiração, rotação e revogação.	20 %	Feedback sobre expiração, validação e segurança.
3. RBAC e perfis de utilizador	Definição e aplicação correta dos papéis e permissões.	20 %	Sugestões de granularidade e consistência.
4. Documentação e diagramas	Diagrama do fluxo de autenticação e tabelas descritivas.	20 %	Anotações sobre clareza visual e textual.
5. Testes e evidências (Postman)	Casos de sucesso, erro e revogação corretamente demonstrados.	15 %	Feedback detalhado em casos de teste e logs.

Apresentação final (última semana de aulas)

Relatório técnico completo.
Demonstração final (apresentação).
Repositório Git com histórico coerente.
Scripts funcionais (docker-compose up, seeds, etc.).
Aplicação Flutter Web com login/logout e refresh automático.
Consumo de APIs protegidas e públicas.
Gestão de estado (Provider/Riverpod) implementada.

Critério	Indicadores observáveis	Peso	Tipo de feedback
1. Funcionalidade global	Todo o sistema funcional (login, RBAC, CRUD, BD, refresh).	25 %	Comentários sobre fiabilidade e completude.
2. Qualidade técnica e modularidade	Código limpo, modularidade (REST/GraphQL/gRPC), coerência de estruturas e serviços.	20 %	Sugestões de refatorização e arquitetura.
3. Segurança e autenticação	Implementação robusta das regras JWT, RBAC e revogação.	15 %	Avaliação de práticas seguras e auditoria.

Critério	Indicadores observáveis	Peso	Tipo de feedback
4. Interface Flutter Web	Funcionalidade, integração com backend, UX e responsividade.	15 %	Feedback visual e técnico.
5. Documentação final (relatório)	Clareza, diagramas, estrutura e comunicação técnica.	15 %	Observações sobre apresentação e redação.
6. Demonstração e defesa	Capacidade de explicar, justificar e responder a perguntas.	10 %	Feedback oral (qualitativo).

Avaliação final:

Fase	Entrega	Peso	Feedback	Integra na final?
Relatório 1	Relatório técnico (BD + APIs)	25 %	Sim (escrito)	Sim
Relatório 1	Relatório de segurança (JWT + RBAC)	15 %	Sim (escrito + oral breve)	Sim
Entrega final	Sistema completo + relatório + demo	60 %	Sim (oral e escrito)	—
Bónus (*)		10%		

(*) Nota: Ver bónus no anexo VIII

Anexos

Anexo I - Endpoints REST

-Implementar os endpoints REST seguintes, com as respetivas validações e códigos de erro (400, 404, 409), abrangendo as entidades principais: Departamentos, Áreas Científicas, Docentes, Cursos, Unidades Curriculares:
PUT /docentes/:id → atualiza os dados de um docente existente (ex.: nome, email, id_area, convidado). Validar se o docente e a área existe antes de atualizar. Se id do docente não existir devolver 404. Validar se o novo email proposto é duplicado (409 Conflict → email duplicado). Se dados em falta ou inválidos, devolver 400 Bad Request.
POST /docentes. Cria novo docente (nome, email, id_area, convidado). Deve falhar se o email já existir (409) ou se id_area não existir (400).
DELETE /docentes/:id → remove um docente.
DELETE /docentes/:id/inativar → marcar o docente como inativo (ativo = false).
GET /docentes/:id → devolve os detalhes de um docente específico (caso ainda não exista). Devolver 404 se o docente não for encontrado.
GET /docentes?incluirInativos=true. Usar query string opcional para devolver ativos e inativos.
GET /ucs → lista todas as UCs (id_uc, nome, ano_curso, ects, horas totais de contacto).
GET /ucs/:id → devolve os detalhes de uma UC, incluindo a distribuição de horas de contacto por tipo (dados da tabela uc_horas_contacto).
GET /ucs/:id/horas. Lista as horas de contacto da UC, por tipo.
POST /ucs/:id/horas. Adiciona ou atualiza a carga horária de um certo tipo de contacto.
GET /departamentos. Lista todos os departamentos (id_dep, nome, sigla, ativo).
GET /departamentos/:id. Devolve um departamento específico. Se não existir → 404.
POST /departamentos. Cria um novo departamento (nome, sigla). Ativo é TRUE por defeito.
PUT /departamentos/:id. Atualiza nome, sigla ou ativo.
DELETE /departamentos/:id. Remover ou marcar ativo = false, se já existirem áreas científicas associadas.
GET /areas. Lista todas as áreas científicas, incluindo id_area, nome, sigla, info do departamento associado (ex.: nome do departamento)
GET /areas/:id. Detalhe de uma área científica específica.
POST /areas. Cria uma nova área científica (nome, sigla, id_dep). Validar se id_dep existe (FK).
PUT /areas/:id. Atualiza sigla/nome/ativo/id_dep.
DELETE /areas/:id ou /areas/:id/inativar
GET /cursos. Lista cursos com campos básicos (id_curso, nome, sigla, tipo, ativo).
GET /cursos/:id. Detalhe de um curso.
POST /cursos. Cria um curso (nome, sigla, tipo, ativo).
PUT /cursos/:id. Atualiza nome, sigla, tipo, ativo.
DELETE /cursos/:id ou DELETE /cursos/:id/inativar. Se o curso já tiver UCs associadas, usar a flag inativar pois a remoção falha por causa da FK.

