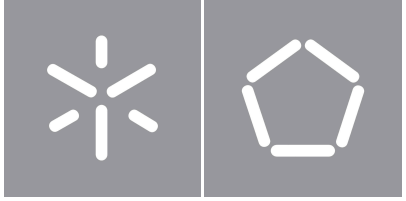
Universidade do Minho
Escola de Engenharia

Carolina Gil Afonso Santejo

Primavera: Microsserviço de Reporting

dezembro 2023



Universidade do Minho
Escola de Engenharia

Carolina Gil Afonso Santejo

Primavera: Microsserviço de Reporting

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de
Professor António Manuel Nestor Ribeiro

dezembro 2023

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:

CC BY

https://creativecommons.org/licenses/by/4.0/

Agradecimentos

Com a conclusão desta dissertação, gostaria de expressar a minha gratidão e reconhecimento a todos aqueles que desempenharam um papel significativo e que tornaram possível a sua realização.

Em primeiro lugar, quero agradecer ao meu supervisor Miguel Dias e ao Michael Pinheiro, que em conjunto me acompanharam durante os 9 meses de estágio na Cegid Primavera, e por todo o apoio e conhecimento que me transmitiram para que fosse possível concretizar este projeto.

Quero agradecer ao professor António Nestor Ribeiro pela sua orientação na realização da presente dissertação e por toda a ajuda no processo de desenvolvimento e escrita.

Um agradecimento aos meus amigos, Luísa, Rita, Filipa, Raquel e Luís, que estiveram ao meu lado durante os últimos cinco anos de curso, por todos os momentos passados e apoio incondicional.

Por fim, quero agradecer à minha família, por todo o apoio e dedicação. Um agradecimento especial aos meus pais e ao meu irmão, porque tornaram possível a concretização do meu desenvolvimento profissional e me deram suporte para enfrentar todos os desafios.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, dezembro 2023

Carolina Gil Afonso Santejo

Resumo

A Cegid Primavera tem vindo a desenvolver a sua *cloud* de microsserviços para facilitar a integração dos seus vários produtos e dos seus produtos com sistemas externos, com um foco particular no reaproveitamento de funcionalidades e de componentes semelhantes em produtos diferentes.

Uma funcionalidade comum a muitos desses produtos é o *reporting*, que consiste na transformação de dados em informações úteis para o utilizador, sendo que neste caso o foco está centrado na impressão de documentos.

Esta dissertação tem como objetivo isolar toda a lógica de reporting num único conjunto de microsserviços, para que os todos os produtos primavera tenham acesso ao mesmo. O sucesso na construção deste microsserviço parte da escolha da melhor ferramenta de *reporting* para o contexto em questão e, principalmente, no desenho da melhor arquitetura de microsserviços.

Palavras-chave Microsserviços, Arquiteturas de *Software*, Ferramentas de *Reporting*, Padrões Arquiteturais

Abstract

Cegid Primavera has been developing a microservices cloud to facilitate the integration between its internal products and external systems, with a particular emphasis on reusing similar features and components across different products.

A common feature among many of these products is the reporting function, which involves transforming data into useful information for users, primarily through document generation.

The main goal of this dissertation is to consolidate all the reporting logic within a single set of microservices, enabling all Cegid Primavera products to access it. The success of building this microservice depends on selecting the most suitable reporting tool for the given context and designing the optimal microservices architecture.

Keywords Microservices, Software Architectures, Reporting Tools, Architectural Patterns

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

IU Interface de Utilizador.

RES Reporting Engine Service.

TI Tecnologias de Informação.

UML Unified Modeling Language.

Glossário

cash flow Representa as entradas e saídas de capital de uma empresa, num determinado período de tempo [17].

deployment Processo de colocar um software em funcionamento em um ambiente de produção. Este processo envolve a instalação do software nos servidores ou infraestrutura de TI necessários, bem como a configuração do software para atender às necessidades do ambiente de produção.

PME Pequena ou média empresa que satisfaça os critérios definidos na legislação europeia [38].

template Ambiente estabelecido como modelo, permitindo criar conteúdos de uma forma rápida.

1 Introdução

Neste capítulo introdutório é apresentado o tema desenvolvido durante a dissertação, dando um contexto geral aos tópicos que serão abordados. Para além disso, são reveladas as razões que motivaram a sua realização e os objetivos que se pretende cumprir para garantir o sucesso da mesma.

1.1 Contextualização

O mercado empresarial, ao longo dos últimos anos, tem vindo a atravessar um momento de grande transformação digital. Num mundo cada vez mais informatizado, a disponibilização de soluções rápidas, inovadoras e cada vez mais acessíveis tanto em termos de custos financeiros, como geográficos, têm sido progressivamente requisitos essenciais que qualquer empresa de desenvolvimento de software pretende cumprir quando apresenta os seus produtos. Esta necessidade levou a um aumento significativo da adoção de uma abordagem baseada em Computação em Nuvem (ou *Cloud Computing*).

Segundo Marston et al. [31], a computação em nuvem representa a convergência de duas importantes tendências na área das TI. Em primeiro lugar oferece uma maior eficiência através da utilização de recursos de hardware e software escaláveis. Para além disso, entregam também ao negócio um serviço ágil, isto é, capaz de ser instalado rapidamente, com um forte poder de processamento e capaz de responder em tempo real aos requisitos do utilizador através de aplicações móveis. Por todas as vantagens que este tipo de soluções oferece, várias empresas de renome como *Amazon*, *Microsoft* e *IBM* têm adotado plataformas *cloud* como modelo principal de entrega das suas aplicações mais modernas [18].

Por sua vez, a computação em nuvem trouxe diversos desafios no que toca à gestão de infraestruturas cada vez mais complexas e distribuídas. Por este motivo, como qualquer tipo de aplicação, é essencial adotar arquiteturas de software adequadas e que permitam à plataforma cumprir os requisitos desejados.

Os padrões arquiteturais são soluções gerais e reutilizáveis que ajudam a definir as caraterísticas base e comportamento de uma aplicação. Assim sendo, é necessário reconhecer as qualidades de cada um, pontos fortes e fracos, para escolher aquele que será mais adequado às necessidades e objetivos do software em questão [32]. O padrão arquitetural que tem ganho cada vez mais relevância no contexto da

computação em nuvem é a arquitetura microsserviços. Microsserviços é uma técnica de desenvolvimento de aplicações de software, onde a aplicação é estruturada como sendo uma coleção de pequenos serviços *loosely coupled*, isto é, com o mínimo de dependências entre si [18].

A Cegid Primavera tem vindo a desenvolver a sua *cloud* de microsserviços para facilitar a integração dos seus vários produtos e dos seus produtos com sistemas externos, com um foco particular no reaproveitamento de funcionalidades e de componentes semelhantes em produtos diferentes.

Por sua vez, um módulo comum a muitos desses produtos é o *reporting*. O *Reporting* engloba um conjunto de funcionalidades, sendo um dos principais objetivos a transformação de dados para informações úteis ao utilizador. No caso concreto dos produtos Cegid Primavera, recorrendo às capacidades de *reporting* pretende-se a impressão de dados de diversos tipos, estruturando documentos simples e de fácil compreensão e com formatos familiares aos utilizadores dos produtos. Um exemplo de um caso de uso para esta funcionalidade pode ser a geração de uma fatura de uma compra para formato *pdf*.

Cegid Primavera

A Cegid Primavera [26] *é uma plataforma de software empresarial independente no mercado ibérico que oferece uma vasta gama de soluções de software baseadas na cloud, abrangendo faturação, contabilidade, processamentos salariais e planeamento de recursos empresariais (ERP)* [27]. Com soluções capazes de servir uma variedade de empresas e setores de atividade, desde PMEs, empreendedores, contabilistas e segmentos de mercado em Espanha, Portugal e África, o grupo tem vindo a expandir cada vez mais o seu ecossistema [27]. O Grupo Primavera iniciou a sua atividade em 2019, graças ao apoio da *Oakley Capital*, uma empresa britânica de *private equity*. Esta começou por adquirir a *Ekon* e mais tarde outras 11 empresas, entre elas a Primavera BSS que acabou por dar o nome ao grupo, que neste momento engloba um total de 13 empresas [27]. Mais recentemente, no dia 5 de setembro de 2022, a Cegid, empresa *líder em soluções de gestão baseadas na Cloud para profissionais dos setores: Financeiro (ERP, tesouraria, impostos), de Recursos Humanos (Processamentos Salariais e Gestão de Talentos), Contabilístico, Retalhista, para Empreendedores e Pequenas Empresas* [28], anunciou a sua fusão com o Grupo Primavera. Esta união permitirá acelerar o crescimento do grupo, *especialmente a nível internacional, com sólidas perspetivas de desenvolvimento na América Latina, onde a Cegid tem uma forte presença [28].*

1.2 Motivação e Objetivos

A Cegid Primavera possui um conjunto de produtos *cloud* desenvolvidos com recurso à *framework Elevation*. As funcionalidades de *reporting*, por serem requisitos comuns a quase todos esses produtos, levaram à criação de soluções de modo a evitar redundância de código. Como resultado, foi criado o módulo de *Reporting* na *framework*, que é o componente de código responsável por atender a todas as necessidades de *reporting* dos produtos *Elevation*.

Por outro lado, este recurso por abranger funcionalidades de uma complexidade crescente, apresenta ainda assim alguns problemas inerentes a uma arquitetura monolítica, tais como a dificuldade de escalabilidade e manutenção, falta de flexibilidade tecnológica e problemas de eficiência. Por este motivo, a solução encontrada para superar todos esses obstáculos foi a transição do módulo para um microsserviço de *reporting* onde estão centralizadas todas as funcionalidades inerentes a este domínio e respetivos dados armazenados.

Assim sendo, de um modo geral o objetivo deste projeto é explorar e implementar tecnologias e algoritmos que facilitem a construção de uma arquitetura microsserviços numa plataforma *cloud* e, posteriormente, criar um conjunto de microsserviços específicos responsáveis pelas funcionalidades de *reporting* dos produtos *cloud* da Cegid Primavera.

Assim sendo pretende-se cumprir os seguintes objetivos para garantir o sucesso da dissertação:

- Estudar o padrão arquitetural microsserviços.
- Estudar padrões arquiteturais de software que possam ser integrados numa arquitetura microsserviços.
- Estudar soluções de *reporting* no mercado, nomeadamente analisar as caraterísticas das diferentes ferramentas de *reporting* e identificar a melhor para o microsserviço a desenvolver.
- Desenhar e desenvolver um conjunto de microsserviços de *Reporting* que possam ser integrados pelos produtos *Cloud* da Cegid Primavera.
- Integrar os microsserviços desenvolvidos com os produtos *Elevation* da Cegid Primavera.

1.3 Estrutura do documento

Nesta secção pretende-se fornecer uma visão geral da estrutura adotada para esta dissertação, descrevendo a sequência dos capítulos apresentados e fazendo uma breve síntese do seu conteúdo.

O capítulo 2 engloba todo o estado da arte, onde será descrito o estudo teórico dos temas que envolvem a dissertação, começando pelos padrões arquiteturais de software, dando um maior destaque à arquitetura microsserviços. De seguida será também abordada a funcionalidade de *reporting* e analisadas algumas ferramentas de *reporting*.

No capitulo 3, é feito um estudo das *frameworks* em uso pela Cegid Primavera no desenvolvimento das suas soluções de software, dando particular destaque à arquitetura implementada e aos benefícios que cada uma proporciona aos desenvolvedores da empresa.

No capitulo 4 é apresentada a abordagem inicial ao desenvolvimento do microsserviço RES. Neste segmento são listados os requisitos levantados e apresentados alguns diagramas UML, a fim de modelar a estrutura e comportamento do software a desenvolver.

O capitulo 5 é feita a descrição das tecnologias adotadas e são descritos todos os passos que permitiram a implementação da solução, nomeadamente o desenho da arquitetura, o esclarecimento das funcionalidades desenvolvidas e o uso de ferramentas e bibliotecas externas.

No capitulo 6 é apresentada a fase posterior à implementação do RES, que inclui a sua integração no ecossistema de produtos *cloud Elevation* da Cegid Primavera. Este processo envolveu a adaptações à *framework Elevation*, de modo a transitar as ações de *reporting* para o microsserviço.

Para concluir, o capítulo 7 engloba a análise crítica do trabalho realizado. Neste capítulo, é feita uma revisão dos principais pontos abordados, destacam-se os resultados alcançados e também são abordadas as dificuldades encontradas ao longo do processo. Para além disso, é realizada uma projeção de trabalhos futuros, com vista a expandir o estudo realizado.

2 Estado da arte

Neste capítulo, encontram-se descritos os conceitos teóricos abordados durante a dissertação. Em primeiro lugar será feito um estudo à arquitetura microsserviços, sendo que para entender os seus benefícios é essencial fazer uma introdução à arquitetura monolítica. De seguida serão apresentados alguns padrões no que toca ao desenvolvimento de aplicações com microsserviços.

Na secção seguinte serão apresentados e analisadas as caraterísticas de alguns padrões arquiteturais que podem ser utilizados em conjunto com microsserviços. Por fim, será explicada a funcionalidade de *reporting* e analisadas as caraterísticas de determinadas ferramentas de *reporting*, sendo que serão comparadas segundo alguns critérios definidos.

2.1 Arquitetura Microsserviços

Como já foi referido, a tendência na engenharia de software para a computação em nuvem levou a uma mudança no estilo arquitetural das aplicações mais modernas. Assim sendo, a adoção de uma arquitetura microsserviços tem vindo ganhar cada vez mais popularidade graças às vantagens que oferece em aplicações cada vez mais complexas e distribuídas.

2.1.1 Evolução de Monolítica para Microsserviços

A arquitetura monolítica (Figura 2a) foi durante muito tempo a forma como as grandes empresas desenvolviam o seu software. Este tipo de aplicação pode ser definido como sendo um único 'bloco' ou seja é uma aplicação onde não é possível dividir em módulos, isto é todas as suas funcionalidades estão encapsuladas no mesmo sítio [18]. Quando uma aplicação ainda é pequena e pouco complexa este tipo de arquitetura oferece várias vantagens como por exemplo a facilidade de deployment, de desenvolvimento, manutenção e realização de testes e até na possibilidade de escalar em várias instâncias [18].

No entanto, à medida que são acrescentadas novas funcionalidades e a complexidade do 'monólito' aumenta, começam a aparecer algumas fragilidades caraterísticas deste tipo de arquitetura. Com o

passar do tempo é complicado manter uma estrutura modular, o que consequentemente torna quase impossível que alterações num módulo não afetem outro [23]. Da mesma forma, qualquer alteração ao código implica fazer novamente o deployment de toda a aplicação, que sendo esta complexa este processo irá ser cada vez mais demorado. Para escalar é necessário fazer réplicas de toda a aplicação o que implica também um maior gasto de recursos [23]. Como se pode concluir pela Figura 1, a partir de um certo nível de complexidade a produtividade numa arquitetura monolítica baixa drasticamente.

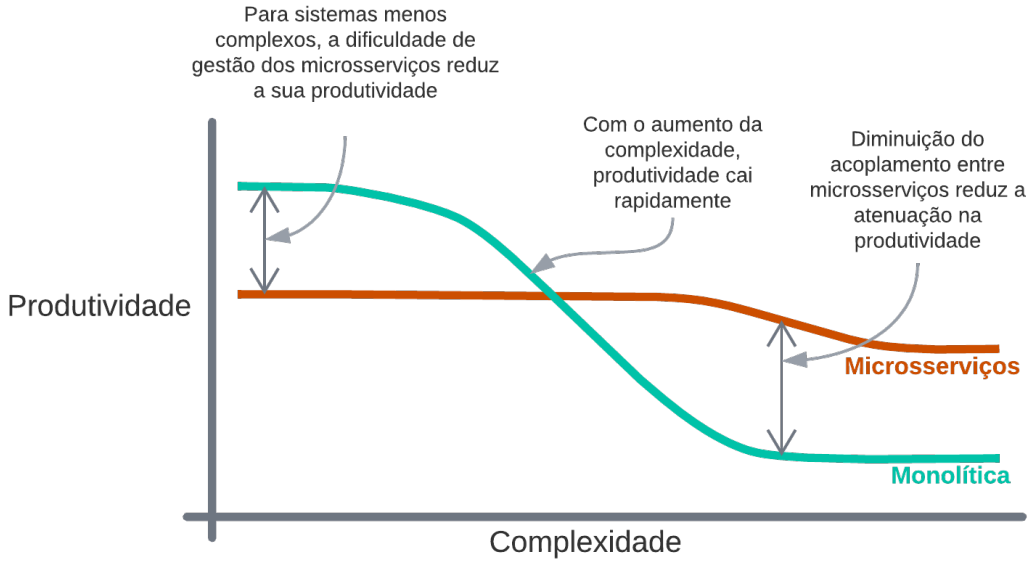


Figura 1: Relação entre produtividade e complexidade de cada arquitetura [22]

Todos estes problemas levaram ao surgimento da Arquitetura Microsserviços (Figura 2b). Este padrão arquitetural assenta em desenvolver uma aplicação constituída por vários serviços sendo que cada um deles é independentemente instalado e escalável. Estes serviços podem comunicar entre si, no entanto conseguem funcionar independentemente uns dos outros. Por este motivo, podem também ser geridos por equipas de desenvolvimento diferentes e escritos em linguagens de programação distintas [23].

Assim sendo, este tipo de arquitetura apresenta inúmeras caraterísticas, que podem ser vantajosas numa aplicação complexa, tais como: [23]

- **Divisão em serviços:** divisão do sistema em componentes independentes e substituíveis, permitindo a implementação e atualização independentes de cada serviço. Isto facilita a manutenção, escalabilidade e evolução contínua do sistema, mas também requer considerações cuidadosas no que toca à comunicação entre esses mesmos serviços.

- **Organização em torno da lógica de negócio:** divisão do sistema com base nas funcionalidades, em vez de camadas tecnológicas, como por exemplo a separação de equipas de *frontend* e *backend*. Ao adotar a abordagem microsserviços, cada serviço representa uma implementação de software completa para uma área especifica de negócios, promovendo a constituição de equipas colaborativas e multifuncionais dentro de um mesmo domínio.

- **Manutenção descentralizada:** permite a flexibilidade no uso de tecnologias, a criação de ferramentas úteis, a gestão flexível de contratos de serviço e a responsabilidade direta das equipas de desenvolvimento pela operação do software.

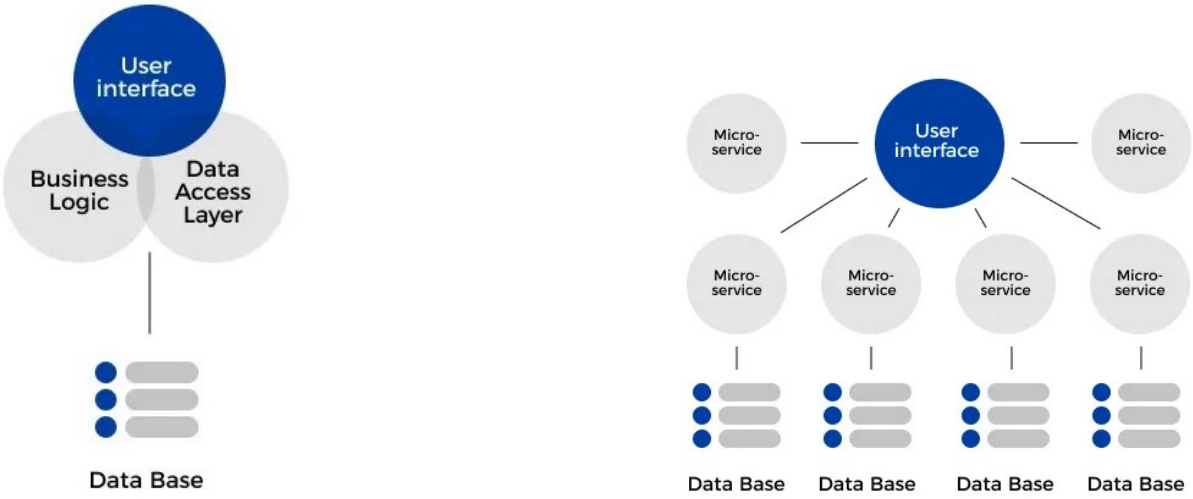
- **Gestão de dados descentralizada:** permite a utilização de modelos de dados diferentes entre sistemas, uso de padrões abertos (como *HTTP*), delimitação de contextos e descentralização do armazenamento de dados. Microsserviços evitam transações complexas, preferindo coordenação sem transações para lidar com eventual consistência. Esta abordagem prioriza flexibilidade e agilidade, mas exige estratégias para gerir inconsistências de forma assíncrona.

