ADR 001: Arquitetura de Segurança, Observabilidade e Resiliência – CashFlow API

- **Status:** Proposto
- **Data:** 08 de Fevereiro de 2026
- **Contexto:** Necessidade de implantar a CashFlow API num cluster Kubernetes atendendo a requisitos rigorosos de conformidade bancária e estabilidade.

Critérios para Aceitação

Para que esta ADR passe do estado "Proposto" para "Aceite", os seguintes critérios precisam ser formalmente verificados e aprovados pela equipa de Arquitetura/Segurança, conforme definido no documento:

Validação Técnica: Confirmação de que a arquitetura funciona corretamente nos ambientes DEV ou QA.
Conformidade do CI/CD: Verificação de que o *pipeline* CI/CD cumpre os requisitos de segurança e assinatura de imagem.
- Scan de imagem sem vulnerabilidades Critical ou High.
- Assinatura da imagem com Cosign.

Implementação no Cluster: Garantir que o *deploy* utiliza a **Gateway API** e certificados TLS automáticos
Monitorização: Assegurar que o *endpoint* `/metrics` está a expor dados e que as métricas estão a ser recolhidas pelo **ServiceMonitor.**

SEÇÃO 1: Segurança da Cadeia de Suprimentos

1.1 Imagem Base: Distroless

- **Decisão:** Utilizaremos imagens **Distroless** (como google distroless).
- **Justificação:** Ao contrário do Ubuntu ou até do Alpine, as imagens Distroless contêm apenas a aplicação e as suas dependências de runtime. Não possuem gestores de pacotes (apt, apk) nem shells (bash, sh), o que reduz drasticamente a superfície de ataque e elimina a maioria das CVEs comuns do SO.
  Alternativa: Podem ser utilizadas imagens **Wolfi**, fornecidas pela **Chainguard**, uma distribuição *distroless-like* focada em segurança e *supply chain integrity*, que oferece **imagens com ausência de vulnerabilidades conhecidas acima de severidades acordadas contratualmente**, mediante licenciamento.

1.2 Ferramentas de Pipeline (CI/CD): Trivy

- **Decisão:** Implementação do **Trivy** no pipeline de CI para escaneamento de imagens .
- **Justificação:** O **Trivy** é um scanner de código aberto abrangente que detecta além de vulnerabilidades de images (como o Docker Scout), verifica repositórios de código, IaC (Terraform, etc.) e Kubernetes.
  A cada nova versão da API a imagem será sempre escaneada e validada a ausência de vulnerabilidades acima de um determinado severity. Caso encontre uma nova vulnerabilidade a equipa será notificada de imediato.

1.3 Integridade da Imagem: Cosign (Sigstore)

- **Decisão:** Utilizar o **Cosign** para assinar as imagens após o build.
- **Justificação:** Garante que apenas imagens assinadas pela nossa chave privada sejam executadas no cluster, prevenindo ataques de *man-in-the-middle* no registo de imagens. No cluster, usar-se-á um *Admission Controller* (como Kyverno ou Policy Agent) para verificar a assinatura.

SEÇÃO 2: Estratégia de Acesso (Network & TLS)

2.1 Exposição do Serviço: Gateway API com Nginx Fabric

- **Decisão:** Utilizar o **Gateway API com Nginx Gateway Fabric** em conjunto com um Service do tipo `ClusterIP`.
- **Justificação:** O Ingress encontra-se em modo de manutenção, sendo a Gateway API o modelo recomendado para novas implementações. Usar o Nginx Gateway Fabric é uma escolha lógica para quem já usava o Nginx Ingress Controller.
  O gateway Nginx Gateway Fabric será o **entrypoint** do tráfego externo e encaminhará as requisições para os services da nossa API através configurações centralizadas de rotas. Ele permite ainda gerir certificados TLS num único ponto de entrada (Load Balancer único), otimizando custos e centralizando o controlo de tráfego L7 (capacidade de **inspecionar, decidir e manipular o tráfego com base no conteúdo da aplicação**, e não apenas em IPs, portas ou protocolos básicos).

2.2 Gestão de Certificados: Cert-Manager + Let's Encrypt

- **Decisão:** Instalação do **Cert-Manager**.

- **Justificação:** Automatiza o ciclo de vida dos certificados (emissão, renovação e aplicação no Secret do K8s) via protocolo ACME, garantindo HTTPS sem intervenção manual.

2.3 Objeto de Roteamento: HTTP Route

- **Decisão:** Criação de um HTTP Route.
- **Justificação**: É neste objeto que definiremos o host (ex: api.cashflow.com) e as regras de path para direcionar o tráfego para o Service da aplicação.

SEÇÃO 3: Observabilidade Total

3.1 Descoberta de Métricas: ServiceMonitor

- **Decisão:** Instalar Kube-Prometheus e criar CRDs do tipo **ServiceMonitor**.
- **Justificação:** A instalação do **kube-prometheus** no cluster Kubernetes introduz um conjunto abrangente de **componentes de observabilidade**, incluindo a recolha, armazenamento e visualização de métricas, bem como uma coleção extensiva de **dashboards pré-configurados** para monitorização do estado e desempenho do cluster.

  A monitorização da API será realizada através da definição de um **Custom Resource Definition (CRD) do tipo `ServiceMonitor`**, disponibilizado pelo **Prometheus Operator**.

  O recurso `ServiceMonitor` estabelecerá, de forma declarativa, os critérios de **descoberta automática de Services** e os parâmetros de **scraping de métricas** pelo Prometheus, incluindo o endpoint `/metrics`, portas expostas, seletores de labels e intervalos de recolha, garantindo a integração consistente da aplicação no sistema de monitorização do cluster.

3.2 Alertas: Alertmanager

- **Decisão:** Configuração de regras de alerta no **Alertmanager**.
- **Justificação:** O AlertManager será o componente responsável por agrupar, silenciar e enviar notificações (Slack, Teams, E-mail, etc..) caso as métricas recolhidas pelo Prometheus indiquem alguma falha no cluster ou na nossa API.

SEÇÃO 4: Resiliência e Disponibilidade

4.1 Gestão de Memória: Resources Requests & Limits

- **Decisão:** Definir **`limits`** de memória superiores ao consumo de pico inicial e **`requests`** que garantam espaço inicial no nó.
- **Justificação:** Configurar corretamente o **`limits.memory`** evita que o container sofra **OOMKill** se houver um pico previsível, enquanto o **`requests.memory`** garante que o Scheduler coloque o Pod num nó com recursos suficientes.

4.2 Probes de Saúde: Startup Probe

- **Decisão:** Implementar uma **Startup Probe** com um parâmetro `failureThreshold` adequado ao startup da aplicação (ex: 35 - 45s), seguida de uma **Readiness Probe**.

- **Justificação:** A **Startup Probe** desativa as outras probes até que a aplicação suba. Isso evita que a **Liveness Probe** reinicie o container prematuramente ou que a **Readiness Probe** falhe enquanto a conexão com a base de dados está a ser estabelecida nos primeiros 30 segundos.

Consequências

- **Benefícios:**
    - Alta segurança (imagens limpas e assinadas).
    - Automação total de TLS.
    - Observabilidade nativa.
    - Gestão de recursos adequados.
    - Alinhamento com requisitos de conformidade e auditoria.

- **Trade-Offs / Riscos:**
    - Maior complexidade na configuração inicial do pipeline de CI/CD.
    - Necessidade de gerir as chaves do Cosign.
    - Maior complexidade na configuração inicial dos manifestos kubernetes.