Anexo II — Concorrência e Controlo de Acessos Simultâneos

Este anexo aborda os mecanismos necessários para garantir a integridade dos dados e a previsibilidade do sistema em ambientes multiutilizador, onde várias operações podem ocorrer em simultâneo sobre a mesma informação (por exemplo, atribuição de docentes às UCs).

1. Conceito e importância

Num sistema multiutilizador, as condições de corrida (race conditions) podem gerar inconsistência de dados ou perda de informação. É, portanto, essencial controlar a concorrência para garantir integridade e previsibilidade.
Estas situações criam condições de corrida (race conditions), podendo levar a:

- -Inconsistência de dados (soma de horas incorreta ou duplicação de atribuições);
- -Leituras incorretas (um utilizador vê informação desatualizada);
- -Perdas de atualizações (o segundo sobrescreve o primeiro);
- -Bloqueios cruzados (deadlocks), se duas transações se esperarem mutuamente.

2. Tipos de anomalias concorrenciais

Tipo de anomalia	Exemplo no contexto da UC	Consequência
Dirty read	Ler uma atribuição ainda não confirmada (transação pendente).	Informação temporariamente inconsistente.
Lost update	Dois utilizadores atualizam as mesmas horas e um sobrescreve o outro.	Perda de dados.
Non-repeatable read	Um coordenador lê valores que mudam durante a operação.	Resultado imprevisível.
Phantom read	O número de docentes atribuídos muda durante uma consulta.	Relatórios inconsistentes.

3. Estratégias de controlo de concorrência

3.1 Na Base de Dados (controlo forte e transacional)
A base de dados relacional (PostgreSQL) deve assegurar a integridade das operações através de transações, bloqueios e níveis de isolamento.

Boas práticas:

- Usar transações (BEGIN, COMMIT, ROLLBACK) para garantir atomicidade.
- Ajustar o nível de isolamento consoante a criticidade:
    - READ COMMITTED – evita “dirty reads” (nível por defeito);
    - REPEATABLE READ – evita leituras não repetíveis;

  - SERIALIZABLE – garante execução como se fosse sequencial.
- Aplicar SELECT … FOR UPDATE para bloquear linhas críticas (ex.: UC a ser atualizada).
- Utilizar pg_advisory_xact_lock() para bloqueios lógicos por recurso (ex.: uma UC ou docente específico).
- Implementar triggers AFTER INSERT/UPDATE que recalculam totais e rejeitam transações que violem limites (ex.: horas excedidas).

3.2 Nos Serviços (REST / gRPC)
A camada de aplicação deve complementar a BD, gerindo concorrência lógica e política de negócio. Abordagens possíveis:

- Otimistic locking → cada registo tem um campo de versão; se outro o alterou entretanto, devolver 409 Conflict.
- Locks distribuídos (por ex. Redlock/Redis) para ambientes com múltiplas instâncias da API.
- Filas de tarefas (Kafka, Temporal) para serializar operações de atribuição.
- Retry/backoff automático em conflitos.
- Idempotência nas operações (repetições não devem criar duplicados).

3.3 No GraphQL
O GraphQL é predominantemente de leitura, mas deve minimizar impacto em consultas concorrentes através de:

- Caching e materialized views;
- Read replicas para separar leitura e escrita;
- Resolvers otimizados que consultam dados consistentes após commit.

4. Locais e mecanismos recomendados

Tipo de operação	**Onde aplicar controlo**	**Mecanismo sugerido**
Criação/atualização de atribuições	BD + Trigger + Lock lógico	SELECT … FOR UPDATE ou pg_advisory_xact_lock
Atualização concorrente da mesma UC/docente	Serviço (otimistic locking)	Campo version e código HTTP **409 Conflict**
Operações multi-serviço (REST ↔ gRPC)	Orquestrador / fila de tarefas	Execução sequencial
Consultas de leitura massiva	GraphQL	*Caching*, *read replicas*, *materialized views*

5. Boas práticas complementares

- Executar todas as operações críticas dentro de transações.
- Evitar triggers complexas — devem apenas validar e rejeitar.
- Controlar concorrência ao nível mínimo necessário (linha ou recurso, não tabela completa).
- Documentar todas as regras de bloqueio e isolamentos utilizados.
- Testar situações de concorrência (dois utilizadores simultâneos).

Síntese

A gestão correta da concorrência é essencial para garantir que, mesmo com vários utilizadores e microserviços em execução, a distribuição de serviço docente permanece consistente, previsível e auditável. O sistema deve, portanto, combinar mecanismos de controlo transacional na BD com validação lógica na camada de serviços.

Anexo III — Autenticação, Perfis e Permissões - Conceitos

1. Conceito geral

A autenticação e a gestão de permissões são mecanismos fundamentais para garantir a segurança e integridade do sistema de distribuição de serviço docente. Pretende-se assegurar que cada utilizador é devidamente identificado e que acede apenas aos recursos e operações que lhe são permitidos.

2. Autenticação

A autenticação confirma quem é o utilizador que tenta aceder ao sistema.

2.1 Métodos

- JWT (JSON Web Token):
  - Gerado após login (POST /login).
  - Inclui claims como id_user, perfil, exp (expiração).
  - Verificado em cada pedido com middleware.
- OAuth 2.0 / OpenID Connect (SSO):
  - Útil se houver integração com sistemas institucionais (ex.: IPSantarém).
- API Key:
  - Para integrações máquina-a-máquina (APIs públicas).

2.2 Boas práticas

- Usar TLS/HTTPS em todos os endpoints.
- Encriptar palavras-passe com bcrypt/argon2.
- Tokens curtos + refresh tokens seguros.
- Invalidar tokens revogados (lista negra).
- Registar tentativas de login e falhas.