- **Automação de infraestruturas:** permite a realização testes automatizados, criação de ferramentas úteis de geração de código e disponibilização rápida dos serviços. Isto contribui para a agilidade e qualidade do desenvolvimento de software baseado em microsserviços.

- **Design inovador:** capacidade de controlar mudanças de forma eficiente. Os serviços são projetados para serem substituídos e atualizados de forma independente. Isto permite uma abordagem mais granular na disponibilização dos serviços e evita a necessidade excessiva de criação de diferentes versões. Deste modo, é favorecida a adaptação rápida e controlada do software.

2.1.2 Padrões Arquiteturais de Microsserviços

O padrão arquitetural microsserviços tem vindo a ganhar espaço como uma alternativa às aplicações monolíticas. No entanto, como se trata de um padrão ainda em desenvolvimento, existe ainda alguma confusão na indústria acerca de como o mesmo deve ser implementado [32]. Deste modo, o artigo [39] categorizou alguns padrões que neste momento tem tido um maior crescimento dentro das arquiteturas microsserviços nos 3 tipos mencionados de seguida.



(a) Arquitetura monolítica

(b) Arquitetura microsserviços

Figura 2: Monolítica versus Microsserviços [29]

Padrões de Organização e Comunicação

Os padrões apresentados nesta secção são orientados à comunicação e coordenação entre os microsserviços.

- **API-Gateway Pattern**

  Numa arquitetura microsserviços, cada serviço pode expor as suas funcionalidades através de uma API. Esta caraterística implica que um cliente que pretende utilizar a aplicação é obrigado a conhecer cada uma delas e fazer pedidos a cada serviço individualmente. Esta técnica evidencia alguns problemas, nomeadamente em casos em que se pretenda implementar funcionalidades complexas que consomem múltiplos serviços. Esta particularidade obriga o cliente a efetuar múltiplos pedidos com base no número de serviços a consumir [33]. Para resolver este problema surge o padrão *API-Gateway* (Figura 3), que consiste na criação de único ponto de entrada do sistema que serve como intermediário e faz o roteamento do pedido para os serviços necessários. O *API-Gateway* é responsável por receber os pedidos dos clientes, direcionar para os microsserviços apropriados e agregar os resultados obtidos para retornar ao cliente [39]. Isto permite que seja possível criar *APIs* personalizadas para cada cliente de acordo com as suas necessidades específicas [39]. Para além disso, pode ser também providenciar outras funcionalidades tais como autenticação, monitorização e processamento de respostas estáticas. Por geralmente ter acesso à localização e endereços dos serviços o *API-Gateway* também pode atuar como um balanceador de carga, caso haja múltiplas instâncias de um mesmo serviço [39].

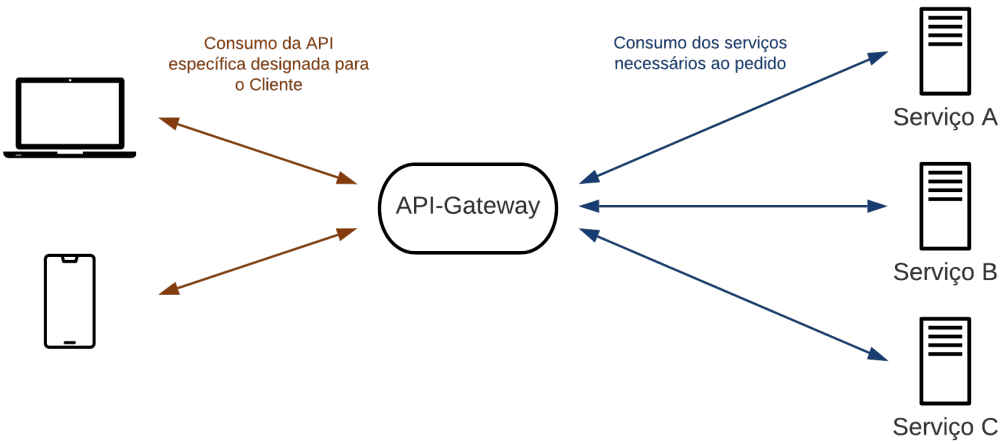


Figura 3: Padrão *API-Gateway* [39]

- **Client-Side Discovery Pattern**

  Num sistema tradicional com serviços distribuídos, estes tipicamente são lançados em localizações e máquinas bem definidas, permitindo assim o rápido acesso através de algum protocolo de comunicação. No entanto, uma aplicação moderna baseada em microsserviços normalmente opera em ambientes de *containers* ou virtualizados onde o número de instâncias de cada serviço e as suas localizações podem variar dinamicamente [34]. Por esse motivo, é essencial criar um método de descoberta da localização das instâncias de todos os seus serviços. Para resolver este problema é normalmente implementado um *Service Registry*, que consiste num serviço de armazenamento de dados onde estão registados todos os serviços e respetivas instâncias e localizações [36]. Assim sendo, uma das formas de obter a localização de um serviço é implementando o padrão *Client-Side Discovery*.

  Tal como se pode verificar na Figura 4, este padrão pode ser demonstrado pela execução de 3 passos essenciais. Numa fase inicial, cada instância lançada tem acesso a uma interface de registo que lhe permite comunicar com o *Service Registry*. É através dela que deve efetuar o registo fornecendo as informações necessárias para que possa estar disponível a receber pedidos do cliente. Por sua vez, o cliente deve consultar o *Service Registry* de modo a obter o conjunto de todas as instâncias ativas e respetivos endereços de um determinado serviço. O passo seguinte consiste na escolha de uma das instâncias da lista recebida, que é feita pelo cliente, recorrendo habitualmente a um algoritmo de balanceamento de carga. Tendo em conta o opção selecionada, é assim feita a conexão direta entre o cliente e a instância do serviço.

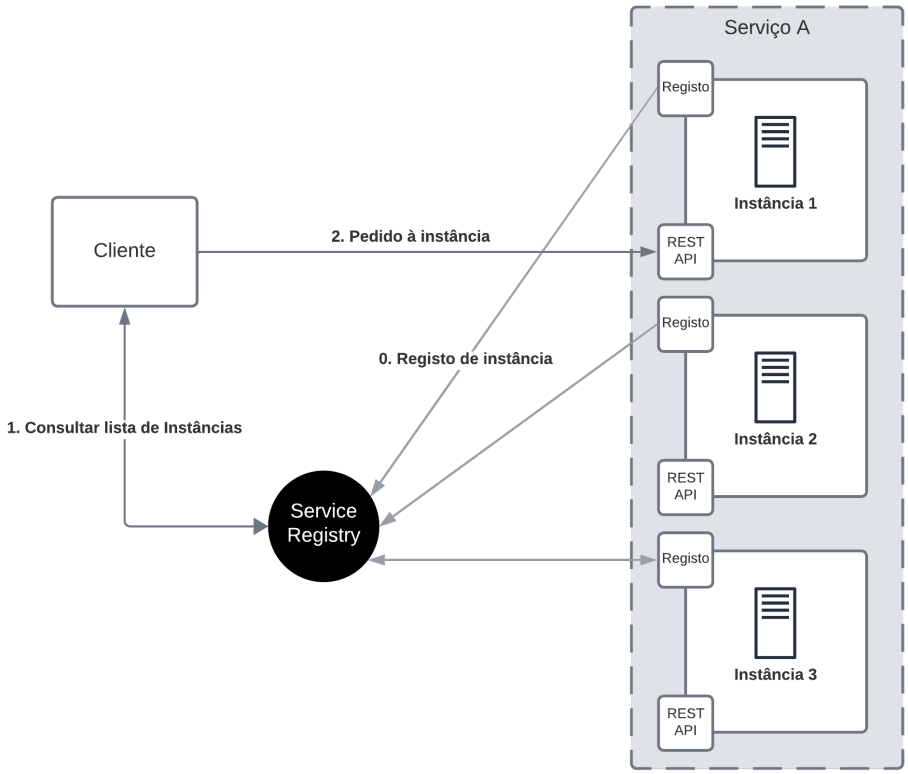


Figura 4: Padrão *Client-Side Discovery* [39]

- **Server-Side Discovery Pattern**

  À semelhança do padrão anterior, o *Server-Side Discovery* também tem como objetivo descobrir a localização de um serviço através do *Service Registry*. Da mesma forma, isto significa que cada instância lançada deve efetuar inicialmente o seu registo (Figura 5). Contrariamente ao *Client-Side Discovery*, este padrão assenta em abstrair a escolha das instâncias através da adição de um balanceador de carga (Figura 5). Assim sendo, o cliente que pretende consumir o serviço efetua o pedido ao balanceador de carga, que por sua vez consulta o *Service Registry* e seleciona a instância mais adequada. Por fim, cabe ao mesmo direcionar o pedido do cliente para o endereço correto. É de notar que neste padrão nunca existe comunicação direta entre o cliente e as instâncias [35].

- **Hybrid Pattern**

  O padrão *Hybrid* tem como objetivo combinar a implementação do *API-Gateway* com o *Service Registry*, com a particularidade do *API-Gateway* funcionar como um encaminhador de mensagens (*message bus*). Assim sendo, os clientes comunicam sempre com o *message bus* que, uma vez que atua como *Service Registry*, conhece todas as instância dos serviços e automaticamente

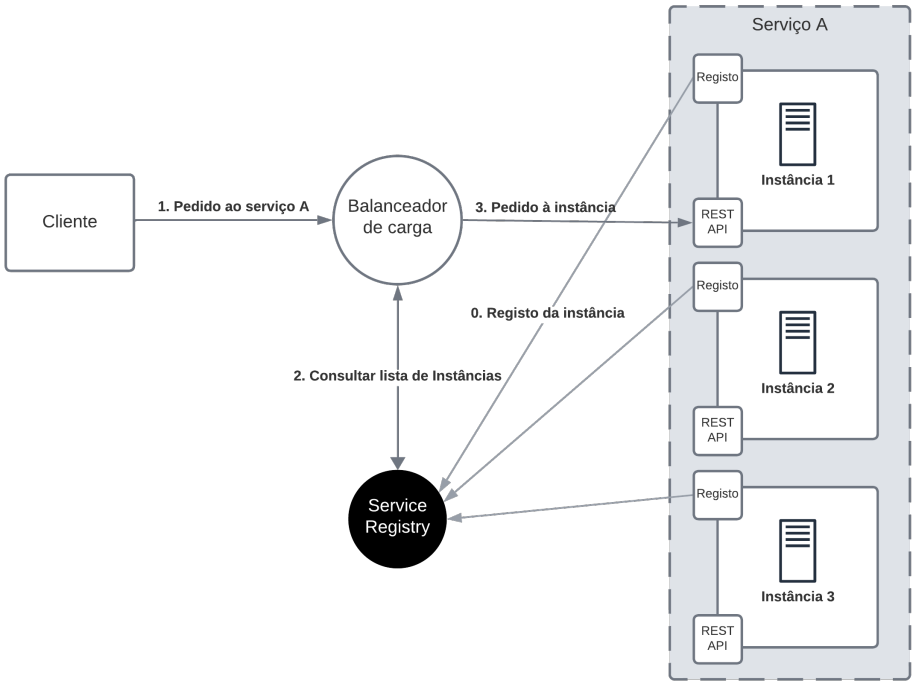


Figura 5: Padrão *Server-Side Discovery* [39]

encaminha o pedido do utilizador para o serviço pretendido [39].

Padrões Orientados a Estratégias de Deployment

A instalação de uma aplicação baseada em microsserviços é uma das fases mais importantes do desenvolvimento uma vez que poderá afetar a performance da mesma. Por ser uma arquitetura complexa, existem diferentes abordagens a tomar no que toca à fase do deployment, no entanto nesta secção serão abordados apenas dois padrões.

- **Multiple Service per Host Pattern**

  Este padrão, tal como o nome indica consiste em executar múltiplas instâncias de diferentes serviços num mesmo *host* (máquina física ou virtual) [39].

- **Single Service per Host Pattern**

  Ao contrário do anterior, este padrão consiste em fazer o *deploy* de apenas uma instância de um único serviço em cada host [39].

Padrões Orientados à Base de Dados

Nesta secção estão presentes padrões orientados à gestão de dados, nomeadamente à forma como são armazenados os dados da aplicação.

- **Shared Database Server Pattern**

  Neste padrão todos os microsserviços acedem uma mesma base de dados [39]. Esta estratégia é normalmente utilizada num estado de transição da aplicação monolítica para microsserviços, por ser mais acessível. No entanto não cumpre o princípio de os microsserviços serem independentes entre si.

- **Database-per-Service Pattern**

  Neste padrão cada serviço tem acesso exclusivo a uma base de dados própria [39]. Isto significa que os serviços não tem acesso a dados de outros serviços tendo apenas à sua API pública, ou seja, ao contrário do anterior respeita o princípio da independência de dados entre microsserviços.

2.2 Padrões de Arquiteturas Aplicacionais

Os padrões de arquiteturas aplicacionais são soluções comprovadas para problemas recorrentes na arquitetura de software. Estes são responsáveis por descrever a estrutura geral de um sistema de software, bem como as interações entre os seus componentes. Ter conhecimento das caraterísticas, pontos fortes e fracos de cada padrão arquitetural é essencial no momento de escolher aquela que mais se adequa, tanto aos requisitos levantados, como também aos objetivos que se pretende da solução final. [32]

2.2.1 Layered Architecture

O padrão mais utilizado no desenvolvimento de software é a arquitetura em camadas (Layered Architecture), à qual também é comum chamar arquitetura multi-camada [32]. Este padrão é bem conhecido pela maior parte dos desenvolvedores e, por este motivo é muitas vezes a escolha preferencial no desenho de uma aplicação por ser bastante intuitiva e eficiente na maior parte dos casos.

A *Layered Architecture* (Figura 6) consiste na organização dos componentes da aplicação em camadas horizontais sendo que cada uma tem um papel específico dentro da mesma [32]. Apesar do padrão não especificar um número ou tipo específico de camadas necessárias, a maior parte das arquiteturas deste tipo consistem em 3 camadas essenciais: apresentação, lógica de negócio e persistência. Não obstante, é bastante comum aplicações complexas incluir 5 ou mais camadas se necessário.

A divisão de responsabilidades é uma caraterística poderosa neste tipo de arquitetura. Os componentes dentro de cada camada apenas devem lidar com a lógica específica da mesma. Desse modo, este tipo de classificação dos componentes permite facilitar o desenvolvimento e manutenção da aplicação, na medida em que, cada elemento terá uma interface bem definida e um âmbito limitado à lógica da camada onde está inserido [32].

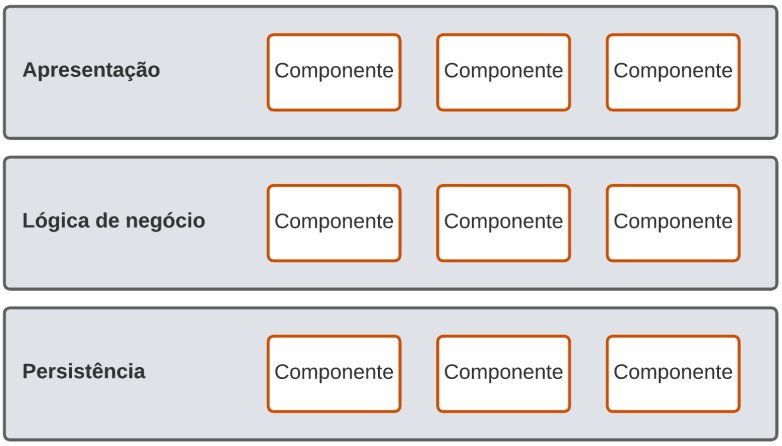


Figura 6: Arquitetura em camadas [32]

2.2.2 Domain-Centric Architectures

Com o aumento da complexidade de uma aplicação a utilização de uma arquitetura em camadas pode deixar de ser uma solução viável para um software que pretenda acompanhar a evolução tecnológica e a adição de novas funcionalidades.

Considerando a arquitetura clássica de 3 camadas da Figura 6, tendo em conta que cada camada só tem acesso à camada inferior e desenhando a arquitetura em formato circular pode-se verificar que a persistência de dados está no localizada no centro. Isto significa que todas as dependências da aplicação apontam para a persistência, ou seja, todo o desenvolvimento do sistema depende da forma como os dados são armazenados.

A arquitetura *Domain-Centric* (Figura 7) consiste numa nova perspetiva de desenho que tem como base colocar no centro aquilo que é essencial na resolução de um problema por parte da aplicação. Por sua vez, elementos com menos importância pertencem a camadas cada vez mais superficiais [30]. Desta forma, todas as dependências da aplicação passam a apontar para o domínio, sendo que a base de dados será uma das camadas mais externas por ser considerada um 'detalhe' da aplicação [30]. Isto permite

dar mais importância às funcionalidades, independentemente das tecnologias que são implementadas, tornando a aplicação mais acessível a mudanças e capaz de evoluir.

Existem diferentes tipos de arquiteturas *Domain-Centric* tais como a *Hexagonal Architecture* ou a *Onion Architecture*, no entanto nesta secção será destacada a *Clean Architecture* por ser a mais complexa, sendo que se pode considerar como uma evolução das duas mencionadas e, por outro lado por ser aquela que é aplicada nos microsserviços da Cegid Primavera.

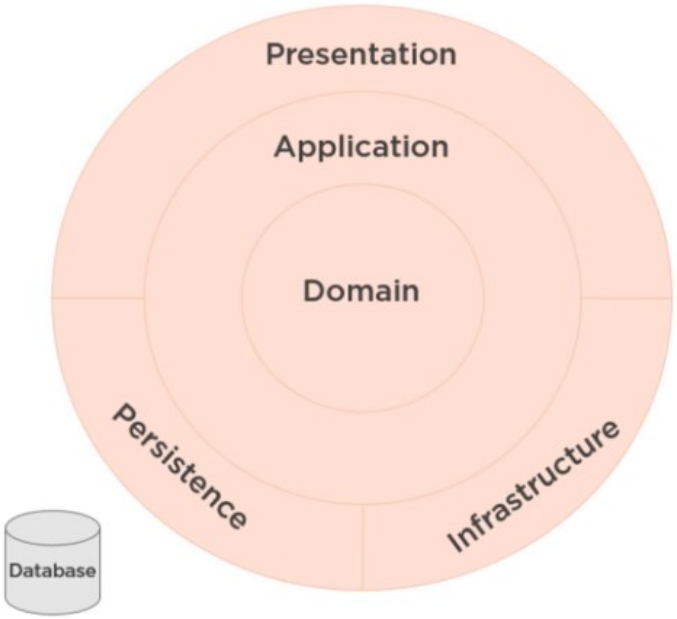


Figura 7: Arquitetura *Domain-Centric* [1]

Clean Architecture

A *Clean Architecture* é uma filosofia de desenho de software que separa os elementos em camadas em forma de anéis [30]. A principal premissa desta arquitetura é que as dependências de código só apontam para o interior, isto é "um elemento declarado numa camada exterior nunca pode ser mencionado no código de qualquer camada interior. Isto inclui métodos, classes, variáveis, ou outra entidade de software"[5].

A escolha deste tipo de arquitetura traz diversas vantagens tais como: [30]

- Independência ao nível da base de dados e das *frameworks* utilizadas.
- A interface de utilizador pode ser facilmente alterada.
- A aplicação é facilmente testável.

- Torna a aplicação reutilizável e de fácil manutenção.

2.2.3 Event-Driven Architecture

Uma arquitetura orientada por eventos (*Event-Driven Architecture*) é um padrão que consiste na execução de eventos como forma de comunicar e executar uma transação em serviços que estejam distribuídos, sendo muito utilizado em aplicações com uma arquitetura microsserviços [2]. Um evento pode ser definido como uma mudança de estado ou uma atualização, como por exemplo um produto ser adicionado ao carrinho de compras num site de compras online.

Esta arquitetura é normalmente bastante complexa, no entanto, tal como se pode verificar na Figura 8, é comum a existência de 3 componentes chave: produtores, *routers* e consumidores [2]. Assim sendo, o fio de execução parte dos produtores que criam um evento e enviam ao *router*. Este por sua vez filtra quais os alvos desse evento e envia aos respetivos consumidores que executam uma ação de acordo a mensagem recebida [2].