3. Perfis de utilizador e escopo de permissões

Perfil	Descrição	Exemplos de operações permitidas
Administrador	Gestão global do sistema.	Criar/editar cursos, UCs, docentes, áreas e utilizadores.
Coordenador de curso	Responsável por um curso ou área científica.	Atribuir docentes às UCs do seu curso, consultar planos de estudo e validar cargas horárias.

Perfil	Descrição	Exemplos de operações permitidas
Docente	Utilizador individual com serviço atribuído.	Consultar o seu serviço e horas, atualizar dados pessoais e submeter CV.
Convidado / Público	Utilizador externo autenticado apenas para leitura.	Consultar informação pública (cursos e planos de estudo).

Os perfis e respetivos privilégios devem ser centralizados em tabela de perfis e permissões na base de dados, de modo a permitir fácil atualização.

4. Tipos de controlo de acesso

4.1 RBAC — Role-Based Access Control
Baseado em papéis fixos (Administrador, Coordenador, Docente…).
Mais simples de implementar, adequado a sistemas com hierarquias estáveis.

4.2 ABAC — Attribute-Based Access Control
Baseado em atributos do utilizador, da ação e do recurso.

Exemplo de regra ABAC:
"Permitir que um coordenador altere apenas UCs cujo id_curso pertence ao seu curso."
Mais flexível, adequado a contextos multi-curso ou multi-institucionais.

4.3 Híbrido (RBAC + ABAC)
Modelo recomendado: papéis base + atributos de contexto para granularidade fina.

5. Aplicação das regras de acesso por camada

Camada / tecnologia	Tipo de regra	Implementação
API REST (Next.js / Express)	Autenticação e RBAC básico	Middleware global que lê o token JWT, verifica o perfil e decide o acesso.
Serviços gRPC	Autorização interna entre microserviços	Inclusão de *metadata* com *service tokens* e validação da origem da chamada.
API GraphQL	Controlo fino por campo / *resolver*	Middleware de autorização (por exemplo, *graphql-shield*), verificando atributos e perfis.

Camada / tecnologia	Tipo de regra	Implementação
Base de dados	Integridade e separação de dados sensíveis	Controlo de privilégios SQL (GRANT / REVOKE) e *views* filtradas (WHERE id_doc = current_user_id()).
APIs públicas	Leitura anónima autenticada por chave	Chave API (*API Key*) com *rate limiting* e acesso apenas de leitura.

6. Boas práticas de design e implementação

- Separar autenticação (identificar) de autorização (decidir o que pode fazer).
- Centralizar lógica de verificação em middlewares reutilizáveis.
- Registar em logs todos os acessos críticos e falhas de permissão.
- Proteger endpoints sensíveis com dupla validação (token + perfil).
- Aplicar rate limiting e CORS controlado.
- Utilizar claims no JWT para evitar consultas repetidas à BD.
- Implementar testes automáticos de segurança (rotas e perfis).

7. Exemplos de aplicação prática

Endpoint	Requisitos de acesso	Justificação
POST /login	Público	Geração de token (autenticação inicial).
GET /docentes	Administrador, Coordenador	Gestão de recursos humanos e consulta de docentes.
GET /docentes/:id/servico	Docente (próprio) ou Coordenador	Visualização do serviço docente individual.
POST /atribuicoes	Coordenador, Administrador	Atribuição de docentes às UCs.
GET /api/public/cursos/:id/ucs	Convidado ou API Key	Consulta pública dos planos de estudo.

8. Localização recomendada das validações

Tipo de verificação	Camada ideal	Motivo
Autenticação de utilizadores (login, tokens)	Serviço REST	Verificação imediata no ponto de entrada.
Permissões por papel (RBAC)	Middleware REST / gRPC	Execução rápida e coesa no fluxo de pedidos.

Tipo de verificação	Camada ideal	Motivo
Regras contextuais (ABAC, curso, área, docente)	*Resolver* GraphQL ou serviço gRPC	Necessitam de contexto e dados adicionais.
Filtragem de dados por utilizador	Base de Dados (views ou RLS)	Garante isolamento e confidencialidade ao nível dos dados.

Nota: O Row-Level Security (RLS) define políticas que determinam quais as linhas (rows) de uma tabela que um determinado utilizador pode ver, inserir, atualizar ou apagar.

9. Extensão e integração

Os tokens emitidos pela API poderão ser utilizados por outras aplicações internas (ex.: portal do docente, dashboards).
A API deve fornecer endpoint de verificação de token (/auth/verify) para integração.
Permitir integração com serviços externos via OAuth 2.0 (por ex., Google, Microsoft, IPS SSO).

10. Auditoria e conformidade

Manter registo detalhado de logins, acessos e alterações (timestamp, IP, utilizador).
Assegurar conformidade com RGPD: apenas os dados necessários devem ser armazenados e expostos.
Permitir a revogação imediata de tokens em caso de compromisso.

Síntese

O sistema deve implementar uma camada de segurança transversal, garantindo que a autenticação, as permissões e as APIs públicas seguem princípios de segurança mínima, rastreabilidade e controlo centralizado.
As regras de acesso devem ser aplicadas em múltiplas camadas (serviço, BD, GraphQL), evitando depender de um único ponto de validação.