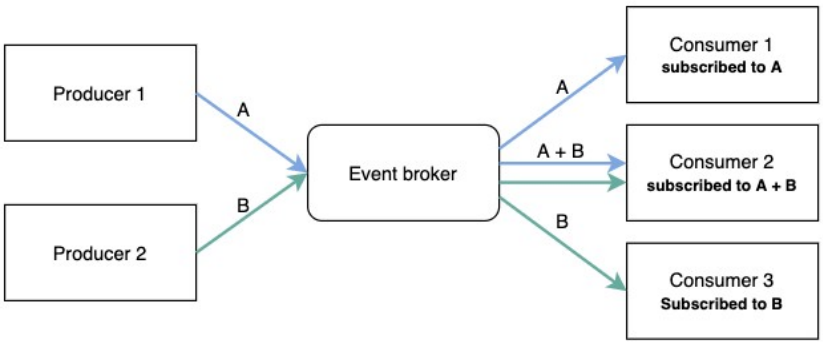


Figura 8: Arquitetura *Event-Driven* [6]

2.2.4 Microkernel Architecture

A *Microkernel Architecture* (ou arquitetura *microkernel*) é um padrão muito utilizado em aplicações *product-based* [32]. Este tipo de aplicações são aquelas que normalmente estão disponíveis em diferentes versões dependendo das funcionalidades que oferecem.

Sendo assim, a arquitetura *microkernel*, tal como se pode verificar na Figura 9, consiste na divisão da aplicação em dois componentes principais: a base ou *core* do sistema e os módulos adicionais (*plug-in modules*) [32]. Deste modo, no *core* estão presentes todas as funcionalidades base da aplicação, isto é, aquelas que são necessárias ao funcionamento da mesma. Por outro lado, as funcionalidades que

não são essenciais mas acrescentam uma utilidade adicional ao sistema estão separados da base por componentes individuais (componente *Plug-in*). Estes módulos adicionais são componentes independentes que contém funcionalidades adicionais e código específico que tem como objetivo expandir o núcleo da aplicação para produzir capacidades complementares ao sistema. Geralmente estes módulos devem ser independentes entre si, no entanto podem existir casos onde um *plug-in module* exige a presença de outro em simultâneo [32].

É de notar que o *core* do sistema tem saber quais os módulos que estão disponíveis e ser capaz de aceder aos mesmos. Esta tarefa pode ser solucionada através de um sistema de registo de cada *plug-in* [32].

Um produto que implementa este tipo de arquitetura é o Eclipse IDE. Quando é descarregado o editor fornece apenas funções basicas, no entanto pode-se tornar bastante costumizável e útil quando são instalados *plug-ins* adicionais [32].

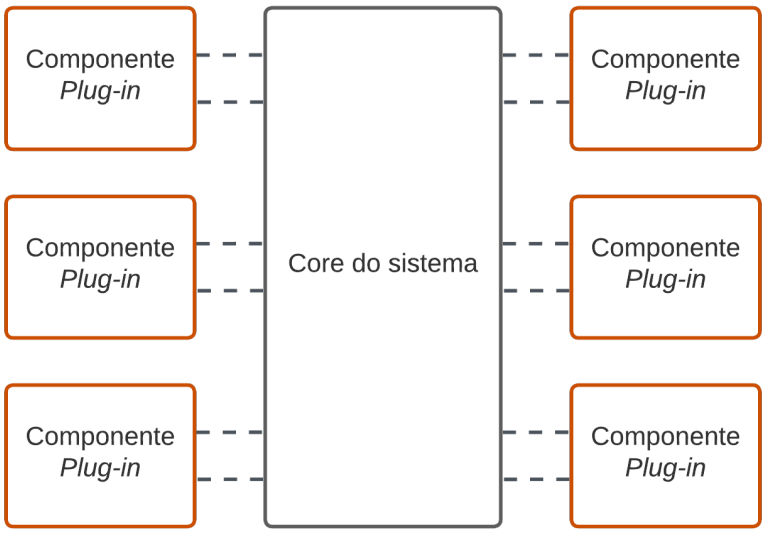


Figura 9: Arquitetura *Microkernel* [32]

2.3 Funcionalidade de Reporting

Nos dias de hoje, as empresas têm cada vez mais a necessidade de armazenar grandes quantidade de dados. Consequentemente, para qualquer negócio que recolhe informações de diferentes tipos, torna-se indispensável ter um sistema que, a partir desses mesmo dados, seja capaz de analisar e monitorizar inúmeros aspetos, tais como a performance da empresa, o cash flow, previsões, vendas e lucros, entre

outros [3].

Por sua vez, a Cegid Primavera, sendo uma plataforma de software empresarial que oferece, por exemplo, soluções abrangendo faturação e contabilidade, torna-se fundamental que os seus produtos sejam capazes de gerar documentos tais como faturas, listas de stocks de produtos, listas de clientes, ou documentos de exigência legal.

Desta forma, quando se fala em textitReporting no desenvolvimento de *software*, trata-se da funcionalidade responsável por transformar os dados fornecidos em informações úteis para o utilizador. Assim sendo, segundo Anderson um Report consiste no resultado da inserção de dados num template pré-definido, os quais podem ser obtidos a partir de distintas fontes de dados, e que passam geralmente por um processamento intermediário.

É de notar que existem diferentes tipos de *reporting*, sendo os mais simples aqueles que requerem um menor processamento, como por exemplo faturas e impressões de dados no geral. Por outro lado, um reporting mais complexo pode envolver uma processamento mais aprofundado dos dados para encontrar padrões e criar modelos preditivos, utilizando por exemplo *machine learning* [3].

A Figura 10 representa o processo geral da criação de um *report* seja ele simples ou complexo. Como se pode verificar, este processo parte sempre da obtenção de dados a partir de uma fonte de dados. É de notar que, estes podem ser obtidos de diversas fontes tais como Bases de dados, ficheiros *json*, *xml*, API externas, entre outros. De seguida, a segunda fase consiste no processamento dos dados, onde por exemplo se podem calcular alguns campos do *report* como valores totais, ou alterar a representação de alguns campos. Por fim, o Reporting Engine injeta os respetivos dados processados num template, normalmente previamente definido pelo utilizador. Da formatação do template com os dados fornecidos resulta um Report Model que, através de um Report Generator pode ser processado, sendo que o seu objetivo passa por formatar o mesmo para um formato legível ao utilizador (exemplos: *pdf*, *excel*, *csv*).

A crescente necessidade da implementação desta funcionalidade no mundo empresarial, levou ao aparecimento de diversas ferramentas de Reporting, as quais desempenham um papel fundamental na simplificação e aceleração deste processo.

2.4 Estudo de ferramentas de Reporting

Tal como o *Reporting*, uma ferramenta de Reporting pode ser definida como um sistema que recebe dados de diferentes fontes e os transforma em tabelas, gráficos, representações visuais e outros formatos para que a informação seja mais acessível de analisar pelo utilizador [4].

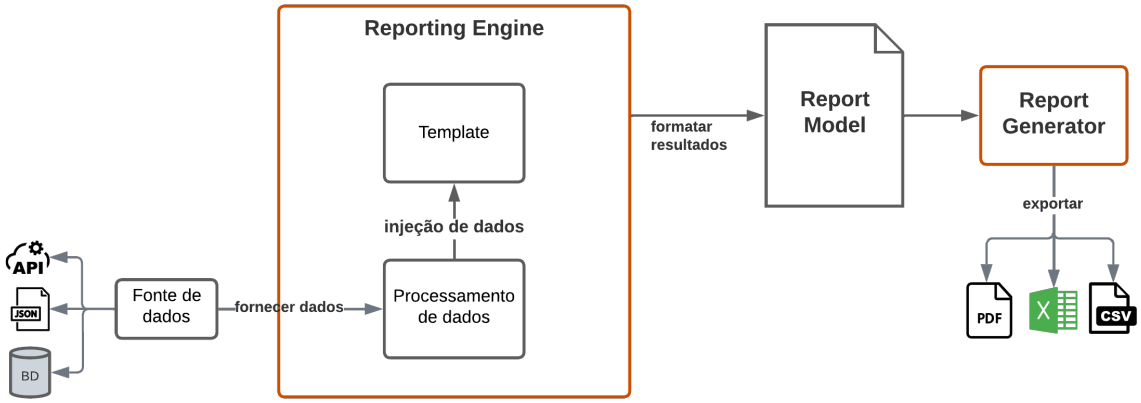


Figura 10: Diagrama geral do processo de Reporting

Como foi referido na secção anterior, existem diferentes tipos de *reporting*, consequentemente, é possível encontrar *ferramentas de reporting* que se focam mais em determinados tipos. No contexto do projeto em questão a desenvolver com a Cegid Primavera, uma vez que o objetivo do microsserviço é a geração de documentos, nesta secção será feito um estudo focado apenas em ferramentas capazes de suportar esta funcionalidade.

Após ter sido feita uma aprofundada pesquisa sobre as diferentes ferramentas de *reporting* existentes no mercado, analisaram-se as caraterísticas de cada uma e foram testadas aquelas que potencialmente poderiam oferecer mais vantagens no contexto do projeto a desenvolver. Sendo assim, foram escolhidas as quatro seguintes: DevExpress [19], Telerik [40], Crystal Reports [37] e Windward Core [41].

No entanto, apesar da Crystal Reports e a Windward Core ambas apresentarem caraterísticas relevantes, acabaram por ser excluídas do estudo por falta de atualização tecnológica. Com a constante evolução na área do desenvolvimento de *software*, é fundamental que sejam utilizadas ferramentas que estejam adaptadas à realidade do momento e que possam evoluir ao longo do tempo.

2.4.1 DevExpress Reporting

DevExpress Reporting [19] é uma ferramenta de geração de *reports* para aplicações Windows e web. Ela permite a criação de relatórios complexos com tabelas, gráficos, imagens e sub-relatórios, e permite exportar para vários formatos incluindo PDF, HTML e Excel.

Esta ferramenta apresenta várias caraterísticas tais como [20]:

- Suporte para vários formatos de saída: O DevExpress Reporting oferece suporte para vários formatos de saída, incluindo PDF, HTML, Excel e muito mais.

- Personalização de *reports*: permite a personalização dos *reports*, incluindo a adição de tabelas, gráficos, imagens e *sub-reports*.
- Integração com diversas fontes de dados: O DevExpress Reporting tem suporte para a integração com várias fontes de dados, incluindo bases de dados SQL, conexão com Entity Framework, objetos customizáveis ou ficheiros *excel*, *json*, entre outros.
- Exportação em massa: capacidade de exportar *reports* em massa para vários formatos de saída, incluindo PDF, HTML e Excel.
- Facilidade de uso: interface intuitiva e fácil de usar, para que os desenvolvedores possam criar *reports* rapidamente e sem dificuldade.
- Suporte a várias plataformas: compatível com plataformas como Windows, Web e Mobile.
- Suporte técnico: DevExpress oferece suporte técnico para os seus produtos, incluindo o DevExpress Reporting.

2.4.2 Telerik Reporting

Telerik Reporting [40] é uma ferramenta poderosa para criar, visualizar e exportar *reports* interativos. A ferramenta inclui um *report designer*, *report viewer* e um *report server* para criar, visualizar e gerir *reports*.

Esta ferramenta apresenta várias caraterísticas tais como:

- Suporte para vários formatos de saída: O Telerik Reporting oferece suporte para vários formatos de saída, incluindo PDF, HTML, Excel, Word, MHTML, CSV e muito mais.
- Personalização de *reports*: personalização de *reports*, incluindo a adição de tabelas, gráficos, imagens e *sub-reports*.
- Integração com diversas fontes de dados: O DevExpress Reporting tem suporte para a integração com várias fontes de dados, incluindo bases de dados SQL, conexão com Entity Framework, objetos customizáveis ou ficheiros *excel*, *json*, entre outros.
- Exportação em massa: Ele oferece a capacidade de exportar relatórios em massa para vários formatos de saída, incluindo PDF, HTML, Excel, Word, MHTML, CSV e muito mais.

- Recursos avançados: O Telerik Reporting também possui recursos avançados, como criação de relatórios com várias páginas, geração de relatórios com base em várias fontes de dados e suporte para relatórios dinâmicos.
- Facilidade de uso: O Telerik Reporting tem uma interface intuitiva e fácil de usar, para que os desenvolvedores possam criar relatórios rapidamente e sem dificuldade.
- Suporte a várias plataformas: O Telerik Reporting é compatível com várias plataformas, incluindo Windows, Web, Mobile, e Cloud.
- Suporte técnico: Telerik oferece suporte técnico para seus produtos, incluindo o Telerik Reporting.
- Preview de relatórios: O Telerik Reporting oferece uma visualização prévia dos relatórios, permitindo que os usuários vejam o *report* final antes de exportá-lo.

2.4.3 Comparação de ferramentas

Como se pode verificar nas secções anteriores ambas as ferramentas de *reporting* estudadas apresentam, na teoria caraterísticas muito semelhantes. Contudo, após testar na prática a criação de um mesmo *report* com cada uma delas foi possível encontrar algumas diferenças que é necessário destacar.

Desta forma de acordo com os seguintes critérios de comparação foi possível concluir:

- **Atualização tecnológica**

  Tendo sido um dos principais requisitos para a escolha, ambas as ferramentas estão atualizadas a nível tecnológico.

- **Preço**

  Ambas as ferramentas têm um preço mínimo de $599.99, sendo que no caso da *Telerik* o preço pode aumentar consoante o suporte pretendido.

- **Capacidade para suportar dados generalizados**

  Ambas as ferramentas suportam várias fontes de dados. Relativamente à utilização de dados em *json* a *DevExpress* apresentou uma limitação na medida em que é necessário fornecer inicialmente os campos do ficheiro antes de receber os dados ao contrário da *Telerik* que através do ficheiro já infere as suas propriedades.

- **Documentação**

  DevExpress apresenta uma documentação mais completa e organizada.

- **Popularidade**

  DevExpress apresenta uma maior popularidade entre a comunidade o que pode demonstrar que pode ser uma melhor solução. Esta caraterística pode levar também a que haja uma maior evolução da ferramenta ao longo do tempo.

- **Facilidade de uso para developer**

  Para testar este critério foi desenvolvido com ambas as ferramentas um *report* em código utilizando as bibliotecas de .NET correspondentes. Por ter uma melhor documentação e maior popularidade entre a comunidade, acabou por ser mais rápido desenvolver o *report* utilizando a DevExpress. Para além disso a sua API revelou ser mais intuitiva o que demonstrou uma curva de aprendizagem menor.

- **Facilidade de uso para utilizador (designer)**

  A *DevExpress* apresenta um designer mais intuitivo ao utilizador, mais precisamente na forma como são definidas as fontes de dados de uma secção específica do *report*. Para além disso a forma como são aplicados os estilos é mais clara e simples de implementar.

2.5 Trabalho relacionado

O desenvolvimento do presente projeto pretende complementar estudos prévios realizados na *Cegid Primavera*, incluindo dissertações elaboradas tanto no âmbito de arquiteturas de software e microsserviços, como no contexto de funcionalidades de *reporting*.

"Arquitetura de micro-serviços e o caso da Framework Lithium da PRIMAVERA BSS"

A dissertação "Arquitetura de micro-serviços e o caso da Framework Lithium da PRIMAVERA BSS"[25], escrita por Fábio Daniel de Sá Gonçalves, fornece informações úteis sobre como os microsserviços são desenvolvidos na Cegid Primavera, incluindo o uso de *frameworks* que facilitam a construção dos mesmos.

Este estudo compreendeu uma extensa análise às capacidades da *framework Lithium* e as vantagens que a mesma oferece no desenvolvimento e integração dos microsserviços no ecossistema da Cegid Primavera. Para além disso, é descrita uma análise comparativa no que diz respeito às *frameworks*

alternativas existentes no mercado, dando a conhecer as caraterísticas específicas de cada uma. O estado da arte abrange também o estudo das arquiteturas monolítica e microsserviços descrevendo as situações pelas quais é mais vantajosa a utilização de cada uma. Do mesmo modo, são listados alguns padrões convencionados na construção de uma arquitetura microsserviços.

Os conhecimentos teóricos adquiridos permitiram o desenvolvimento de um microsserviço, recorrendo, tanto ao uso da *framework lithium*, como também de alguns padrões de *design* requeridos para uma melhor conceção do mesmo.

Esta dissertação representa uma fonte importante de ideias para o desenvolvimento do projeto atual, especialmente no que diz respeito à escolha de ferramentas e abordagens para a arquitetura de microsserviços a desenvolver. A análise comparativa realizada das *frameworks* existentes permitiu garantir uma melhor escolha em relação às tecnologias a utilizar.

"Análise e Concepção de uma Framework de Reporting genérica e parametrizável"

No que toca à funcionalidade de *reporting* a dissertação "Análise e Concepção de uma Framework de Reporting genérica e parametrizável"[24], escrita por Igor Gonçalo Gomes de Sá, consistiu na conceção de um método genérico para gerar *reports* independentemente da ferramenta de *reporting*. Esta framework foi projetada com o objetivo de proporcionar uma abordagem de impressão de *reports* flexível e adaptável a evoluções tecnológicas.

Esta dissertação compreende inicialmente um estudo teórico de alguns conceitos úteis para o desenvolvimento de uma *framework*, nomeadamente metodologias de desenvolvimento de software e padrões de design. De seguida é analisada a solução de *reporting* em uso no momento pela Primavera BSS e consequentemente são levantados os requisitos para o desenvolvimento da nova *framework*. A criação desta *framework* de *Reporting* envolveu, numa fase inicial, a definição da sua arquitetura e o estabelecimento das funcionalidades correspondentes a cada componente. Para além disso, foi necessário encontrar a melhor forma de padronizar e generalizar as funcionalidades de *reporting*, para que sejam eficientes e cumpram todos os casos de uso pretendidos. Uma das caraterísticas essenciais no desenvolvimento desta *framework* consistiu na sua capacidade de abstrair a ferramenta de *reporting* em uso.

A compreensão deste estudo é essencial para o desenvolvimento do microsserviço de *reporting* no projeto atual, uma vez que, a *framework* desenvolvida corresponde à solução atual de *reporting* da Cegid Primavera. Isto permite fazer uma melhor análise das suas caraterísticas e obter uma perceção clara dos pontos positivos e a melhorar, de modo a idealizar a melhor solução possível para o microsserviço a desenvolver.

3 Arquitetura Existente

Tendo em conta os desafios encontrados ao longo do tempo, o desenvolvimento de software na Cegid Primavera levou à adoção de algumas soluções que visam aumentar a eficiência e velocidade de implementação mas também diminuir o risco de problemas futuros. Surgem assim as *frameworks Lithium* e *Elevation* que têm como objetivo solucionar problemas de contextos diferentes.

3.1 Framework Elevation

No mundo atual, o desenvolvimento de produtos *cloud* é uma das principais tendências para as empresas que desejam promover flexibilidade, escalabilidade, segurança e desempenho. Nesse contexto, a *framework Elevation* desenvolvida pela Cegid Primavera, surge como uma solução abrangente, composta por ferramentas, arquiteturas e diretrizes, que auxiliam as equipas de desenvolvimento a alcançar os objetivos propostos de forma eficiente e consistente.

Entre os principais benefícios da sua utilização está o facto de assentar num desenvolvimento orientado a modelos (*Model-Driven Development*). Essa abordagem tem como objetivo abstrair os detalhes de implementação das soluções, tendo como foco o domínio e a lógica de negócio.

Uma parte significativa do código dos produtos desenvolvidos com esta *framework* é gerada automaticamente a partir de meta-modelos. O código é gerado desde a camada de dados até às *Web APIs*, incluindo SQL, C# e meta-modelos de visualização para serem consumidos pela aplicação cliente. Essa geração automática de código traz diversos benefícios, como a redução da probabilidade de erros, o incentivo à reutilização de código e a garantia de consistência entre os produtos da empresa. Por outro lado, a capacidade de geração automática permite também a abstração tecnológica, uma vez que possíveis evoluções não necessitam reformulações de código muito substanciais. Desta forma, as equipas conseguem garantir que os seus produtos evoluem constantemente do ponto de vista tecnológico.

3.1.1 Arquitetura

A arquitetura dos produtos *Elevation* (Figura 11) segue um modelo cliente-servidor tradicional. A aplicação cliente é implementada em *Angular* e é composta por vários módulos, tanto os fornecidos pela *framework* como os personalizados adicionados posteriormente pelo desenvolvedor.

Relativamente ao lado do servidor, os produtos expõem uma *API Web RESTful* desenvolvida em *ASP.NET*. Essa *API* é utilizada tanto pela aplicação cliente *Elevation* como por outras aplicações que integram com o produto. A arquitetura é projetada para promover a escalabilidade e permitir a evolução tecnológica sem a necessidade de reescrever grande parte do código.