Anexo IV — Glossário técnico e siglas

Siglas e conceitos principais

Sigla / termo	Significado	Explicação prática
JWT	*JSON Web Token*	Formato padrão (RFC 7519) de token digital assinado usado para autenticação "stateless". O servidor cria o token com informações (claims) e assina-o; o cliente envia-o em cada pedido (Authorization: Bearer <token>).
Stateless	Sem estado persistente	Significa que o servidor não guarda informação da sessão do utilizador — valida tudo apenas com o conteúdo assinado do JWT. (O estado, quando necessário, é gerido na BD através de sessions.)
RBAC	*Role-Based Access Control*	Modelo de controlo de acesso baseado em papéis. Cada utilizador tem um **perfil (role)** que define o que pode fazer (ex.: Administrador, Coordenador, Docente, Convidado).
API	*Application Programming Interface*	Conjunto de endpoints (funções disponíveis via HTTP ou gRPC) que permitem interagir com o sistema.
API Key	Chave de aplicação	Código secreto (normalmente em cabeçalho HTTP) usado para aceder a APIs públicas ou integrações sem autenticação de utilizador.
CORS	*Cross-Origin Resource Sharing*	Política de segurança de browsers que define quais domínios podem comunicar com a API (ex.: permitir apenas o domínio do frontend Flutter).
HTTPS / TLS	*HyperText Transfer Protocol Secure* / *Transport Layer Security*	Protocolo que cifra a comunicação entre cliente e servidor. TLS é a tecnologia usada no HTTPS para garantir confidencialidade e integridade.
bcrypt / argon2	Algoritmos de hashing de palavras-passe	Funções de hashing seguras usadas para guardar senhas cifradas na BD (nunca se guarda a senha em texto).
BD	Base de Dados	Repositório central onde são armazenados utilizadores, sessões, tokens, e dados académicos.
UUID	*Universally Unique Identifier*	Identificador único (ex.: 550e8400-e29b-41d4-a716-446655440000) usado como chave primária.

Sigla / termo	Significado	Explicação prática
PK / FK	*Primary Key* / *Foreign Key*	Chaves primária e estrangeira — identificam de forma única uma linha e criam relações entre tabelas.
TIMESTAMPTZ	*Timestamp with time zone*	Tipo de dados PostgreSQL que guarda datas/horas com fuso horário.
Hash	Resumo criptográfico	Valor gerado a partir de um texto (ex.: token) que não pode ser revertido — usado para verificar autenticidade sem guardar o valor real.
Seed	Dados de inicialização	Conjunto de registos de exemplo (ex.: utilizadores de teste) inseridos automaticamente para facilitar testes.

Claims e campos do token

Campo	Significado	Utilização
sub	*Subject*	ID do utilizador (user_id). Identifica quem é o dono do token.
sid	*Session ID*	ID da sessão atual (linha da tabela sessions), usado para revogação por dispositivo.
tv	*Token Version*	Versão do token associada ao utilizador. Sempre que se faz logout global (logout-all), incrementa-se token_version → todos os tokens anteriores tornam-se inválidos.
role	Papel do utilizador	Ex.: ADMIN, COORDINATOR, TEACHER, GUEST. Usado no RBAC.
exp	*Expiration time*	Data/hora de expiração do token JWT (acesso válido por 10–15 min).
courseIds	IDs de cursos do utilizador	Claims adicionais opcionais: cursos sob responsabilidade do coordenador ou do docente.

Estrutura das tabelas de base de dados

Tabela	Propósito	Campos principais
users	Armazena credenciais e papéis dos utilizadores.	id, email, password_hash, role, token_version, created_at, updated_at
sessions	Representa cada sessão (dispositivo/login). Permite revogação específica.	id (sid), user_id, family_id, revoked_at, expires_at, created_at
refresh_tokens	Tokens de renovação de acesso, guardados com hash.	id, session_id, token_hash, is_revoked, expires_at

Conceitos de ciclo de autenticação

Termo	Explicação prática
Login	O utilizador envia email + password. O servidor valida, cria uma *session* e devolve access_token + refresh_token.
Access token	JWT curto (10–15 min) usado para autenticar cada pedido. Contém sid e tv.
Refresh token	String longa (7–30 dias) usada para gerar novo *access token* sem refazer login. Guardado com **hash** e **revogável**.
Logout	Revoga apenas a sessão atual (revoked_at = now()).
Logout-all	Incrementa token_version, invalidando todos os tokens do utilizador.
Revogação imediata	Qualquer pedido com sid revogado → **401 Unauthorized**.
Rotação de refresh	Sempre que se usa o refresh, o antigo é revogado e cria-se um novo na mesma "família".
Auditoria	Registo de eventos: login, refresh, logout, erros 401/403, revogações.

Siglas e códigos HTTP usados nos testes

Código	Nome	Significado no contexto do projeto
200 OK	Sucesso	Operação executada corretamente.
401 Unauthorized	Não autenticado	Token ausente, expirado ou sessão revogada.
403 Forbidden	Proibido	Token válido, mas papel (RBAC) insuficiente.
409 Conflict	Conflito lógico	Ex.: atualização concorrente detetada.
429 Too Many Requests	Excesso de chamadas	*Rate limiting* em endpoints públicos.

Anexo V —Desenvolvimento em Flutter

1. Introdução ao Flutter

Flutter é um framework open-source da Google para criar aplicações multiplataforma (Android, iOS, Web, Windows, macOS e Linux) a partir de uma única base de código, escrita em Dart.

1.1 Características principais

- Renderização nativa (usa o motor gráfico Skia, não componentes WebView);
- Hot Reload / Hot Restart — recarregamento instantâneo durante o desenvolvimento;
- Biblioteca de widgets consistente e extensível;
- Excelente integração com REST APIs, GraphQL e WebSockets;
- Suporte a Material Design 3 e Cupertino (iOS).

1.2 Estrutura de um projeto Flutter

```
my_app/
├── android/          # Configurações Android (Gradle)
├── ios/              # Configurações iOS (Xcode)
├── lib/              # Código Dart principal
│   ├── main.dart     # Ponto de entrada
│   ├── screens/      # Interfaces de ecrã
│   ├── widgets/      # Componentes reutilizáveis
│   ├── services/     # Acesso a dados / API
│   └── models/       # Estruturas de dados
├── pubspec.yaml      # Dependências e recursos
└── test/             # Testes automatizados
```

2. Instalação e configuração

2.1 Requisitos

Flutter SDK → flutter.dev
Android Studio ou VS Code com o plugin Flutter/Dart
Emulador Android ou dispositivo físico

2.2 Verificação

flutter doctor
Verifica se todas as ferramentas estão corretamente instaladas.

2.3 Criação de um novo projeto

flutter create my_app

```
cd my_app
flutter run
```

O comando abre a aplicação padrão “Counter App”.

3. Fundamentos de Dart

3.1 Sintaxe básica

```
void main() {
  var nome = 'Maria';
  int idade = 30;
  print('Olá $nome, idade $idade');
}
```

Tipagem forte, mas com inference (var / final);
Suporte a async/await, classes, extensions, null safety.

3.2 Estruturas comuns

```
class Pessoa {
  final String nome;
  int idade;
  Pessoa(this.nome, this.idade);
  void aniversario() => idade++;
}
```

4. Conceitos essenciais de Flutter

Conceito	Descrição
Widget	Unidade básica da UI. Tudo em Flutter é um widget.
StatelessWidget	Interface fixa (não altera estado).
StatefulWidget	Interface dinâmica, pode atualizar estado.
BuildContext	Fornece contexto hierárquico para widgets.
MaterialApp	Raiz da app, aplica tema e navegação.
Scaffold	Estrutura base (AppBar, Body, FAB, etc.).

4.1 Exemplo simples

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      title: 'Exemplo Flutter',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: const HomeScreen(),
    );
  }
}

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});
  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  int contador = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Contador')),
      body: Center(child: Text('Valor: $contador')),
      floatingActionButton: FloatingActionButton(
        onPressed: () => setState(() => contador++),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

5. Navegação e rotas

5.1 Navegação simples

```
Navigator.push(context,
  MaterialPageRoute(builder: (context) => const OutraPagina()));
```

5.2 Rotas nomeadas

```
MaterialApp(
  routes: {
    '/': (context) => const Home(),
    '/login': (context) => const Login(),
  },
);
Navigator.pushNamed(context, '/login');
```

6. Consumo de APIs REST

6.1 Dependência

```
dependencies:
  dio: ^5.3.0
```

6.2 Exemplo de uso

```
import 'package:dio/dio.dart';
class ApiService {
  final Dio dio = Dio(BaseOptions(baseUrl: 'https://api.exemplo.pt'));
  Future<List<dynamic>> getDocentes() async {
    final response = await dio.get('/docentes');
    return response.data;
  }
}
```

6.3 Integração com UI

```
FutureBuilder(
  future: ApiService().getDocentes(),
  builder: (context, snapshot) {
    if (!snapshot.hasData) return const CircularProgressIndicator();
    final docentes = snapshot.data!;
    return ListView.builder(
      itemCount: docentes.length,
      itemBuilder: (_, i) => ListTile(title: Text(docentes[i]['nome'])),
    );
  },
)
```

7. Armazenamento local

Recurso	Biblioteca	Uso típico
SharedPreferences	shared_preferences	Armazenar preferências simples (tema, idioma, configurações do utilizador).
Secure Storage	flutter_secure_storage	Guardar tokens e dados sensíveis de forma encriptada e segura.
SQLite	sqflite	Gerir bases de dados locais estruturadas e persistentes.

8. Gestão de estado

Abordagem	Características
setState()	Simples, nativo, indicado para aplicações pequenas.

Abordagem	Características
Provider	Solução oficial do Flutter; oferece boa separação entre lógica e interface.
Riverpod	Evolução do Provider, mais modular e escalável.
Bloc / Cubit	Arquitetura reativa robusta, ideal para aplicações grandes e colaborativas.

Exemplo simples com Provider:

```
class Contador extends ChangeNotifier {
  int valor = 0;
  void inc() { valor++; notifyListeners(); }
}
```

9. Autenticação (conceitos gerais)

- Comunicação segura via HTTPS.
- Guardar tokens em flutter_secure_storage.
- Usar interceptors para renovar access tokens.
- Logout apaga todos os dados locais.

10. Testes e depuração

```
flutter test              # testes unitários
flutter run --verbose     # log detalhado
flutter analyze           # lint e boas práticas
```

Usar flutter devtools (profiling, memória, performance).

11. Empacotamento e publicação

11.1 Android

```
flutter build apk --release
```

O ficheiro estará em build/app/outputs/flutter-apk/app-release.apk.

11.2 iOS

```
flutter build ipa --release
```

Publicar via Xcode ou TestFlight.

11.3 Web

```
flutter build web
```

Gera build/web/ pronto para hospedar.