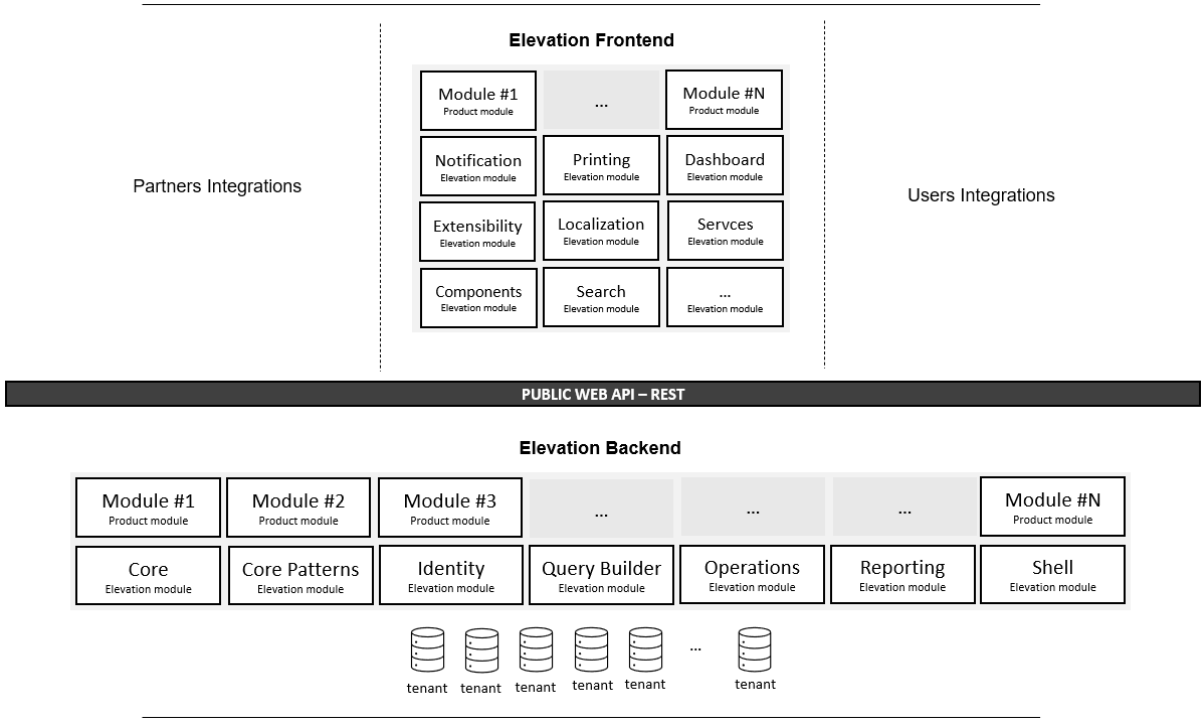


Figura 11: Módulos de um produto com a *framework Elevation* [8]

O ciclo de vida de um pedido ao produto *Elevation* passa por várias camadas, as quais estão representadas na Figura 12. Em primeiro lugar, a chamada é intercetada por *middlewares*, onde são realizadas verificações e validações, como por exemplo autenticação e autorização. De seguida, o pedido passa pelos *controllers* da *API* e pela camada de lógica de negócio, até chegar à camada de dados, sendo que o acesso aos dados é mediado pela *Entity Framework*.

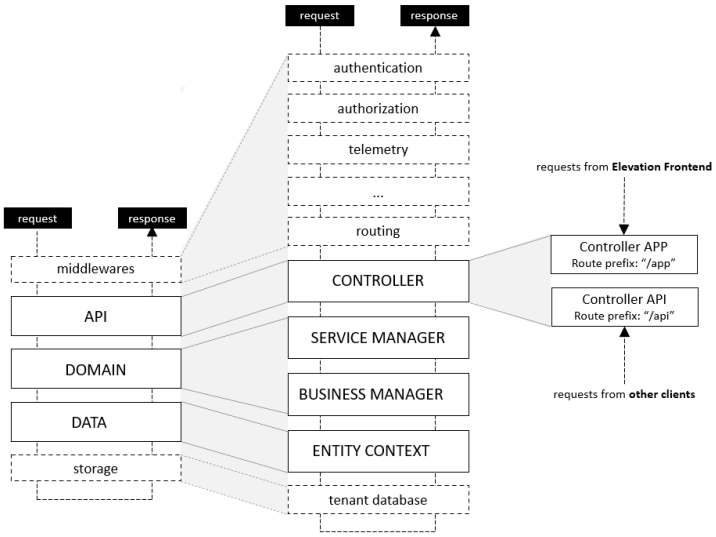


Figura 12: Ciclo de vida de pedidos ao produto *Elevation* [8]

3.2 Framework Lithium

O desenvolvimento de aplicações empresariais enfrenta frequentemente desafios recorrentes relacionados com a convergência de lógicas de negócio, integração de serviços de terceiros e administração de recursos. Nesse contexto, a arquitetura microsserviços surge como uma solução eficaz para lidar com esses problemas. A *framework Lithium* desenvolvida pela Cegid Primavera é uma solução inovadora projetada de modo a facilitar o desenvolvimento, operação e manutenção de um conjunto de microsserviços integrados entre si e com serviços externos.

O principal objetivo do *Lithium* é fornecer um conjunto de microsserviços partilhados por múltiplos produtos, oferecendo funcionalidades simples e independentes tanto para consumidores, como para aplicações e outros serviços. Essa abordagem visa facilitar a manutenção e evolução tanto dos microsserviços como dos consumidores, reduzindo os custos associados ao desenvolvimento e permitindo a partilha e reutilização de recursos. Para além disso, o Lithium permite integrar esses microsserviços em aplicações em nuvem, tais como os produtos ROSE [16], Jasmin [11] e Ekon [7], mas também disponibiliza suporte para aplicações locais sempre que possível.

Em conjunto com as vantagens inerentes à utilização de uma arquitetura microsserviços, a *Lithium Framework* oferece uma série de benefícios que podem ser explorados em diferentes contextos. Esses benefícios incluem:

- **Novos modelos de negócio:** possibilita a produtização dos próprios microsserviços, ou seja,

estes podem ser oferecidos como produtos independentes, permitindo que outras empresas os utilizem nas suas próprias aplicações. Essa abordagem abre novas oportunidades de receita e colaboração entre empresas.

- **Independência das tecnologias utilizadas:** cada serviço pode ser desenvolvido utilizando a tecnologia mais adequada para a sua finalidade, sem a necessidade de se alinhar com as tecnologias utilizadas pelos restantes serviços. Isto permite uma maior flexibilidade e agilidade no desenvolvimento, além de facilitar a adoção de tecnologias emergentes.

- **Separação de responsabilidades:** cada microsserviço é responsável por uma funcionalidade específica, o que simplifica o desenvolvimento, teste e manutenção das aplicações. Além disso, esta abordagem permite também que diferentes equipas de desenvolvimento trabalhem de forma independente, mudando o foco para as interfaces de cada serviço em vez das suas implementações concretas.

- **Agilidade e resposta rápida a novos requisitos:** possibilita uma maior agilidade no desenvolvimento e uma resposta mais rápida a novos requisitos e correção de bugs. Como os microsserviços são distribuídos e independentes, é possível atualizar ou corrigir apenas o serviço afetado, sem a necessidade de atualizar todo o sistema.

- **Testagem facilitada:** como cada microsserviço é isolado e possui uma responsabilidade única, é mais fácil criar testes específicos para cada serviço e validar o seu comportamento de forma independente.

- **Escalabilidade das aplicações melhorada:** como cada microsserviço é independente, é possível escalá-los de forma individual, de acordo com a necessidade específica de cada serviço. Isto resulta numa utilização mais eficiente dos recursos disponíveis, evitando desperdícios e garantindo um melhor desempenho global do sistema.

3.2.1 Arquitetura

Uma das principais características do *Lithium* é a sua arquitetura base, que fornece um conjunto de componentes essenciais para a construção de microsserviços (Figura 13). Esses componentes incluem recursos como rotas, gestão de configurações, autenticação e autorização, entre outros.

Além disso, o *Lithium* também disponibiliza componentes *runtime*, como o *Hydrogen*, que oferece funcionalidades padrão e transversais que podem ser reutilizadas em qualquer microsserviço. Isso significa

que programadores podem aproveitar recursos comuns já implementados na *framework*.

Figura 13: Componentes da *framework Lithium* [15]

Relativamente à anatomia de cada microsserviço, a *framework* define uma arquitetura constituída por vários elementos, com responsabilidades específicas, que comunicam entre si de uma forma bem estipulada. Na Tabela 1 estão listados os principais componentes e respetivas responsabilidades de cada um.

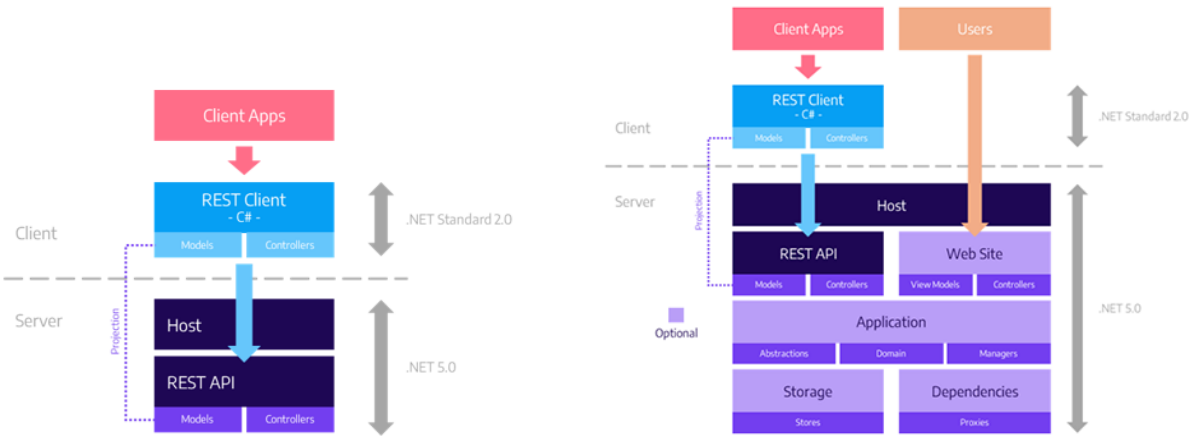
Componente	Responsabilidade
Client App	Aplicação que consome o microsserviço, utilizando a biblioteca de cliente ou, em casos específicos, a *API REST* diretamente. Ela utiliza as funcionalidades disponibilizadas pelo microsserviço para atender às suas necessidades específicas.
REST Client	Biblioteca de cliente *REST* que fornece uma abstração da *API REST*, permitindo que a aplicação cliente faça as invocações corretas ao microsserviço. Essa biblioteca simplifica o uso de recursos transversais, como autorização, versionamento e localização. No caso da *framework Lithium*, essas bibliotecas são geradas automaticamente em C#, mas podem ser produzidas em qualquer linguagem de programação.
REST API	Inclui uma *ASP.NET Core Web API*, *RESTful*, desenvolvida em *.NET 7*. Ela expõe os *endpoints* que podem ser acessados pelos consumidores do microsserviço.
Controllers	*Controllers ASP.NET Core* que disponibilizam todos os *endpoints* do microsserviço.

Models	Estruturas de dados geridas pelo microsserviço, também conhecidas como recursos. Esses modelos são desenvolvidos na camada da *API REST* e projetados para a biblioteca de cliente. Estes representam as informações que são trocadas entre a aplicação cliente e o microsserviço.
Host	Aplicação *ASP.NET Core* que hospeda a *API REST* do microsserviço. É responsável por iniciar e gerir a execução do microsserviço.

Tabela 1: Principais componentes microsserviço *Lithium*

É de notar que a *framework* fornece ainda a *Lithium Modeling Framework*, que permite modelar os *endpoints* e o comportamento do microsserviço. Isso possibilita a geração automática da maior parte do código de implementação necessário. Os *Models* e o código da *Rest Client* são gerados automaticamente, mas também podem ser personalizados. Quanto à API REST, os geradores de código produzem o esqueleto de todas as operações, deixando para os desenvolvedores a implementação da lógica de negócio.

Conforme os microsserviços se tornam mais complexos, a arquitetura elementar da Figura 14a pode não ser suficiente para atender a todos os requisitos de lógica de negócio. Por esse motivo, a partir da versão 3, o *Lithium* adotou a *Clean Architecture* como base para implementar microsserviços complexos. Como já referido no presente documento, este tipo de arquitetura permite lidar com a complexidade crescente, ao mesmo tempo em que mantém a modularidade e a flexibilidade.



(a) Arquitetura elementar

(b) Arquitetura complexa

Figura 14: Arquitetura Microsserviço *Lithium* [14]

Na Figura 14b está representada a arquitetura complexa onde foram introduzidas 3 novas camadas. A

camada de aplicação (*Application*), que é responsável por mapear as interações entre as diferentes partes do microsserviço. Para além disso, contém toda a lógica de negócio e respetivo domínio e faz a ponte entre as interfaces externas e os serviços internos. Relativamente ao acesso a bases de dados ou outro tipo de armazenamento, este é gerido na camada *Storage*. Por fim, o componente *Dependencies* fornece meios de comunicação com o mundo exterior, como por exemplo chamadas a *APIs* externas.

4 Microsserviços: Reporting Engine Service (RES)

Neste capítulo é apresentada a abordagem inicial da construção do Reporting Engine Service (RES), um conjunto de microsserviços capazes de atender às necessidades de *Reporting* de todos os produtos Cegid Primavera. Nas secções seguintes é esclarecida de forma detalhada a abordagem da solução a desenvolver, independentemente das tecnologias posteriormente adotadas.

4.1 Levantamento de Requisitos

Tendo em conta os objetivos da solução a desenvolver fez-se, em primeiro lugar, o levantamento dos requisitos funcionais e não funcionais do sistema. Esta recolha baseou-se maioritariamente nas necessidades dos desenvolvedores da Cegid Primavera, uma vez que o principal objetivo do software é, por um lado melhorar a eficiência das funcionalidades em questão, mas também centralizar e uniformizar todo o código das mesmas.

4.1.1 Requisitos Funcionais

RF1: O sistema disponibiliza a listagem dos *templates* de sistema.

RF2: O sistema disponibiliza a listagem dos *templates* de subscrição.

RF3: O sistema permite criar um *template* de subscrição a partir da edição de um *template* de sistema.

RF4: O sistema permite eliminar um *template* de subscrição.

RF5: O sistema permite editar um *template* de subscrição.

RF6: O sistema permite criar um *template* de subscrição.

RF7: O sistema apresenta ao utilizador uma interface para facilitar a edição de um *template*.

RF8: O sistema permite a impressão de um *report* do tipo lista a partir dos dados fornecidos.

RF9: O sistema permite a impressão de um *report* do tipo entidade a partir dos dados fornecidos.

4.1.2 Requisitos Não Funcionais

RNF1: O software não deve apresentar erros de funcionamento.

RNF2: O software deverá ser implementado em .NET e C#, tecnologias utilizadas pela Cegid Primavera.

RNF3: O armazenamento de dados deve recorrer aos serviços da *Microsoft Azure*.

RNF4: O software deve estar adaptado de modo a ser implementado no ecossistema da Cegid Primavera.

RNF5: O software deve estar protegido por autenticação e autorização segundo as regras de contratualização dos clientes da Cegid Primavera.

RNF6: Todo o código desenvolvido no software deve ser comentado segundo as normas de boas práticas de programação da Cegid Primavera.

4.2 Use Cases

De modo a identificar as principais funcionalidades do sistema, e atendendo aos requisitos funcionais previamente listados, elaborou-se um diagrama de *use cases*. Na Figura 15 estão devidamente identificadas as principais ações que o utilizador pode executar no sistema. É de notar que, apesar de não estar representado no diagrama, todas as funcionalidades requerem previamente autenticação e autorização do utilizador em questão.

Assim sendo, o sistema desenvolvido conta com processos no âmbito da gestão de *templates*, dando liberdade ao utilizador para criar, eliminar, obter um template específico e listar todos aqueles aos quais o utilizador tem acesso.

Do mesmo modo, pretende-se também possibilitar a edição de um template através da disponibilização de uma interface gráfica simples e clara com intuito de ser manuseada por utilizadores com conhecimentos tecnológicos elementares.

Por fim, a funcionalidade principal do sistema consiste na impressão de um *report*, que através de um template especificado e de uma definição de dados permite gerar o documento resultante. É importante realçar que, para cumprir os requisitos da Cegid Primavera, distinguiram-se dois tipos de impressão: listas e entidades. A impressão de uma entidade envolve a devida identificação de cada campo específico da

mesma, sendo o template associado um componente sobretudo estático. Por outro lado, a impressão de listas permite representar os dados resultantes da obtenção de uma listagem dos elementos de uma determinada entidade. Assim sendo, a implementação desta funcionalidade deve ser projetada de modo a generalizar a impressão, independentemente da entidade que estiver a ser listada.

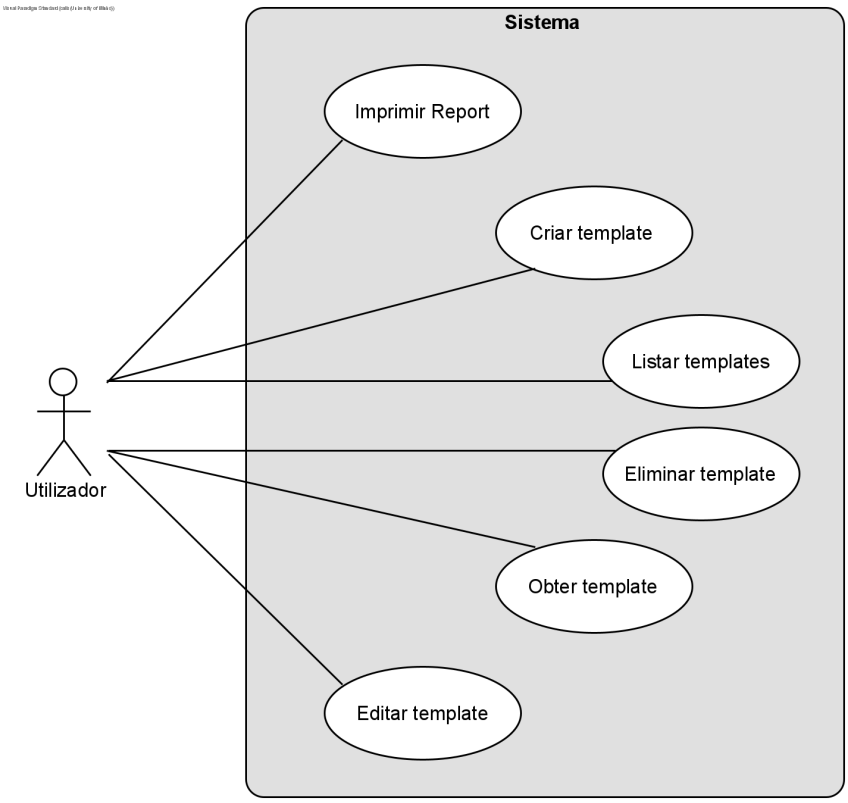


Figura 15: Diagrama de Use Cases

4.3 Modelo de Domínio

Para além dos requisitos e use cases, houve a necessidade de representar o sistema através de um modelo de domínio. Desta forma, é possível esclarecer alguns conceitos utilizando uma linguagem mais natural, sendo acessível tanto a profissionais da área como a pessoas comuns.

Como se pode verificar na Figura 16, partindo dos dados de impressão, que são todas as informações que o utilizador pretende representar no *Report*, estes terão de ser formatados para uma estrutura de dados bem definida, isto é um contrato. Para ser possível generalizar alguns modelos de *report*, diferenciaram-se dois tipos: entidades e listas. Um contrato de entidade inclui todos os campos necessários à impressão de um *report* com um template bem definido, não sendo possível editar dinamicamente

qualquer secção do mesmo. Por outro lado, um contrato de listas, para além dos campos estáticos contém ainda um conjunto de informações, a lista da definição de cada coluna que pretende especificar o formato da tabela onde são listados posteriormente os dados pretendidos. Desta forma, serão assim adicionadas dinamicamente as secções ao template selecionado.

Para ser possível a geração de um *report*, é necessário recorrer a uma ferramenta de *Reporting* que, utilizando um *template* e a definição de dados presente no contrato é capaz de imprimir o *Report* com as informações pretendidas.

Para além da impressão o sistema permite também ao utilizador a gestão dos seus *templates* guardados através de um *Designer*, onde pode guardar, editar e eliminar os mesmos.

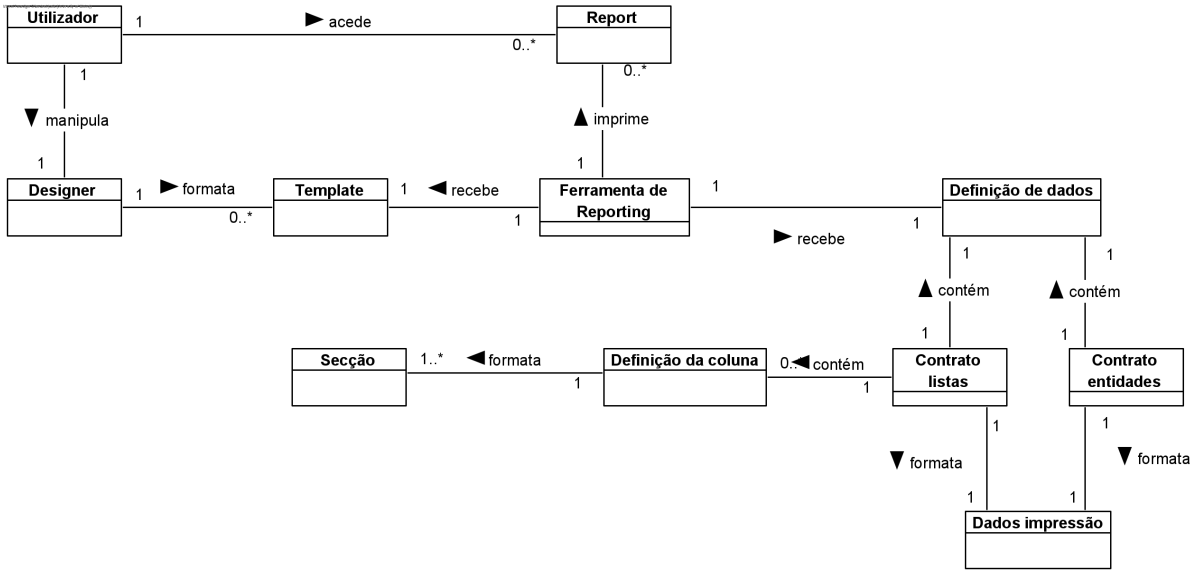


Figura 16: Modelo de domínio

4.4 Diagrama de Componentes

Para estabelecer uma visão mais alto nível do sistema representou-se o diagrama de componentes da Figura 17.

O componente principal RES engloba toda a lógica de negócio do sistema, que por sua vez é constituído por 3 subsistemas:

- **Printing:** compreende toda a lógica de impressão, ou seja, a transformação de dados para um *Report*.

- **TemplateManager:** engloba todas as ações de gestão dos templates armazenados.
- **Designer:** contém as funcionalidades necessárias que permitem disponibilizar uma interface simples ao utilizador para que possa editar as secções de um template.

Por outro lado, o *Designer* inclui também um componente de apresentação *DesignerView*, que engloba todos os elementos necessários ao desenho da interface para o utilizador.

Relativamente à persistência de dados está representada no componente *Persistence*, e inclui todas as ações necessárias ao acesso e gestão dos *templates* guardados de forma persistente. Por sua vez, o acesso a este componente é disponibilizado através da respetiva interface *IPersistence*. Assim sendo, é de notar que somente o subsistema *TemplateManager* tem permissão para aceder ao mesmo.

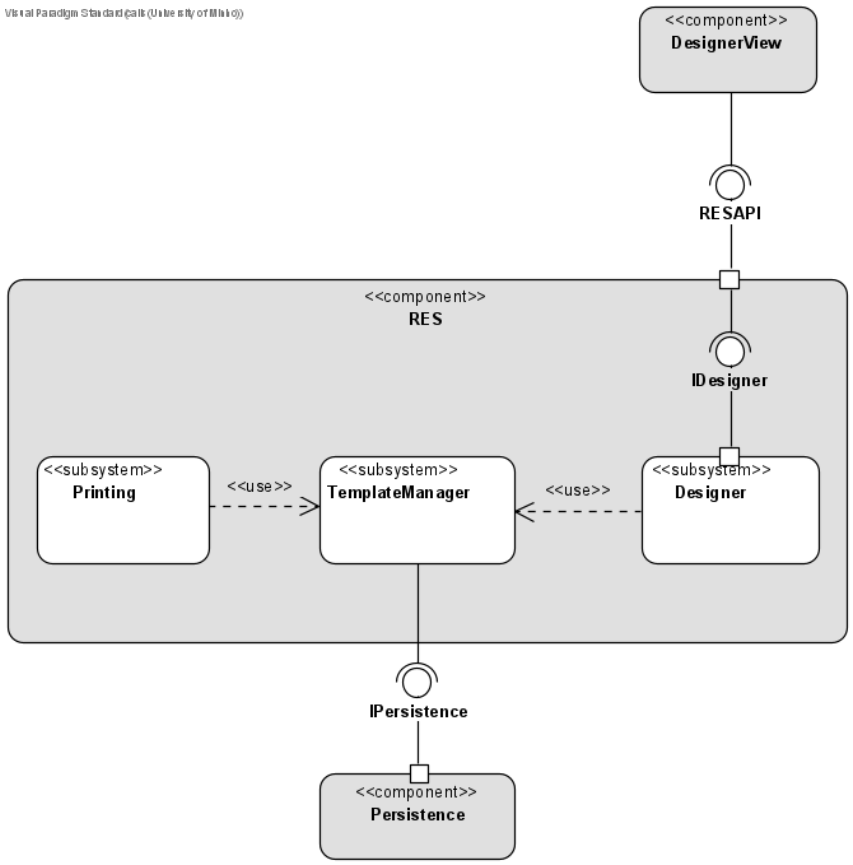


Figura 17: Diagrama de Componentes

4.5 Componentes e suas funcionalidades

A convergência da arquitetura previamente definida com o diagrama de componentes resultou numa representação visual abrangente e estruturada do sistema em questão. No diagrama da Figura 18, é possível observar de forma clara e concisa como os diversos elementos interagem e se relacionam para criar uma base sólida que sustentará a implementação do microsserviço.

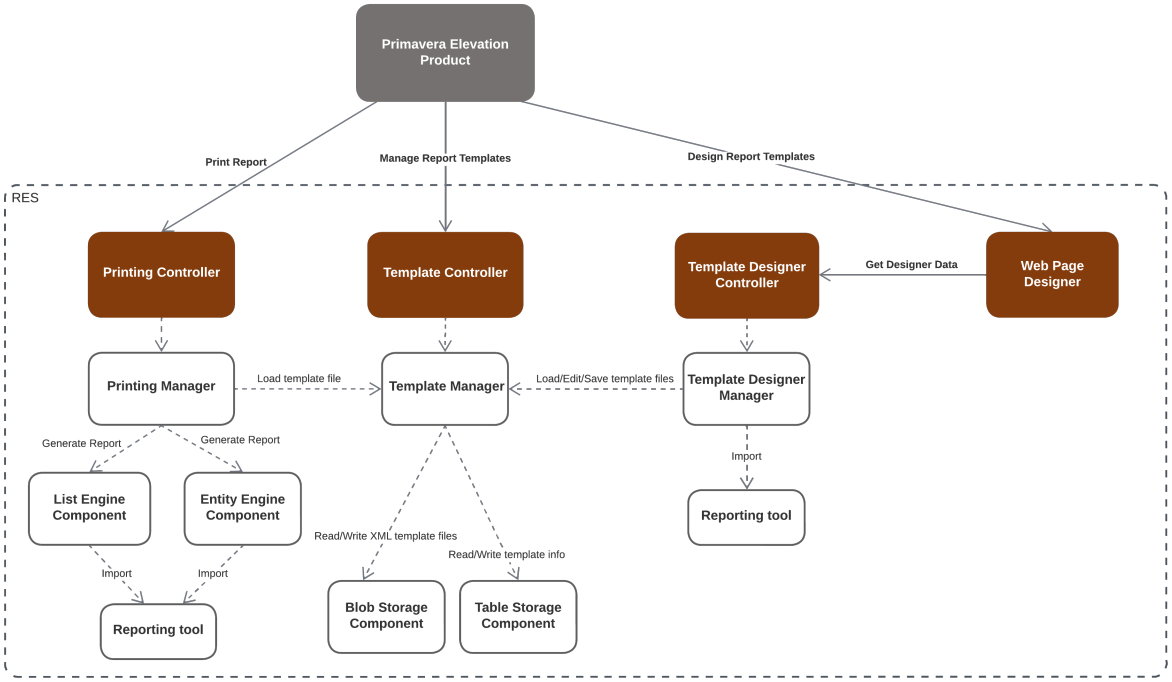


Figura 18: Representação de alto nível das responsabilidades de cada componente

Como se pode verificar, toda a lógica de negócio é englobada numa API, que disponibiliza os *endpoints* através de 3 *controllers* diferentes. Tal como referido no diagrama de componentes, e fazendo a respetiva correspondência, cada *controller* contém os métodos necessários para a impressão de documentos (*Printing Controller*), gestão de *templates* (Template Controller) e design de *templates* (*Template Designer Controller*). Por sua vez, cada *Controller*, invoca o respetivo método ao *Manager* encaminhando os dados necessários.

No que toca à gestão dos *templates* guardados, o *Template Manager* pode aceder ao *blob storage* onde estão armazenados todos os ficheiros dos *templates* em formato *xml*. Para gerir as permissões, localização e outros meta-dados de cada template existe também um armazenamento em tabelas *NoSQL* (*Table Storage*).

Relativamente à impressão de documentos, tal como referido no modelo de domínio, é feita a distinção entre impressão de listas e entidades, sendo que cada uma é suportada por um *Engine*. O *Engine* tem a responsabilidade de, com o auxílio das bibliotecas da ferramenta de *reporting*, efetuar ajustes nos campos do modelo predefinido, se necessário. Para além disso, é também encarregue de fornecer os dados destinados à impressão e, por fim, criar o *report* final resultante. O template fornecido ao *Engine* é previamente carregado fazendo a comunicação com o *Template Manager*.

Por fim, o *designer* de *templates* é disponibilizado ao utilizador via página web, suportada por uma aplicação cliente onde é apresentada uma IU simplificada que permite aos usuários criar, editar e eliminar os seus *templates*. Por sua vez, as ações do lado do servidor são geridas pelo *Template Designer Controller*, que invoca os métodos do *Manager* correspondente. Tal como para a impressão, o *Designer* obtém os *templates* através do *Template Manager*, sendo que poderá modificar e até eliminar aqueles que forem permitidos ao utilizador em questão.

5 RES - Abordagem e desenvolvimento da solução

Tendo bem definidas tanto as funcionalidades a desenvolver, como as entidades do domínio e relações entre elas, constituiu-se uma base sólida de conhecimento para avançar para a fase da implementação da solução. Desta forma e considerando a arquitetura estabelecida, nesta fase foi tomada a decisão de incluir toda a lógica do RES num único microsserviço, uma vez que a sua complexidade não exige uma maior divisão.

5.1 Arquitetura

Após o estudo aprofundado das diversas arquiteturas existentes e tendo em conta as inúmeras vantagens que a *framework Lithium* oferece no desenvolvimento de microsserviços foi incontestável a sua escolha para a implementação do RES. Entre as principais razões que motivaram esta escolha estão:

- **Estrutura Modular:** O *Lithium* adota uma estrutura modular que se alinha perfeitamente com os princípios da arquitetura microsserviços. Cada serviço pode ser desenvolvido de maneira independente, com suas próprias funcionalidades e interfaces.
- **Padrão MVC:** A adoção do padrão *Model-View-Controller* (MVC) facilita a separação de responsabilidades e a organização do código, essencial para manter a clareza.
- **Flexibilidade de Dados:** A capacidade de integrar diferentes sistemas de base de dados de forma eficaz é fundamental em um ambiente de microsserviços, onde diferentes serviços podem requerer diferentes soluções de armazenamento.
- **Rotas e Pedidos:** A capacidade do *Lithium* de lidar com rotas e encaminhamento de pedidos permite a construção de *APIs* robustas e eficientes para cada serviço.

Assim sendo, o desenho da arquitetura do microsserviço (Figura 19) baseou-se, tanto na arquitetura base da *framework*, *Clean Architecture*, como no padrão *Model-View-Controller* (MVC). Tendo em conta o

diagrama funcional previamente elaborado foi feita naturalmente a associação dos elementos pertencentes a cada camada.

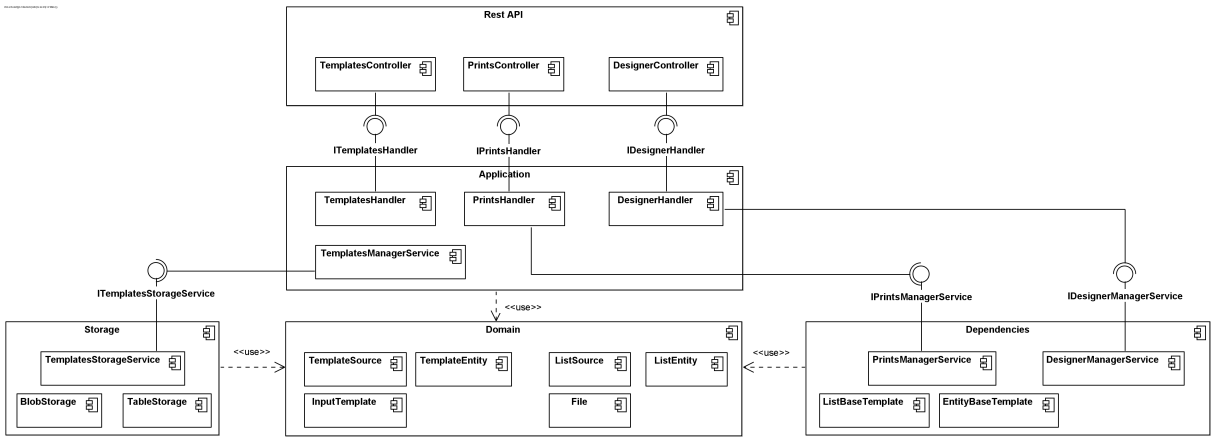


Figura 19: Arquitetura do RES

Relativamente à camada mais superficial, a Rest API, esta inclui os 3 *controllers*, sendo que cada um agrupa as rotas e respetivos métodos do seu domínio. Na Figura 19, observando as dependências entre os componentes, é possível identificar o domínio pertencente a cada um, sendo eles a gestão de *templates*, impressão de *reports* e o desenho de *templates*

Na camada *Application* existe um *Handler* que, seguindo o padrão *CQRS*, separa as operações de leitura e escrita. Para além disso, esta classe é também responsável por validar os dados recebidos e tratar das exceções de cada método. Por sua vez, cada *Handler* invoca os métodos necessários que estão implementados nos respetivos *Managers*. É importante realçar que tanto o *PrintsManagerService*, como o *DesignerManagerService* estão incluídos na camada de dependências (*Dependencies*), uma vez que fazem uso das bibliotecas da ferramenta de *reporting*. Isto significa que, no futuro, caso seja necessário mudar de ferramenta as alterações apenas serão maioritariamente feitas ao nível desta camada.

A camada *Domain* agrupa todas as estruturas de dados elementares de cada domínio utilizadas tanto na gestão de *templates* como na impressão de *reports*.

Por fim, a camada *Storage* contém toda a lógica de acesso ao armazenamento de dados. É de notar que no RES são utilizados dois tipos distintos de armazenamento suportados pelo *Lithium*:

- **BlobStorage:** Sistema de armazenamento na *cloud*, semelhante a um sistema de arquivos, otimizado para armazenar dados brutos (bytes), texto, imagens e arquivos. Todos os *templates* geridos pelo microsserviço serão armazenados neste sistema, em formato *xml*.

- **TableStorage:** Armazenamento estruturado em tabelas *NoSQL*, otimizado para armazenar dados com o formato chave/valor. Nestas tabelas são armazenados alguns meta-dados de cada template, como por exemplo o ID da subscrição que lhe tem acesso.

Em suma, com a adoção desta arquitetura pode-se garantir a conceção de microsserviços que atendem às necessidades atuais, mas que também estão preparados para evoluir e adaptar-se a mudanças tecnológicas no futuro.

5.2 Diagramas de Sequência

No contexto do desenvolvimento de software, os diagramas de sequência desempenham um papel crucial na representação das interações dinâmicas entre os diferentes módulos, componentes e atores envolvidos. Estes diagramas permitem oferecer uma visão detalhada do fluxo de eventos e mensagens que ocorrem durante a execução de diversas funcionalidades e processos-chave do sistema. Por este motivo, antes de passar à implementação da solução, considerou-se essencial elaborar alguns diagramas de sequência para representar as funcionalidades mais complexas do microsserviço.

5.2.1 Impressão de Listas

A principal funcionalidade e, por sua vez, mais complexa num serviço de *reporting* corresponde à impressão de um *report*. É de notar que, numa primeira fase de desenvolvimento do RES, deu-se especial foco à impressão de listas, isto porque é uma operação muito comum nos produtos Primavera e onde existe uma maior capacidade para generalizar devido às caraterísticas comuns que se pretende.

É importante notar que num *report* deste tipo existem algumas secções opcionais que são previamente definidas e devidamente identificadas no template. Esses campos incluem os dados da empresa como nome ou número de identificação fiscal, os filtros que foram aplicados à listagem e o campo de procura, caso tenha sido feita uma filtragem por palavra. Para além disso, o utilizador pode ainda acrescentar ao template outros campos fixos como a paginação, data, entre outros.

O fluxo da impressão de uma lista está representado no diagrama da Figura 20, sendo que como ator foi selecionado o *Jasmin*, um dos produtos *Elevation* da Cegid Primavera que requer o consumo de funcionalidades de *reporting*. Desta forma, de acordo com a API disponibilizada pelo microsserviço RES, é feito um pedido *POST* fornecendo as informações necessárias de autorização e autenticação e os respetivos dados de impressão.

Em primeiro lugar, o pedido é intercetado pelo *Controller* do RES, que após fazer as devidas verificações dos dados recebidos encaminha o mesmo para a camada aplicacional onde estão centradas todas as ações a executar. Assim sendo, o comando *PrintList* verifica inicialmente o campo *culture* que é um parâmetro opcional da API do microsserviço. Se esta informação não for fornecida, o *report* será impresso na língua padrão do sistema, caso contrário o mesmo será convertido para a língua especificada.

De seguida, recorrendo ao *Template Manager* é carregado o ficheiro *xml* do template previamente identificado no pedido, tendo em conta que caso o ficheiro não exista é lançada uma exceção. Por fim, tendo o template e os dados a responsabilidade de impressão pertence ao *Printing Manager*.

Para gerar um *report*, o template, inicialmente carregado em memória é convertido na classe correspondente à ferramenta de *reporting* utilizada. Desta forma, torna-se possível manipular as secções pretendidas.

Assim sendo, em primeiro lugar é aplicada a *culture* fornecida. O RES permite ao utilizador fornecer uma lista variável de filtros, portanto são adicionadas em tempo de execução as colunas correspondentes ao template. De seguida, é verificada a existência dos campos opcionais que, caso sejam nulos, são retirados do template. Consequentemente, de modo a proporcionar uma melhor apresentação são ajustadas todas as posições com o intuito de ocupar o espaço vazio deixado. No que toca a alterações ao template, à semelhança dos filtros, é definida a tabela principal dos dados ajustando as colunas e suas dimensões à entidade a ser listada.

Tendo completado todos os ajustes ao template, é então fornecida a *datasource* e feita a impressão convertendo o *report* para formato *pdf*.

5.2.2 Obter um template

Uma das funcionalidades mais utilizadas no RES será a obtenção de um template. Como se verificou na elaboração dos diagramas anteriores, esta ação é necessária, tanto para a impressão de um *report*, como também para a edição de um template através do designer. Por este motivo, considerou-se fundamental estudar a melhor abordagem de modo a tornar a funcionalidade eficiente e escalável. Na Figura 21 está representado o comportamento da obtenção de um template, neste caso carregado diretamente via *API*, no entanto, tal como referido é também acessado outros componentes dentro do microsserviço.