12. Boas práticas gerais

Tema	Boas práticas
Estrutura de pastas	Separar UI, lógica e dados.
Responsividade	Usar LayoutBuilder e MediaQuery.
Acessibilidade	Usar *semantics*, contraste adequado e tamanhos escaláveis.
Internacionalização	Utilizar flutter_localizations e ficheiros .arb para idiomas.
Performance	Evitar *rebuilds* desnecessários e usar const widgets sempre que possível.
Segurança	Nunca expor segredos (como chaves API) no código; usar sempre HTTPS.

13. Recursos recomendados

Documentação oficial: https://docs.flutter.dev
Curso interativo (DartPad): https://dartpad.dev
Biblioteca de widgets: https://fluttergems.dev
Design system: https://m3.material.io
Comunidade: Flutter Community Medium

Anexo VI —Tema 2 - Planeamento e Gestão de Horários

Descrição do problema
O planeamento de horários é um processo complexo e crítico na gestão académica. Implica conciliar **disponibilidades de docentes**, **características das unidades curriculares**, **recursos físicos limitados** (salas e laboratórios) e **restrições institucionais** (cargas horárias, tipos de sala, turmas, cursos e lotações).
O objetivo deste tema é **conceber e desenvolver uma solução integrada** que permita **gerar, visualizar e gerir horários** de aulas, garantindo que:

- cada **docente** tem as suas aulas distribuídas sem sobreposição;
- cada **UC/turma** é alocada a uma **sala adequada** ao seu tipo (teórica, prática, laboratório, etc.);
- a **lotação** da sala é suficiente para a turma;
- são respeitadas as **restrições de tempo** (dias, blocos horários, indisponibilidades);
- são evitados **conflitos e duplicações**;
- a gestão pode ser feita de forma colaborativa por administradores e coordenadores de curso.

A aplicação deve ainda permitir **geração automática ou assistida** dos horários, **visualização gráfica** em grelha semanal e **exportação de relatórios** por docente, curso ou sala.
O **sistema de autenticação e perfis (RBAC)** é o mesmo do **Tema 1**, garantindo continuidade e coerência entre os módulos da plataforma de gestão académica.

O Tema 2 reutiliza (do tema 1):

- a estrutura de **utilizadores e autenticação** (JWT + RBAC);
- os dados base de **docentes** e **unidades curriculares**;
- o mesmo **modelo de permissão** (Administrador, Coordenador, Docente, Convidado);
- o **sistema de autenticação e perfis (RBAC)**

A diferença central está no domínio funcional: **alocação de aulas, salas e tempos**, em vez da distribuição de serviço docente.

Domínio do problema e entidades principais

Entidade	Descrição resumida	Relações principais
docente	Dados do professor e respetivas restrições de horário	Relação com aula e uc
uc	Unidade curricular (herdada do Tema 1)	Ligada a turma e aula
turma	Grupo de estudantes de uma UC	Pertence a um curso e tem várias aulas
sala	Espaço físico (sala ou laboratório)	Tem tipo_sala, lotacao, e aulas atribuídas
tipo_sala	Categoria de sala (teórica, prática, laboratório, auditório)	FK em sala

Entidade	Descrição resumida	Relações principais
janela_horaria	Bloco de tempo (dia_semana, início, fim, duração)	Usado em aula
aula	Instância de uma aula agendada (turma + docente + sala + janela horária)	Central na geração de horários

Situação a resolver
Atualmente, o processo de elaboração de horários é manual, exigindo várias iterações e trocas de informação entre coordenações de curso e serviços administrativos.

O sistema proposto deve:

- **automatizar a alocação de aulas** segundo regras e restrições definidas;
- **detetar e prevenir conflitos** (de docente, de sala e de sobreposição de horários);
- **gerar propostas de horários** otimizadas;
- permitir **ajustes manuais** e **validação de conflitos**;
- permitir **consulta pública** dos horários finais (por curso, sala ou docente).

A aplicação deve permitir:
- Gestão de **salas** e respetivas **características** (tipo, lotação, edifício, campus).
- Gestão de **turmas** e **janelas horárias disponíveis**.
- Atribuição de **aulas** a docentes, turmas e salas.
- Verificação automática de **regras de integridade e coerência**:
  - sem sobreposição de docente;
  - sem sobreposição de sala;
  - tipo de sala compatível;
  - lotação adequada;
  - carga horária semanal respeitada.
- Registo e consideração de **preferências de docentes** (dias, horários, pausas).
- Geração automática do horário semanal (heurística construtiva ou algoritmo simples de otimização).
- **Visualização do horário** em grelha semanal (por docente, sala, curso).
- **Exportação** de relatórios e horários em PDF/CSV.

Requisitos técnicos e metodológicos
O desenvolvimento segue as **mesmas etapas estruturais do Tema 1**, adaptadas ao novo contexto:

Etapa	Descrição resumida (equivalente ao Tema 1)
Etapa 0	Levantamento de requisitos e modelo de dados (integração com docente e uc).
Etapa 1	Setup da infraestrutura e seed de dados (salas, tipos de sala, janelas).
Etapa 2	Definição das regras de integridade e triggers (restrições hard e soft).
Etapa 3	Implementação das APIs REST/GraphQL (/salas, /horarios, /aulas, /gerar) e integração com serviço gRPC.

Etapa	Descrição resumida (equivalente ao Tema 1)
Etapa 4	Desenvolvimento do **motor de geração de horários** (heurístico/otimização), exposto via gRPC.
Etapa 5	Desenvolvimento da interface Flutter Web (visualização em grelha, conflitos, exportação).
Etapa 6	Concorrência e edição segura (locks, optimistic locking).
Etapa 7	Relatórios e visualização (ocupação de salas, horários por docente/curso).
Etapa 8	Relatório técnico final e demonstração integrada.

Regras essenciais (hard constraints)

Regra	Descrição	Implementação sugerida
Conflito de docente	Um docente não pode ter duas aulas no mesmo horário	EXCLUSION (docente_id, janela_id)
Conflito de sala	Uma sala não pode ter duas aulas no mesmo horário	EXCLUSION (sala_id, janela_id)
Tipo de sala	O tipo de aula deve corresponder ao tipo de sala	Trigger check_tipo_sala()
Lotação	A lotação da sala deve ser ≥ nº de estudantes da turma	Trigger check_lotacao()
Carga horária	As horas atribuídas por UC devem corresponder ao previsto	View ou trigger valida_carga_uc()

Restrições e preferências (soft constraints)

- Preferências de docentes por dia/horário (opcionais, peso configurável).
- Minimização de "buracos" entre aulas do mesmo docente.
- Distribuição equilibrada ao longo da semana.
- Evitar aulas noturnas ou horários consecutivos excessivos.

Estas restrições devem ser **consideradas no algoritmo de geração** (Etapa 4) e refletidas nos relatórios finais (Etapa 7).

Integração e autenticação

- Reutiliza a mesma **autenticação JWT + RBAC** do Tema 1.
- Papéis e permissões mantêm-se:
  - **Administrador:** gestão global, criação de salas e janelas.
  - **Coordenador:** atribuição de aulas e validação do horário do curso.
  - **Docente:** consulta do seu horário e inserção de preferências.
  - **Convidado:** consulta pública dos horários aprovados.

Demonstração final

A demonstração deverá evidenciar:

1. Inserção/edição de salas, docentes e UCs.
2. Geração automática do horário.
3. Deteção e correção de conflitos.
4. Visualização em grelha semanal (por docente e por sala).

5. Exportação de relatório em PDF.
6. Consulta pública via API (ex.: /api/public/horario/curso/:id).

Avaliação (segue as regras do Tema 1)

(*) Nota: Ver bónus no anexo VIII

Extensões opcionais

- Preferências de docentes com pesos dinâmicos.
- Algoritmo evolutivo (genético) para otimização.
- Integração com calendários (ICS export).
- Painel de monitorização em Grafana (ocupação de salas).

Anexo VII —Tema 3 - Gestão da Bolsa de Formadores e Histórico de Serviço Docente

Descrição do problema
As instituições de ensino superior necessitam frequentemente de recorrer a formadores externos — docentes convidados ou especialistas — para assegurar parte do serviço letivo. Atualmente, a recolha e atualização destes dados é manual e dispersa, dificultando o planeamento e a análise do histórico de colaboração.
O objetivo deste tema é conceber e desenvolver uma aplicação integrada que permita:

- registar e gerir uma bolsa de formadores, ativos e potenciais;
- consultar o histórico de serviço docente prestado por cada formador em anos letivos anteriores;
- permitir que os formadores atualizem o seu perfil e CV;
- apoiar a seleção e atribuição futura de serviço docente, integrando-se com os módulos anteriores (Tema 1 e Tema 2).

Integração com os Temas 1 e 2
O Tema 3 é o terceiro módulo da mesma plataforma académica e:

- reutiliza a autenticação e RBAC definidos no Tema 1;
- utiliza os mesmos dados base de docentes, UCs e cursos;
- relaciona-se com os horários e atribuições do Tema 2;
- introduz novas entidades e funcionalidades orientadas à gestão de perfis e histórico.

Domínio do problema e entidades principais

Entidade	**Descrição resumida**	**Relações principais**
formador	Registo individual de cada docente/formador externo (ativo ou potencial).	Ligação a historico_servico, cv, area_cientifica.
area_cientifica	Classificação por área/disciplina (Ex.: Matemática, Informática).	FK em formador e em uc.
cv_formador	Dados curriculares (grau académico, experiência, publicações, competências).	Relacionado 1:1 com formador.
historico_servico	Histórico de serviço docente anterior (UCs, horas, cursos, anos).	FK para formador, uc, curso.
documento_cv	Ficheiros anexados (PDFs, certificados).	FK para formador.
avaliacao_formador *(opcional)*	Avaliação qualitativa do desempenho.	FK para formador e ano_letivo.

A aplicação a desenvolver deverá permitir:

- registar novos formadores (inserção manual ou auto-registo);
- manter um histórico estruturado do serviço docente anterior (UCs, horas, avaliações);

- possibilitar que o formador atualize os seus dados e CV diretamente (autenticação individual);
- permitir que coordenadores e administradores filtrem formadores por área, grau académico ou disponibilidade;
- gerar relatórios de carga letiva passada e previsões de alocação.

Âmbito funcional

A aplicação deve suportar as seguintes operações principais:

1. Gestão de Formadores
    - Criar, editar e desativar registos de formadores;
    - Atribuir áreas científicas, graus e disponibilidade;
    - Upload e gestão de CVs e documentos.
2. Histórico de Serviço Docente
    - Registar anos letivos, UCs e horas atribuídas;
    - Consultar histórico por formador, curso ou área;
    - Importar ou sincronizar dados do Tema 1 (atribuições anteriores).
3. Consulta e Filtragem
    - Filtrar por grau, área, experiência, avaliações ou disponibilidade;
    - Exportar listagens em PDF/CSV.
4. Atualização e Acesso do Formador
    - Login individual (perfil "Formador" no RBAC);
    - Edição do seu próprio CV e dados pessoais;
    - Consulta do seu histórico e estatísticas de serviço.
5. Relatórios e Apoio à Decisão
    - Relatório de carga letiva histórica por área/curso;
    - Distribuição de graus académicos na bolsa;
    - Taxa de renovação de formadores (novos vs. antigos).