Assim sendo, o pedido *GET* é recebido no *Controller* responsável pela gestão de *templates* (*TemplatesController*), que invoca imediatamente a *query* correspondente (*TemplatesGet*). Após ter sido feita a devida verificação de possíveis campos nulos, cabe ao *TemplateManager* certificar, em primeiro lugar, a existência do template com o nome especificado, e caso contrário retornar uma exceção. Para efetuar

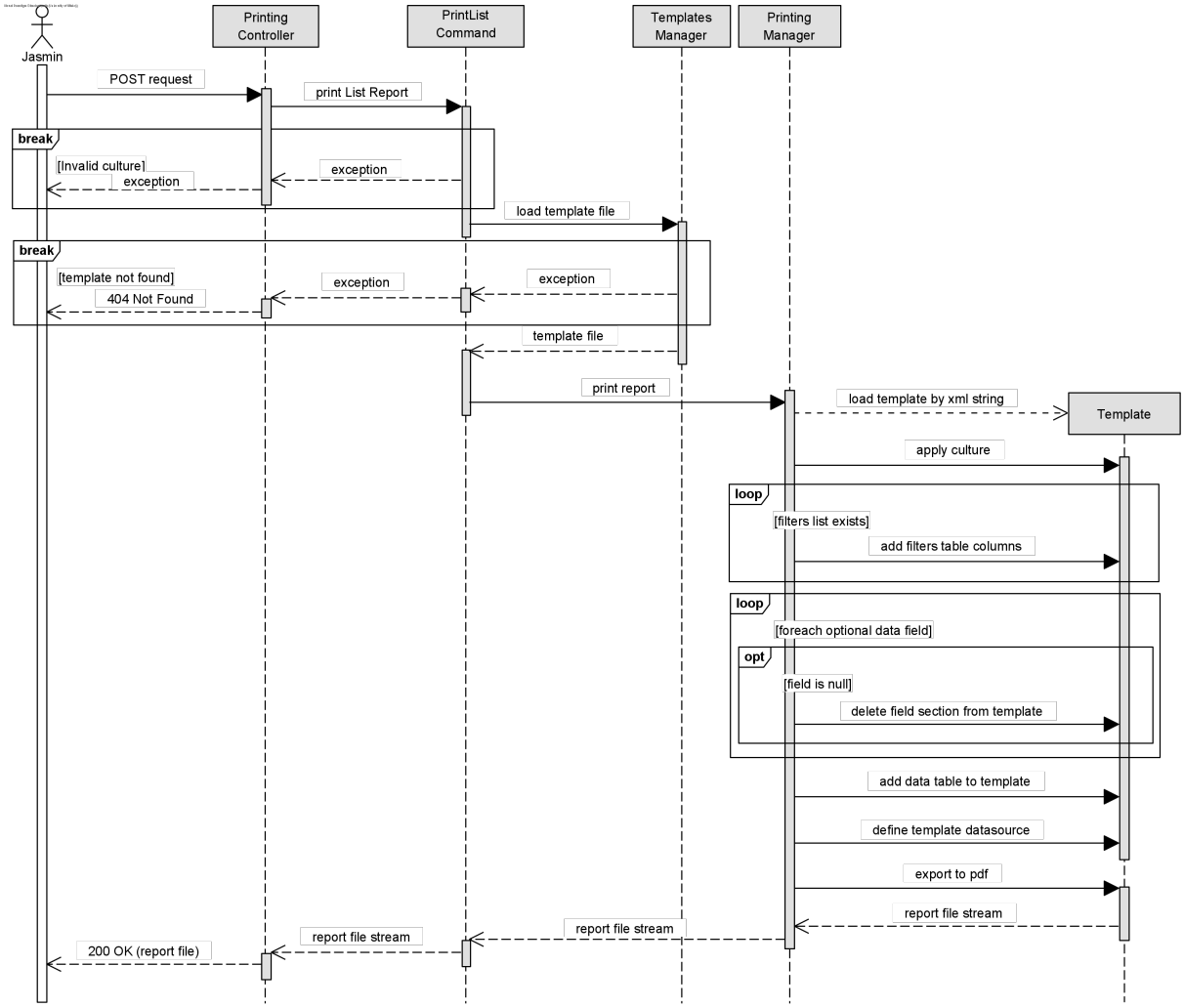


Figura 20: Diagrama de Sequência - Impressão de listas

esta validação é acedida uma tabela de dados (*Table Storage*), onde estão especificadas as informações necessárias respetivas a cada template guardado, como o seu nome, ou dados de acesso ao mesmo.

Após obtidas as informações do *template* (*Template Entity*), o ficheiro *xml* correspondente, armazenado num *Blob Storage*, é carregado e retornado como resposta.

5.3 Desenvolvimento da Solução

Nesta secção, é abordado em detalhe o processo de implementação da solução, explorando as metodologias, estratégias e tecnologias adotadas para garantir a concretização da visão inicial.

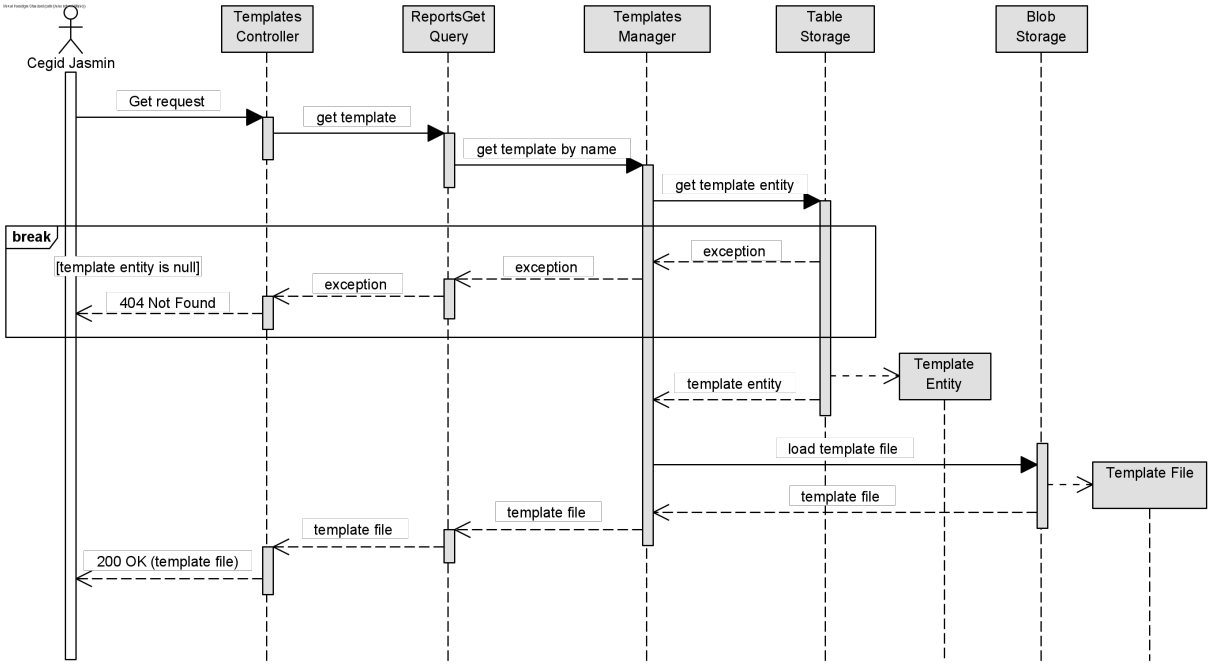


Figura 21: Diagrama de Sequência - Obter template

5.3.1 Autenticação e Autorização

A autenticação e autorização são componentes fundamentais em qualquer estratégia de segurança de dados e controlo de acesso. Estas duas práticas desempenham um papel fundamental na preservação da integridade e confidencialidade das informações, estabelecendo uma barreira contra acessos não autorizados. Em conjunto, estas medidas garantem que apenas utilizadores com credenciais apropriadas e permissões necessárias possam interagir com recursos e dados sensíveis.

O microsserviço RES dispõem de dois serviços distintos para gerir o acesso ao mesmo. No que toca à autenticação, este faz uso do *Identity Server* no entanto, a autorização é assegurada pelo microsserviço *Juice Apps Service*.

Identity Server

O *Identity Server* é uma solução de gestão de identidade e acesso (*Identity and Access Management - IAM*) projetada e mantida pela Cegid Primavera para atender aos requisitos de *OAuth 2.0* e *OpenID Connect (OIDC)* em todos os seus produtos, tanto soluções *cloud* como locais. A abordagem de aproveitar os padrões mencionados, juntamente com tecnologias de ponta, como *.NET Core, ASP.NET Core e Duende Identity Server*, teve como principal objetivo garantir uma fácil adaptação a futuras evoluções.

Esta solução foi adaptada e configurada de acordo com as necessidades da Cegid Primavera, tendo

em conta os diferentes cenários do modelo de negócio implementado. Assim sendo, do ponto de vista da Cegid Primavera oferece vantagens como a centralização da autenticação de utilizadores e login único (*single sign-on*) em todos os produtos, uma implementação de IAM extensível e personalizável, com pontos de extensão para suportar outros protocolos no futuro e ainda um *back-office* para configurar e operar o sistema. Por outro lado, em relação ao ponto de vista do utilizador final o *Identity Server* permite uma experiência única de login e logout entre todos os produtos, a integração com fornecedores de identidade populares como Apple, Google, Microsoft, etc., sendo que permite também a associação com provedores personalizados.

No geral, o *Identity Server* oferece uma proposta de valor abrangente que beneficia a empresa, os usuários finais e terceiros, proporcionando segurança, conveniência e flexibilidade na gestão de identidade e acesso.

A arquitetura alto nível do *Identity Server* é constituída por 4 componentes principais:

- **Core Endpoints:** *Endpoints* que implementam os protocolos OAuth e OpenID Connect para autenticação e autorização.
- **Front-office:** Abrange a interface de *login* dos utilizadores finais e uma plataforma web que permite a gestão das suas contas pessoais.
- **Back-office:** Plataforma web que permite a gerentes e colaboradores configurar e gerir todo o sistema. Isto permite uma administração eficaz da solução.
- **REST API:** Fornece acesso aos recursos do *Identity Server*. Isto é útil para integrações com outros sistemas e para aceder a informações sobre utilizadores e políticas.

Assim sendo, o *Identity Server* pode ser utilizado tanto pelos *Resource servers* para proteger os seus recursos, como autenticar aplicações cliente através de *tokens* de acesso.

A incorporação do *Identity Server* no microsserviço RES foi implementada com auxílio da *framework Lithium*. Como resultado, esta ligação proporcionou à solução final uma centralização e uniformização de todo o processo de autenticação.

Juice Services

A plataforma Primavera Juice é um sistema distribuído composto por vários microsserviços, todos com o objetivo comum de dar suporte ao ciclo de vida das aplicações *cloud* da Cegid Primavera, como *Jasmin* e *Rose*. Na Figura 23 estão representados todos os serviços que disponibiliza sendo que cada um serve um propósito funcional distinto:

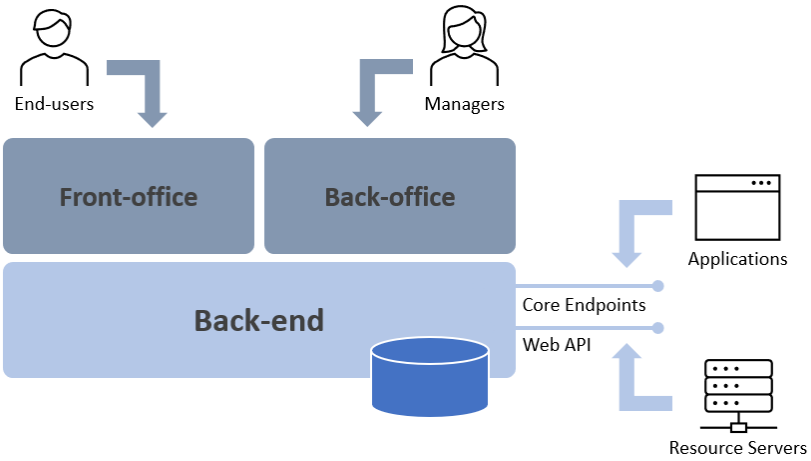


Figura 22: Componentes do Identity Server [9]

- **Billing Service:** lida com a gestão de clientes que utilizam serviços em nuvem da Cegid Primavera e os contratos associados a esses clientes.
- **Apps Service:** concentra-se na gestão das aplicações *cloud* da Cegid Primavera e suas configurações. Isso permite às empresas personalizar e ajustar suas aplicações conforme necessário.
- **Provisioning Service:** gere a infraestrutura *cloud* e as operações de aprovisionamento. Isso garante que as aplicações estejam prontas para uso de forma eficiente.
- **Usage Service:** responsável pela monitorização do uso das aplicações *cloud* da Cegid Primavera, permitindo que as empresas controlem e otimizem os seus recursos de acordo com as métricas calculadas.

Para melhor entender a metodologia de autorização efetuada com recurso ao *Juice* é importante esclarecer o fluxo após a contratualização de um produto *cloud* Primavera com um cliente. Depois de um cliente adquirir o produto, a equipa responsável da Cegid Primavera efetua a criação da respetiva subscrição através do serviço de *billing*. Este, por sua vez, comunica com o *Apps Service* para adicionar a nova subscrição à base de dados e permitir o acesso à mesma do novo cliente. Entretanto este serviço faz ainda uma chamada ao *Provisioning Service* para que seja feito o aprovisionamento da instância do produto de acordo com os recursos pretendidos e acordados com cliente.

Relativamente à estrutura das subscrições, estas seguem uma organização hierárquica em árvore (Figura 24), estando obrigatoriamente inseridas em 3 patamares: *workspace*, *organization* e *appinstance*.

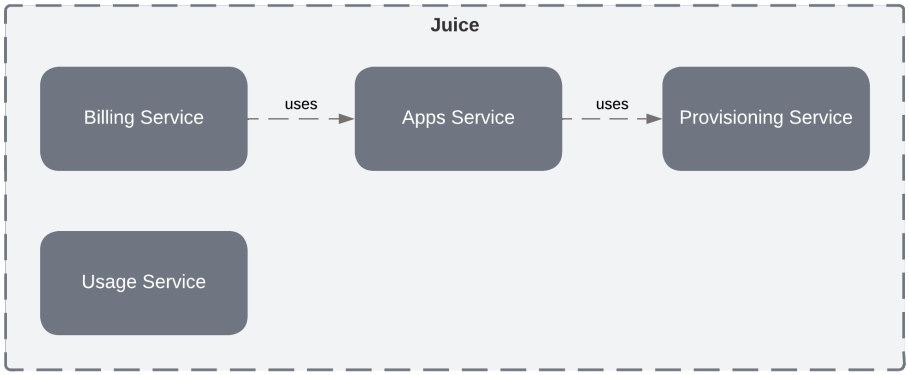


Figura 23: Microsserviços Juice [13]

Esta distribuição possibilita a autorização de acesso a utilizadores, tanto a uma subscrição única, como a um conjunto englobado por uma *organization* ou até um *workspace*.

Tendo terminado todo o fluxo de criação de uma subscrição, o acesso de um utilizador é verificado através do *Apps Service*, que trata de se certificar que o mesmo pode aceder a uma determinada instância do produto.

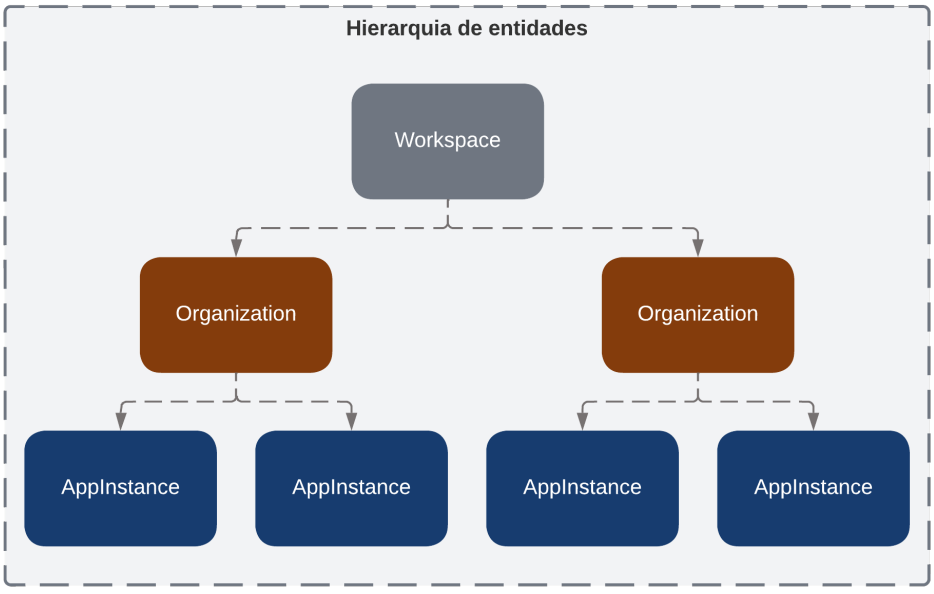


Figura 24: Apps Service - Hierarquia de subscrição [12]

A utilização de uma arquitetura de microsserviços distribuídos como o *Primavera Juice* permite uma maior flexibilidade, escalabilidade e especialização ao nível de cada funcionalidade do sistema em ques-

tão. Isto é fundamental para atender às complexas exigências que a gestão de aplicações em nuvem proporciona e oferecer suporte eficaz aos clientes da Cegid Primavera.

5.3.2 Armazenamento de Dados

Os microsserviços, como qualquer aplicação, requerem frequentemente a capacidade de armazenar dados. Sendo o RES um serviço que permite a criação, manipulação e exclusão de *templates* definiram-se-se dois tipos de armazenamento: *Blob* e *Table Storage*. É de notar que ambos são suportados pelo *Lithium*, o que facilitou a sua implementação.

Blob Storage

O armazenamento de *blob*, abreviação de *Binary Large Object* (Objeto Binário Grande) é um sistema de armazenamento semelhante a um sistema de ficheiros, otimizado para armazenar diversos tipos de dados não estruturados, incluindo dados brutos (bytes), texto, imagens e arquivos.

Este tipo de armazenamento é muito utilizado em plataforma em nuvem por oferecer várias caraterísticas vantajosas como:

- **Escalabilidade:** possibilidade de armazenar grandes quantidades de dados. Os serviços oferecem uma capacidade virtualmente ilimitada, sendo que só é pago o armazenamento em utilização, tornando-se assim uma solução económica para lidar com volumes crescentes de dados.
- **Redundância e Fiabilidade:** oferece redundância e replicação de dados em vários servidores, garantindo a durabilidade e a disponibilidade dos dados, mesmo em caso de falhas de hardware.
- **Controlo de Acesso:** oferece mecanismos de controlo de acesso detalhados, permitindo definir quem pode ler, gravar ou excluir dados armazenados.

Tendo o RES a necessidade de armazenar vários templates em formato *xml*, o *blob storage*, atendendo às vantagens que proporciona, foi o tipo de armazenamento selecionado. Mais concretamente, uma vez que a Cegid Primavera usufrui do ecossistema de serviços da Microsoft, recorreu-se ao *Azure Blob Storage*.

A organização no armazenamento de *templates* (Figura 25) implicou uma distinção entre dois tipos: de sistema e de subscrição. Os *templates* de sistema estão agrupados num *blob container* e são aqueles que estão disponíveis inicialmente para qualquer subscrição. Estes possuem caraterísticas particulares como a impossibilidade de serem alterados ou eliminado pelo utilizador final. Por sua vez, a edição de um template de sistema pelo utilizador, através do designer, origina a criação de um novo template de

subscrição ao qual apenas tem acesso usuários dessa mesma subscrição. Este tipo de template pode, no entanto, ser editado ou eliminado, pelos utilizadores que lhe têm acesso. Os *templates* de uma subscrição são guardados num *blob container* exclusivo da mesma.

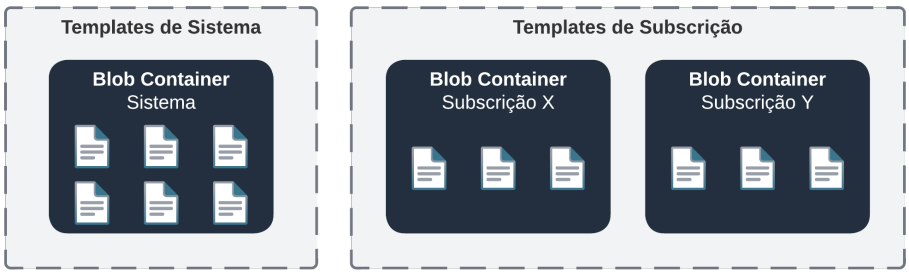


Figura 25: Armazenamento de templates

Table Storage

O *Azure Table Storage* é um serviço de armazenamento *NoSQL* oferecido pela *Microsoft Azure*, projetado para armazenar dados em formato de tabela otimizada para consulta eficiente. Ao contrário das bases de dados relacionais tradicionais, o *Table Storage* segue um modelo de chave-valor, onde os dados são indexados por meio de duas colunas principais: a chave de partição (*partition key*) e a chave de linha (*row key*).

Após escolher a solução de armazenamento de *templates* entendeu-se ser necessário o armazenamento de informações adicionais e ainda de alguns dados redundantes para aumentar a eficiência dos acessos. Por este motivo, recorreu-se ao *Table Storage* principalmente devido à sua flexibilidade pelo facto de seguir um modelo *NoSQL*, mas também pela escalabilidade, desempenho e eficiência de consulta graças à indexação.