Requisitos técnicos e metodológicos

Tal como nos Temas 1 e 2, o desenvolvimento segue as mesmas etapas estruturais:

Etapa	**Descrição resumida (equivalente ao Tema 1)**
Etapa 0	Levantamento de requisitos e modelação E-R (formadores, histórico, CVs).
Etapa 1	Setup da infraestrutura (Docker, PostgreSQL, Next.js, GraphQL) e seed de dados iniciais (áreas, graus, formadores).
Etapa 2	Implementação de regras de integridade e validações (unicidade de email, relação formador–área, coerência com anos letivos).
Etapa 3	Gateway REST/GraphQL: /formadores, /cv, /historico, /documentos (incluir chamadas gRPC a serviços internos).
Etapa 4	**Microserviços gRPC**: CVService (parsing de CVs), MatchingService (recomendações/score), ImportSyncService (histórico do Tema 1)
Etapa 5	Interface Flutter Web: listagem e perfil de formador, formulário de CV, upload de documentos, histórico interativo.
Etapa 6	Concorrência e edição segura (locks em CV, uploads, atualizações).

Laboratório de Desenvolvimento de Software
2025_2026

Etapa	Descrição resumida (equivalente ao Tema 1)
Etapa 7	Relatórios: carga letiva passada, distribuição de áreas e graus, desempenho dos formadores.
Etapa 8	Relatório técnico e demonstração final.

Regras essenciais (hard constraints)

Regra	Descrição	Implementação sugerida
Identidade única	Email e NIF do formador são únicos.	Constraint UNIQUE(email, nif)
Área obrigatória	Cada formador deve pertencer a pelo menos uma área científica.	FK + Trigger de validação
Histórico coerente	Anos letivos e UCs atribuídas devem existir e corresponder a períodos válidos.	FK para uc e ano_letivo
Upload seguro	Apenas ficheiros PDF permitidos, tamanho máximo definido.	Validação no serviço e no front-end
Integridade histórica	Não apagar registos de serviço anteriores, apenas marcar como inativos.	Campo ativo ou deleted_at

Perfis e permissões (RBAC)

Perfil	Responsabilidades no Tema 3
Administrador	Gestão global da bolsa, criação e validação de registos.
Coordenador	Consulta e seleção de formadores por área/curso.
Formador	Atualização dos seus próprios dados e CV; consulta do histórico.
Convidado	Consulta pública limitada (ex.: lista parcial de áreas).

Exemplo de funcionalidades do frontend (Flutter)

- Página de listagem de formadores com filtros (área, grau, disponibilidade).
- Página de perfil detalhado (CV, histórico, contacto).
- Formulário de edição de CV (experiência, publicações, uploads).
- Visualização do histórico de serviço docente (ano, UC, curso, horas).
- Exportação de relatórios PDF e listagens CSV.

Avaliação (segue as regras do Tema 1)

(*) Nota: Ver bónus no anexo VIII

Anexo VIII - Interoperabilidade e APIs Públicas

Com vista à integração entre os diferentes módulos desenvolvidos no âmbito dos Temas 1, 2 e 3, cada solução poderá disponibilizar APIs públicas que permitam a partilha e reutilização de dados essenciais entre sistemas.

Estas APIs destinam-se a facilitar a interoperabilidade, evitando duplicação de informação e permitindo que os restantes módulos possam consultar dados estruturados, como:

- Tema 1 — Distribuição de Serviço Docente: lista de docentes, unidades curriculares e atribuições anuais;
- Tema 2 — Planeamento de Horários: horários por curso, sala ou docente;
- Tema 3 — Bolsa de Formadores: formadores disponíveis, áreas científicas e histórico de serviço.

A disponibilização destas APIs deve respeitar um conjunto de regras mínimas de consistência e segurança:

1. Acesso controlado:
    - As APIs públicas destinam-se apenas à leitura de dados não sensíveis;
    - O acesso pode exigir uma API Key e estar sujeito a limites de utilização (rate limiting).
2. Formato e padrões:
    - Respostas no formato JSON;
    - Endpoints REST padronizados (ex.: /api/public/docentes, /api/public/horario/curso/:id);
    - Utilização de datas ISO-8601 e nomenclatura coerente (snake_case em atributos).
3. Segurança e privacidade:
    - Nenhum dado pessoal (e.g., email, telefone, NIF) deve ser exposto;
    - Apenas informação agregada, pública ou institucional pode ser publicada.
4. Versão e documentação:
    - As APIs devem identificar a versão (ex.: /api/public/v1/...);
    - Devem incluir documentação básica (OpenAPI/Swagger ou ficheiro Markdown com exemplos).
5. Integração entre temas:
    - O Tema 2 poderá consultar docentes e UCs do Tema 1 para gerar horários;
    - O Tema 3 poderá aceder ao histórico de serviço do Tema 1 e às áreas de docência do Tema 2;
    - Todos os módulos podem expor endpoints públicos para facilitar relatórios e estatísticas conjuntas.

A criação destas APIs é opcional, mas será valorizada como bónus com um peso de até **2 valores** (10%) na avaliação.