Assim sendo, cada linha da tabela corresponde a um template tendo em consideração que, a *partition key* identifica a subscrição e a *row key* o nome do template. Com estas duas chaves é possível aceder a todas as informações guardadas relativas ao template em questão. Com estes dados é possível, por exemplo, otimizar a verificação da existência de um template evitando assim o tempo prolongado de carregar o ficheiro.

5.3.3 Ferramenta de Reporting: DevExpress

Como já evidenciado no capítulo do estado da arte, foi realizado um estudo aprofundado das ferramentas *reporting* disponíveis no mercado. A análise das opções disponíveis e respetiva comparação e dos requisitos do microsserviço RES levou à escolha da ferramenta *DevExpress*. O facto de oferecer uma base sólida de suporte, graças à sua completa documentação e a capacidade para suportar tecnologias atualizadas foram as principais razões que motivaram esta escolha.

Assim sendo, as funcionalidades de impressão recorreram sobretudo ao *namespace DevExpress.XtraReports.UI* [21]. Esta biblioteca inclui classes como a *XtraReport* que serviu de base para carregar o ficheiro *xml* de um template, editar as secções necessárias em tempo de execução e imprimir o *report* correspondente. A manipulação de um *XtraReport* envolveu o uso de classes como a *XRLabel*, que representa uma secção elementar de texto do template e ainda a *XRPanel* que permitiu agrupar *XRLabel's* de um determinado tipo, por exemplo as informações dos detalhes da empresa.

O formato escolhido para representar a *datasource* de um template foi o *json*. Esta escolha teve em conta a maior capacidade de generalização e flexibilidade em relação ao tipo de dados fornecidos, assim como a facilidade de conversão no que diz respeito aos dados recebidos pela *API REST*. Graças ao *namespace DevExpress.DataAccess.Json* foi possível providenciar a estrutura de dados ao *XtraReport* em tempo de execução, para posteriormente realizar a impressão.

Por outro lado, numa versão inicial do RES utilizou-se o *ASP.NET Core Report Designer* da *DevEpress* para permitir a criação e edição de *templates* pelos utilizadores finais. Este *Designer* oferece uma interface relativamente simplificada onde o usuário é capaz de modificar as variadas secções de um template e permite também especificar diferentes *templates* para linguagens distintas.

5.3.4 Funcionalidades implementadas

As funcionalidades representam a parte visível e interativa do projeto, sendo a ponte que conecta a estrutura técnica ao usuário final. Ao longo do presente projeto, cada uma delas foi projetada com base nos requisitos levantados, e a sua implementação permitiu concretizar os objetivos propostos inicialmente.

Assim sendo, as funcionalidades implementadas estão representadas na Tabela 2, onde se encontram listados todos os *endpoints* disponibilizados pela *API* do RES. No que diz respeito às rotas, como se pode verificar considerou-se essencial existir uma separação entre as ações de cada *controller*: impressão, gestão de *templates* e *designer*.

Por outro lado, é importante destacar que todos os *endpoints* relativos ao designer foram implemen-

tados recorrendo às classes base das bibliotecas da *DevExpress*, nomeadamente a *ReportDesignerController* que tem como objetivo processar todos os pedidos da aplicação cliente, o *Web Report Designer*.

Método	Endpoint	**Descrição**
POST	/api/v1/subscritionPath[1]/templates	Criar um template
GET	/api/v1/subscritionPath[1]/templates	Obter todos os templates disponíveis para a subscrição
DELETE	/api/v1/subscritionPath[1]/templates/{name}	Elimina o template especificado
GET	/api/v1/subscritionPath[1]/templates/{name}	Obter a definição do template especificado
POST	/api/v1/subscritionPath[1]/prints/list	Impressão de um *report* do tipo lista
POST	/api/v1/subscritionPath[1]/prints/entity	Impressão de um *report* do tipo entidade
POST	/DXXRD	Lida com os pedidos da aplicação cliente do *designer*.
POST	/DXXQB	Permite criar *queries* e *views* para uma fonte de dados.
GET | POST	/DXXRDV	Permite visualizar um *report* com uma fonte de dados predefinida.

Tabela 2: Reporting Engine Service Endpoints

[1] subscritionPath: {product}/{workspace}/{appInstance}

Fazendo uma análise aos métodos implementados, em relação à gestão de *templates* é importante notar que um utilizador, para além dos *templates* de sistema disponíveis, é apenas capaz de obter *templates* correspondentes à sua subscrição. Do mesmo modo, a exclusão dos mesmos é restrita aos *templates* pessoais, isto é, não é possível eliminar *templates* de sistema, assim como modificar os mesmos. Como já referido anteriormente, a modificação de um *template* de sistema implica a criação de um novo correspondente à subscrição em uso.

No que diz respeito às funcionalidades de impressão, houve uma distinção entre a impressão de listas e entidades. Numa fase inicial, teve-se mais em conta a implementação da primeira e, por este motivo, a impressão de entidades acabou por ser considerada um trabalho futuro, a desenvolver consoante o

sucesso do projeto em questão. Assim sendo, a impressão de listas foi desenhada, tal como pretendido, de maneira atender às necessidades de qualquer produto que pretenda obter o documento correspondente à lista de dados fornecida.

Por fim, para a implementação do designer, que permite a manipulação de *templates* pelo utilizador final, utilizou-se o *Web Designer* predefinido da *DevExpress*. O desenvolvimento deste componente envolveu alguns ajustes fundamentais, nomeadamente a devida identificação da subscrição em uso, para a correta gestão de *templates*. Do mesmo modo, tal como definido na arquitetura da solução, toda a administração de *templates* é da responsabilidade do *TemplateManager*, pelo que todo o acesso e modificação dos mesmos através designer foi devidamente mediado por esse componente.

Com a concretização das funcionalidades mencionadas permitiu-se obter um conjunto de métodos que permite atender a todas necessidades de *reporting* dos produtos *cloud* da Cegid Primavera.

6 Integração do RES com Produtos Elevation

Tal como já foi referido ao longo do documento, a origem do projeto surgiu da necessidade de uniformizar e centralizar as funcionalidades de *reporting* dos produtos *cloud* da Cegid Primavera.

A *framework Elevation*, através do módulo de *Reporting* que implementa, oferece no momento alguma uniformização no que diz respeito a este âmbito. No entanto, o facto de seguir as caraterísticas de um monólito, pela imposição de estar sempre agregada a um produto, revela inevitavelmente os problemas associados a este tipo de arquitetura.

Tendo concluído o desenvolvimento do microsserviço *Reporting Engine Service*, segue-se portanto a etapa de substituir o módulo de *reporting* da *framework* que, em vez de implementar as funcionalidades, trata de encaminhar essa responsabilidade para o RES.

Nesta secção será abordado o caso de estudo da integração do microsserviço desenvolvido com um dos produtos *Elevation Primavera*, o *Cegid Jasmin*.

6.1 Cegid Jasmin

O *Cegid Jasmin* é um software de gestão na nuvem projetado para empreendedores e PMEs. Este produto oferece uma ampla gama de recursos, incluindo faturação rápida, controlo de despesas, gestão de compras, gestão de inventários e armazéns. Para além disso, possibilita também o acompanhamento de contas correntes e tesouraria, cumprimento de obrigações fiscais e monitorização do desempenho do negócio. A sua principal vantagem é disponibilidade de acesso através de qualquer dispositivo com conexão à Internet e em qualquer localização. Representa, por todas as suas caraterísticas, uma solução versátil no que diz respeito à gestão eficiente de pequenas e médias empresas. [10]

6.2 Impressão de Reports no Cegid Jasmin

Após a conclusão bem-sucedida do desenvolvimento do microsserviço *Reporting Engine Service*, o próximo passo consiste na modificação do módulo *Reporting* existente na *framework Elevation*. Em vez de

implementar diretamente as complexas funcionalidades de geração de *reports*, a abordagem adotada será a de direcionar essa responsabilidade para o RES.

O *Cegid Jasmin*, produto previamente construído usando a *framework Elevation*, foi escolhido como o foco para este caso de estudo. Tendo em conta que o mesmo já implementa funcionalidades de *reporting*, o objetivo é realizar a integração estratégica do RES com o *Cegid Jasmin* para uniformizar e otimizar essas mesmas ações. Com isto pretende-se alcançar uma impressão mais eficiente de *reports* e uma gestão dos *templates* mais abrangente e eficaz dentro do ambiente, não só do *Cegid Jasmin* mas também de todos os produtos *Elevation*. Esta adaptação procura proporcionar benefícios significativos, tanto para os utilizadores finais como para a operação geral do sistema.

O caso de estudo foi desenvolvido recorrendo a um protótipo do *Cegid Jasmin*, tendo em conta que todos os dados utilizados não são reais e servem apenas para efeito de demonstração. Assim sendo, para esclarecer a função do RES no ecossistema de produtos *Elevation*, serão considerados dois casos de uso no *Cegid Jasmin*: impressão da listagem de faturas e impressão da listagem de clientes.

Nas Figuras 26 e 27 estão representadas as interfaces Web onde podem ser visualizados os dados guardados, tanto dos clientes como das faturas. Desta forma, o objetivo que se pretende é a impressão de todas as informações listadas, para um documento em formato *pdf*, utilizando um template previamente definido pelo produto. Esta funcionalidade, anteriormente implementada pelo módulo de *Reporting* da *framework Elevation*, passa a ser responsabilidade do RES.

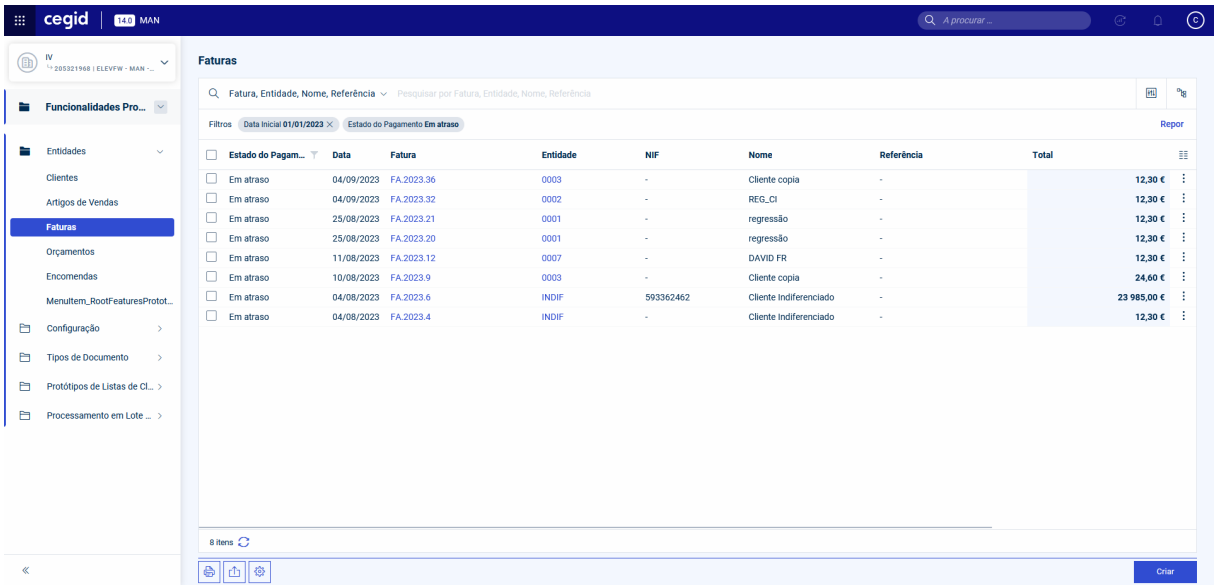


Figura 26: Listagem de faturas no Cegid Jasmin

Em primeiro lugar, o passo inicial da integração consistiu em adicionar a subscrição do protótipo nos

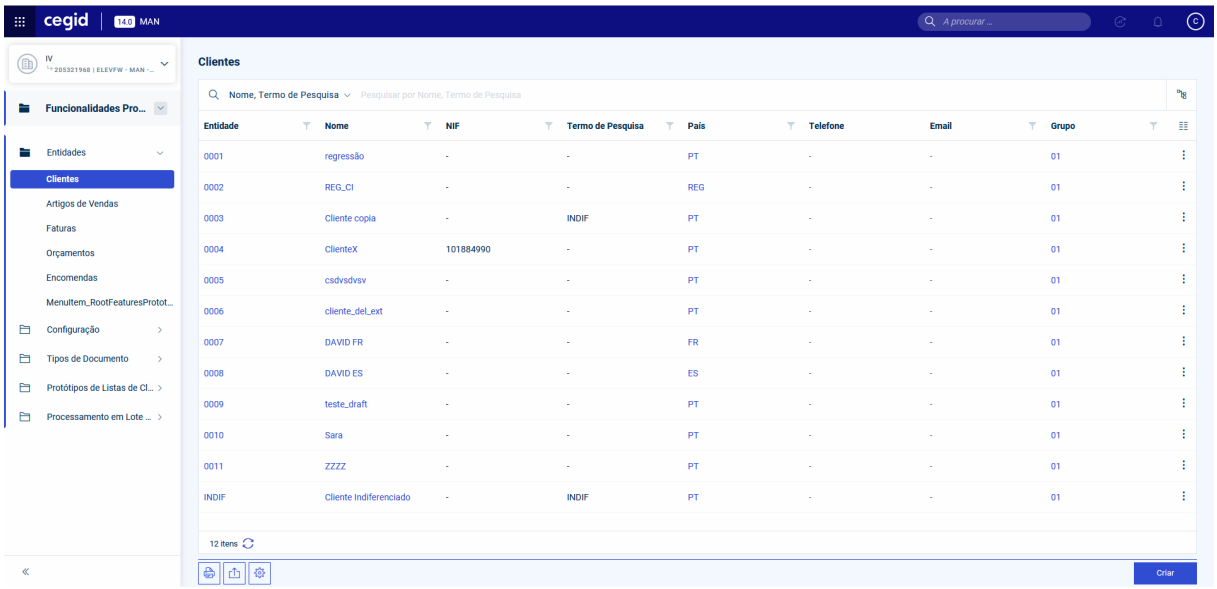


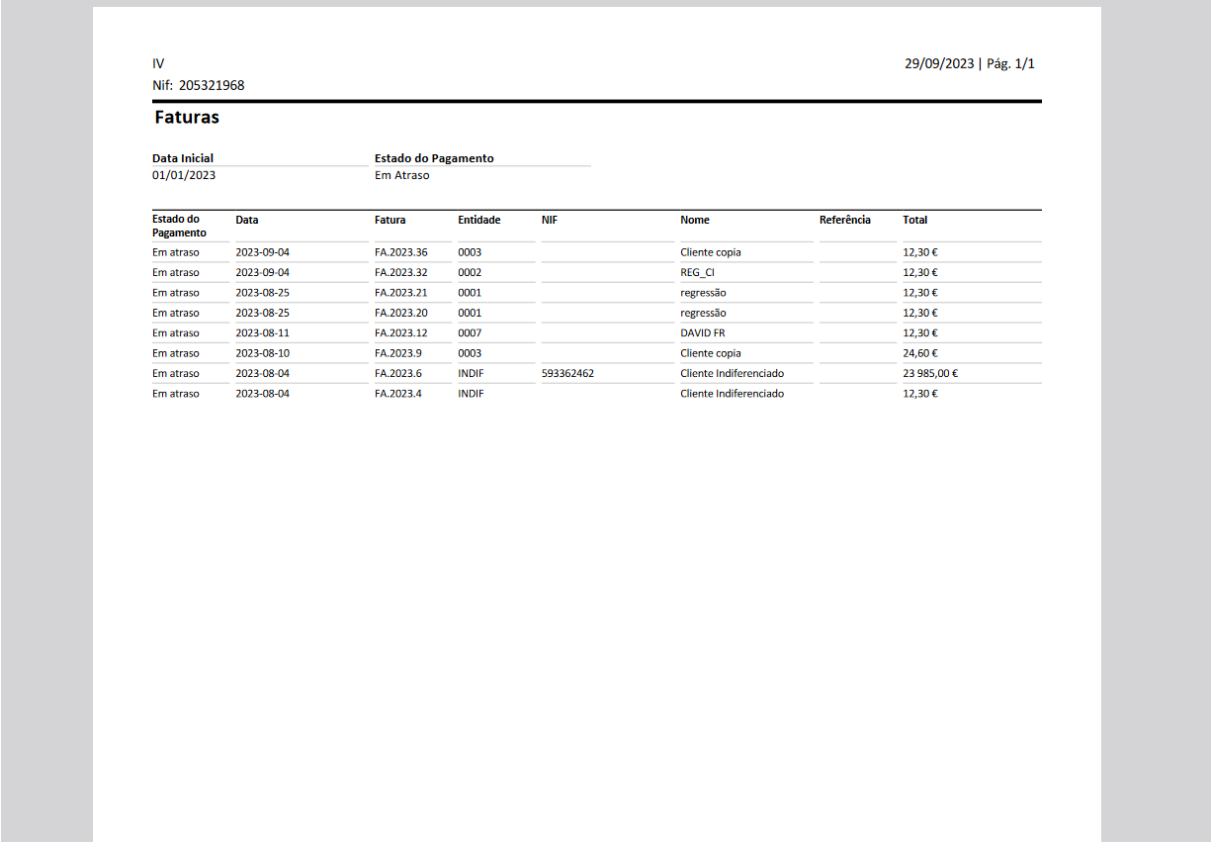
Figura 27: Listagem de clientes no Cegid Jasmin

serviços *Juice* para autorizar o acesso do produto ao microsserviço, que com as devidas permissões passa a ser capaz de aceder à *REST API* disponibilizada.

Com os acessos garantidos, seguiu-se a fase de implementação da nova solução através da atualização do método de impressão do módulo da *framework*. Numa primeira fase, foi necessário obter os dados de impressão, para o qual se recorreu a outro módulo do *Elevation*, o *QueryBuilder*. Este módulo permite efetuar vários tipos de *queries* à base de dados, de modo a obter informações das entidades guardadas de uma forma estruturada. Assim, foi possível obter as listagens de clientes e faturas, de acordo com os parâmetros especificados pelo utilizador, como por exemplo filtros ou parâmetro de pesquisa. Com as informações obtidas, foi necessário criar um novo objeto *json* com o formato exigido pelo *endpoint* de impressão da *API* do microsserviço, onde são agrupados todos os dados de impressão. Por este motivo, para além da lista das entidades foi ainda preciso especificar os parâmetros opcionais existentes para serem apresentados na impressão, como os filtros ou as informações da empresa.

Antes de efetuar a chamada ao RES, obteve-se o *token* de autenticação do microsserviço fornecendo as credenciais adequadas ao *Identity Server*. De seguida, é criado o pedido *http*, de acordo com o *url* correspondente e são fornecidos todos os dados e parâmetros necessários ao pedido (Figura 30). Por fim, através da invocação ao RES é recebido o ficheiro resultante da impressão para ser apresentado ao utilizador do produto, como se pode verificar nas Figuras 28 e 29. Estes dois *reports* apresentam algumas diferenças, nomeadamente a apresentação das informações da empresa no caso das faturas. Para além disso, observa-se na listagem de faturas a presença dos filtros que são parâmetros opcionais.

A implementação do método de impressão seguiu a lógica definida previamente no diagrama de sequência (Secção 5.2.1), ou seja todos os ajustes necessários aos campos do template foram realizados em tempo de execução. Assim sendo, na Figura 31 é possível observar o código correspondente, onde estão presentes todas as ações efetuadas para a geração de um *report* relativo a qualquer listagem de dados fornecida.



IV 29/09/2023 | Pág. 1/1

Nif: 205321968

Faturas

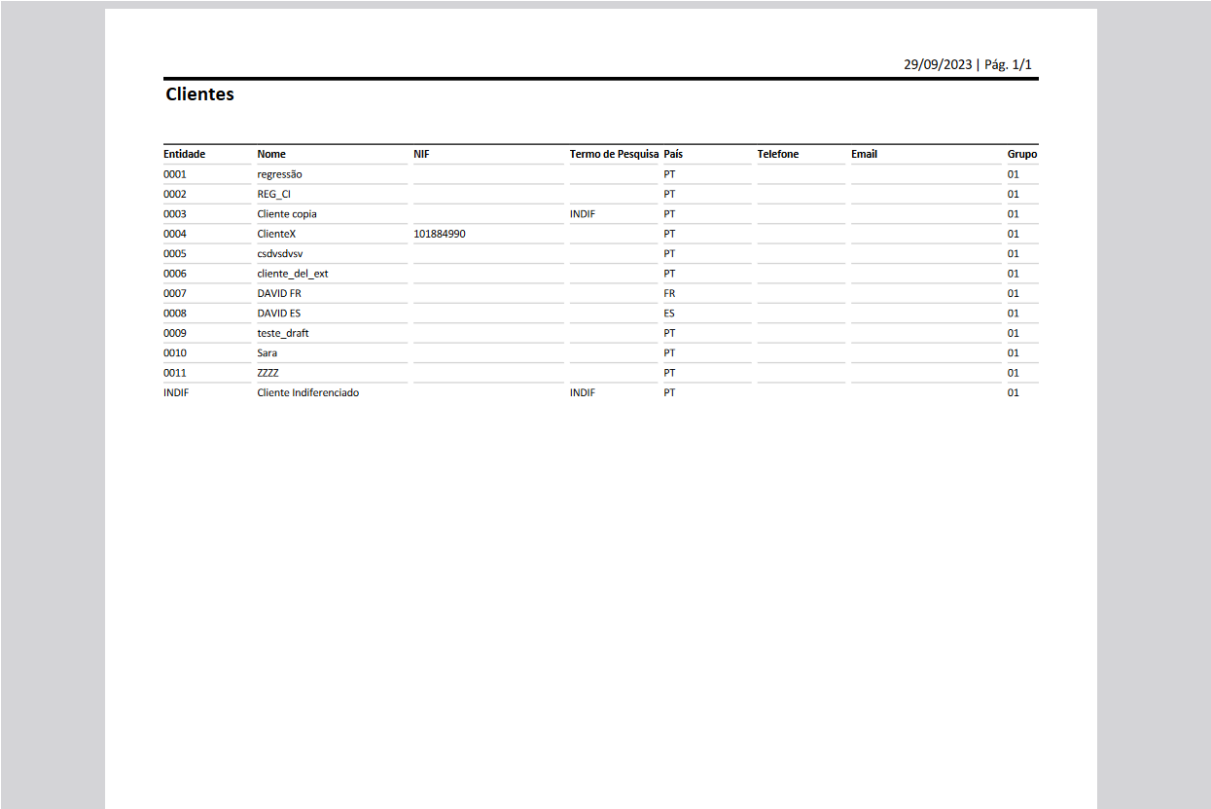
Data Inicial	Estado do Pagamento
01/01/2023	Em Atraso

Estado do Pagamento	Data	Fatura	Entidade	NIF	Nome	Referência	Total
Em atraso	2023-09-04	FA.2023.36	0003		Cliente copia		12,30 €
Em atraso	2023-09-04	FA.2023.32	0002		REG_CI		12,30 €
Em atraso	2023-08-25	FA.2023.21	0001		regressão		12,30 €
Em atraso	2023-08-25	FA.2023.20	0001		regressão		12,30 €
Em atraso	2023-08-11	FA.2023.12	0007		DAVID FR		12,30 €
Em atraso	2023-08-10	FA.2023.9	0003		Cliente copia		24,60 €
Em atraso	2023-08-04	FA.2023.6	INDIF	593362462	Cliente Indiferenciado		23 985,00 €
Em atraso	2023-08-04	FA.2023.4	INDIF		Cliente Indiferenciado		12,30 €

Figura 28: *Report* da listagem de faturas

É de notar que os *reports* mencionados utilizaram o mesmo template (Figura 32), no entanto, como referido anteriormente, o RES permite a edição ou criação de outros *templates* para serem utilizados na impressão. Assim sendo, na Figura 33 está presente a representação visual de um template distinto, e na Figura 34 o respetivo *report* resultante, tendo em conta que foram fornecidos todos os campos opcionais e a linguagem da impressão foi alterada para inglês, ação suportada pelo template em questão.

A utilização do RES para a integração do componente de *reporting* no Cegid Jasmin permitiu demonstrar como uma arquitetura microsserviços torna este processo mais ágil, escalável e eficiente. Apesar da implementação do microsserviço ter sido um procedimento complexo, pelas exigências adicionais inerentes à adoção desta arquitetura, as vantagens que oferece permitem ultrapassar estas contrariedades. A



29/09/2023 | Pág. 1/1

Clientes

Entidade	Nome	NIF	Termo de Pesquisa	País	Telefone	Email	Grupo
0001	regressão			PT			01
0002	REG_CI			PT			01
0003	Cliente copia		INDIF	PT			01
0004	ClienteX	101884990		PT			01
0005	csdvsdvsv			PT			01
0006	cliente_del_ext			PT			01
0007	DAVID FR			FR			01
0008	DAVID ES			ES			01
0009	teste_draft			PT			01
0010	Sara			PT			01
0011	ZZZZ			PT			01
INDIF	Cliente Indiferenciado		INDIF	PT			01

Figura 29: *Report* da listagem de clientes

concretização desta integração permitiu garantir que com apenas algumas linhas de código e fornecendo os acessos necessários é possível implementar um módulo complexo como o *Reporting*. Para além disso, o facto das funcionalidades estarem previamente integradas no módulo de *Reporting* da *framework Elevation*, significa que a sua atualização para o consumo do RES, será aplicada automaticamente a todos os produtos que utilizem a *framework*. Desta forma, a migração em termos funcionais será aplicada de forma instantânea. Por outro lado, em relação ao armazenamento de dados a sua transferência terá de ser feita gradualmente ao longo do tempo.

7 Conclusão

A adoção de uma arquitetura microsserviços desempenha um papel crucial no mundo empresarial moderno, dando capacidade às organizações de se tornarem cada vez mais ágeis, eficientes e orientadas para o cliente. Esta abordagem arquitetónica proporciona a flexibilidade necessária para enfrentar os desafios em constante evolução do mercado e promove a inovação contínua, tornando-se uma parte fundamental da estratégia de negócios de muitas empresas de sucesso.

Ao longo deste documento foi possível demonstrar as inúmeras vantagens da adoção de uma arquitetura microsserviços apesar da sua complexidade de implementação. Por outro lado, a existência de diversos padrões que refletem diferentes abordagens dentro do âmbito da construção de microsserviços permitem que a sua implementação seja mais rápida e de fácil compreensão para todos os desenvolvedores de software envolvidos. Para além disso, indicam diferentes estratégias de implementação que se podem adaptar a diferentes contextos do problema.

Por outro lado, o estudo das funcionalidades de *reporting* permitiu compreender as particularidades do funcionamento das mesmas e a sua importância no mundo empresarial. Consequentemente a análise das ferramentas de *reporting* existentes no mercado possibilitou o estudo das vantagens e desvantagens de cada uma para que no final tenha sido feita a escolha certa no que diz respeito à convergência com as necessidades do microsserviço de *reporting*.

Por fim, o desenvolvimento do Reporting Engine Service (RES) permitiu concretizar os objetivos propostos inicialmente. A integração desta solução com os produtos *cloud* Primavera conseguiu comprovar que, em sistemas complexos, a substituição de uma arquitetura monolítica por microsserviços é essencial no desenho de um software eficiente, flexível, escalável e inovador. Desta forma, na solução inicial o módulo de *reporting* da *framework Elevation* para ser utilizado teria de estar associado ao produto em uso, incluído todos os recursos necessários. No entanto, com o surgimento do RES é garantida uma centralização de todos os recursos num único ponto, a uniformização do *reporting* para todo o ecossistema da Cegid Primavera e uma maior facilidade de acesso às funcionalidades que abrange.

7.1 Dificuldades encontradas

No que diz respeito às dificuldades encontradas, numa fase inicial o maior obstáculo foi compreender o conceito de *reporting*, que por ser um domínio ainda desconhecido levou à necessidade de uma pesquisa ainda mais aprofundada. Para além disso, o facto do estudo de ferramentas de *reporting* assentar em funcionalidades específicas, como a impressão de documentos levou a uma restrição da pesquisa a plataformas que cumprissem os requisitos pretendidos. Por outro lado, durante o desenvolvimento do RES, houve dificuldades devido à adaptação a novas tecnologias. Em primeiro lugar, o microsserviço foi implementado em C#, uma linguagem de programação com a qual ainda não havia familiaridade. Para além disso, houve inclusive uma fase de aprendizagem para ter capacidades a fim de dominar as *frameworks* utilizadas pela Primavera como *Lithium* e *Elevation*.

7.2 Perspetiva de trabalho futuro

Fazendo uma análise e discussão dos resultados obtidos com a concretização deste projeto e tendo uma base sólida relativamente ao domínio estudado, é essencial considerar os próximos passos a tomar, uma vez que ainda existem particularidades a explorar.

Em relação ao desenvolvimento do microsserviço, houve um compromisso em priorizar funcionalidades como a impressão de listas e o armazenamento de *templates*, o que levou à exclusão, numa fase inicial, da impressão de entidades. Assim sendo, a principal tarefa a realizar no futuro será a exploração desta funcionalidade, procurando uma forma de a generalizar, de modo a atender as necessidades dos produtos e serviços do ecossistema da Cegid Primavera.

Numa perspetiva da empresa, a etapa seguinte será disponibilizar o microsserviço na nuvem e efetuar gradualmente a transição dos *templates* pertencentes a cada subscrição, de cada produto para o armazenamento do novo serviço. Em simultâneo, é também necessário garantir que novos *templates* serão, a partir desse momento, geridos pelo microsserviço. Completada a transição, poderá assim ser substituído integralmente o módulo de *reporting* da *framework Elevation* para direcionar todas as suas ações para o RES.

Para além disso, o facto de o microsserviço estar disponível na *cloud* permite que qualquer produto ou serviço possa usufruir das capacidades do mesmo. Isto significa que no futuro, qualquer sistema que requeira funcionalidades de *reporting* tem acesso fácil ao *RES* que foi especialmente desenhado para atender às necessidades do domínio.

8 Bibliografia

[1] Clean architecture flashcards. [Online]. URL https://quizlet.com/728229255/clean-architecture-flash-cards/. Último acesso: 22 Janeiro, 2023.

[2] Amazon Web Services. Event driven architecture. [Online], 2003. URL https://aws.amazon.com/pt/event-driven-architecture/. Último acesso: 22 Janeiro, 2023.

[3] Chris Anderson. *Printing and Reporting*, pages 521–563. Apress, Berkeley, CA, 2012. ISBN 978-1-4302-3501-9. doi: 10.1007/978-1-4302-3501-9_15. URL https://doi.org/10.1007/978-1-4302-3501-9_15.

[4] Ben Aston. 10 Best Reporting Tools & Software of 2023. [Online], Dec 2022. URL https://thedigitalprojectmanager.com/tools/best-reporting-tools/. Último acesso: 22 Janeiro, 2023.

[5] Alessandro Beltrão, Fábio Farzat, and Guilherme Travassos. Technical debt: A clean architecture implementation. In *Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática*, pages 131–134, Porto Alegre, RS, Brasil, 2020. SBC. doi: 10.5753/cbsoft_estendido.2020.14620. URL https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/14620.

[6] Oskar uit de Bos. The engineers guide to event-driven architectures: Benefits and challenges. [Online], Nov 2020. URL https://medium.com/swlh/the-engineers-guide-to-event-driven-architectures-benefits-and-challenges-3e96ded8568b. Último acesso: 22 Janeiro, 2023.

[7] Cegid Primavera. Cegid Ekon. [Online], . URL https://pt.primaverabss.com/pt/pagina/cloud-erp/. Último acesso: 23 Outubro, 2023.

[8] Cegid Primavera. Elevation Documentation - Architecture. [Online], . URL https://docs.primaverabss.com/elevation-documentation-shared/documentation/architecture/. Último acesso: 23 Outubro, 2023.

[9] Cegid Primavera. Identity Server Documentation - Vision. [Online], . URL https://docs.primaverabss.com/lithium-product-core.ids/vision/. Último acesso: 23 Outubro, 2023.

[10] Cegid Primavera. Software de gestão para empresas com Futuro - Cegid Jasmin. [Online], . URL https://www.jasminsoftware.pt/software-gestao/. Último acesso: 23 Outubro, 2023.

[11] Cegid Primavera. Cegid Jasmin - Software de Gestão Cloud para Pequenos Negócios. [Online], . URL https://www.jasminsoftware.pt/. Último acesso: 23 Outubro, 2023.

[12] Cegid Primavera. Applications Service (APPS) Specification v1.0. [Online], . URL https://docs.primaverabss.com/juice-documentation-shared/dir/spec/apps-spec/. Último acesso: 23 Outubro, 2023.

[13] Cegid Primavera. Juice Documentation - Vision. [Online], . URL https://docs.primaverabss.com/juice-documentation-shared/vision/. Último acesso: 23 Outubro, 2023.

[14] Cegid Primavera. Lithium Documentation - Microservice Architecture. [Online], . URL https://docs.primaverabss.com/lithium-documentation-shared/vision/4-microservice-architecture/. Último acesso: 23 Outubro, 2023.

[15] Cegid Primavera. Lithium Documentation - Lithium Architecture. [Online], . URL https://docs.primaverabss.com/lithium-documentation-shared/vision/3-lithium-architecture/. Último acesso: 23 Outubro, 2023.

[16] Cegid Primavera. Rose People - Processamento de salários automático. [Online], . URL https://pt.primaverabss.com/pt/pagina/rose-people/. Último acesso: 23 Outubro, 2023.

[17] Coolfin.pt. Cash Flow - Significado e importância. [Online]. URL https://www.coolfin.pt/info/cash-flow. Último acesso: 22 Janeiro, 2023.

[18] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96, 2019. doi: 10.1109/ISSREW.2019.00050.

[19] DevExpress. .NET reporting tools - core, blazor, WinForms, MVC. [Online], . URL https://www.devexpress.com/subscriptions/reporting/. Último acesso: 22 Janeiro, 2023.

[20] DevExpress. Report Features. [Online], . URL https://docs.devexpress.com/XtraReports/2162/reporting#report-features. Último acesso: 22 Janeiro, 2023.

[21] DevExpress. DevExpress.XtraReports Namespace. [Online], . URL https://docs.devexpress.com/XtraReports/DevExpress.XtraReports. Último acesso: 23 Outubro, 2023.

[22] Martin Fowler and James Lewis. MicroservicePremium. 2014. URL https://www.martinfowler.com/bliki/MicroservicePremium.html. Último acesso: 22 Janeiro, 2023.

[23] Martin Fowler and James Lewis. Microservices. 2014. URL http://martinfowler.com/articles/microservices.html. Último acesso: 22 Janeiro, 2023.

[24] Igor Gonçalo Gomes de Sá. Análise e Concepção de uma Framework de Reporting genérica e parametrizável. Dissertação de Mestrado em Engenharia Informática, Universidade do Minho, 2012. Disponível em: https://hdl.handle.net/1822/27812.

[25] Fábio Daniel de Sá Gonçalves. Arquitetura de micro-serviços e o caso da Framework Lithium da PRIMAVERA BSS. Dissertação de Mestrado em Engenharia Informática, Universidade do Minho, 2023. Disponível em: https://hdl.handle.net/1822/84563.

[26] Grupo Primavera. Grupo Primavera - Software de Gestão. [Online], . URL https://grupoprimavera.com/pt-pt/. Último acesso: 22 Janeiro, 2023.

[27] Grupo Primavera. Cegid e Grupo Primavera celebram acordo de união. [Online], . URL https://pt.primaverabss.com/pt/comunicados-de-imprensa/cegid-e-grupo-primavera-celebram-acordo-de-uniao/. Último acesso: 22 Janeiro, 2023.

[28] Grupo Primavera. Cegid confirma aquisição do Grupo Primavera. [Online], . URL https://pt.primaverabss.com/pt/comunicados-de-imprensa/cegid-confirma-aquisicao-do-grupo-primavera/. Último acesso: 22 Janeiro, 2023.

[29] Taufiq Hidayat. Micro services versus monolithic architecture. what are they? [Online], May 2020. URL https://medium.com/javanlabs/micro-services-versus-monolithic-architecture-what-are-they-e17ddc8d3910. Último acesso: 22 Janeiro, 2023.

[30] Javiera Laso. What is domain-centric architecture? [Online], Mar 2022. URL https://jlasoc.medium.com/what-is-domain-centric-architecture-e030e609c401. Último acesso: 22 Janeiro, 2023.

[31] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing — the business perspective. *Decision Support Systems*, 51(1):176–189, 2011. ISSN 0167-9236. doi: https://doi.org/10.1016/j.dss.2010.12.006. URL https://www.sciencedirect.com/science/article/pii/S0167923610002393.

[32] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015. URL https://isip.piconepress.com/courses/temple/ece_1111/resources/articles/20211201_software_architecture_patterns.pdf.

[33] Chris Richardson. Pattern: Api gateway / backends for frontends. [Online], . URL https://microservices.io/patterns/apigateway.html. Último acesso: 22 Janeiro, 2023.

[34] Chris Richardson. Pattern: Client-side service discovery. [Online], . URL https://microservices.io/patterns/client-side-discovery.html. Último acesso: 22 Janeiro, 2023.

[35] Chris Richardson. Pattern: Server-side service discovery. [Online], . URL https://microservices.io/patterns/server-side-discovery.html. Último acesso: 22 Janeiro, 2023.

[36] Chris Richardson. Pattern: Service registry. [Online], . URL https://microservices.io/patterns/service-registry.html. Último acesso: 22 Janeiro, 2023.

[37] SAP. SAP Crystal Reports: Business intelligence reporting tools. [Online]. URL https://www.sap.com/products/technology-platform/crystal-reports.html. Último acesso: 22 Janeiro, 2023.

[38] SAPO. Afinal, o que é uma PME? [Online]. URL https://pmemagazine.sapo.pt/afinal-o-que-e-uma-pme/. Último acesso: 22 Janeiro, 2023.

[39] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. 03 2018. doi: 10.5220/0006798302210232.

[40] Telerik. Telerik Reporting - .NET Reporting Tool. [Online]. URL https://www.telerik.com/products/reporting.aspx. Último acesso: 22 Janeiro, 2023.

[41] Windward Studios. Simple, easy & seamless docgen integration software: Windward Studios. [Online]. URL https://www.windwardstudios.com/solution/windward-core. Último acesso: 22 Janeiro, 2023.

A Anexos

```
private async Task<FileStreamResult> GetFileFromRESAsync(string sourceModel)
{
    string token = await this.GetTokenUsingDefaultCredentialsAsync().ConfigureAwait(false);

    using StringContent data = new StringContent(sourceModel, Encoding.UTF8, "application/json");

    // Get subscription fields
    string workspace = ApplicationContext.Tenant.TenantKey;
    string organization = ApplicationContext.Organization.OrganizationKey;

    // Get RES url address
    string resAddress = ConfigurationService.GetAppSetting("ReportingEngineServiceAddress");

    string url = $"{resAddress}/prototype/{workspace}/{organization}/prints/list";

    // Call RES
    using HttpClient httpClient = HttpClientBuilder.Create();
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
    httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

    HttpResponseMessage response = await httpClient.PostAsync(new Uri(url), data);

    if (response.IsSuccessStatusCode)
    {
        string responseContentType = response.Content.Headers.ContentType.MediaType;

        // Read the response content as a stream
        Stream responseStream = await response.Content.ReadAsStreamAsync();

        // Convert to ASCII encoding
        MemoryStream resultStream = this.ToASCIIMemoryStream(responseStream);

        // Create the FileStreamResult with the memory stream
        FileStreamResult fileStreamResult = new FileStreamResult(resultStream, responseContentType);

        return fileStreamResult;
    }
    else
    {
        return null;
    }
}
```

Figura 30: Chamada ao RES para obtenção de um *report*

```
/// <summary>
/// Implementation for the List Printing Service Provider.
/// </summary>
public class PrintListManagerService : IPrintListManagerService
{
    /// <inheritdoc/>
    public async Task<Domain.File> PrintListReportByTemplateAsync(string product, string workspace, string appInstance, string
    xmlTemplateString, string culture, ListDefinition listDefinition, ListData listData, CancellationToken cancellationToken = default)
    {
        Domain.File file = new Domain.File();

        using (ListBaseTemplate listTemplate = new(xmlTemplateString))
        {
            CultureInfo cultureInfoNewSymbol = new CultureInfo(culture);
            CultureInfo.CurrentCulture = cultureInfoNewSymbol;
            listTemplate.ApplyLocalization(cultureInfoNewSymbol);

            listTemplate.AddCustomDataToDataSource(listData);

            if (listData != null && listData.Filters != null)
            {
                listTemplate.AddFilterTable(listData.Filters);
            }

            listTemplate.AdjustHeaderFields(listData!);

            listTemplate.AddTableByListDefinition(listDefinition);

            byte[]? data = null;
            using (MemoryStream inputStream = new MemoryStream())
            {
                await listTemplate.ExportToPdfAsync(inputStream, null, cancellationToken).ConfigureAwait(false);
                data = inputStream.ToArray();
            }

            file = new Domain.File
            {
                Name = "DocumentList.pdf",
                ContentType = MediaTypeNames.Application.Pdf,
                Data = data!
            };
        }

        return file;
    }
}
```

Figura 31: Código para impressão de um *report* do tipo lista

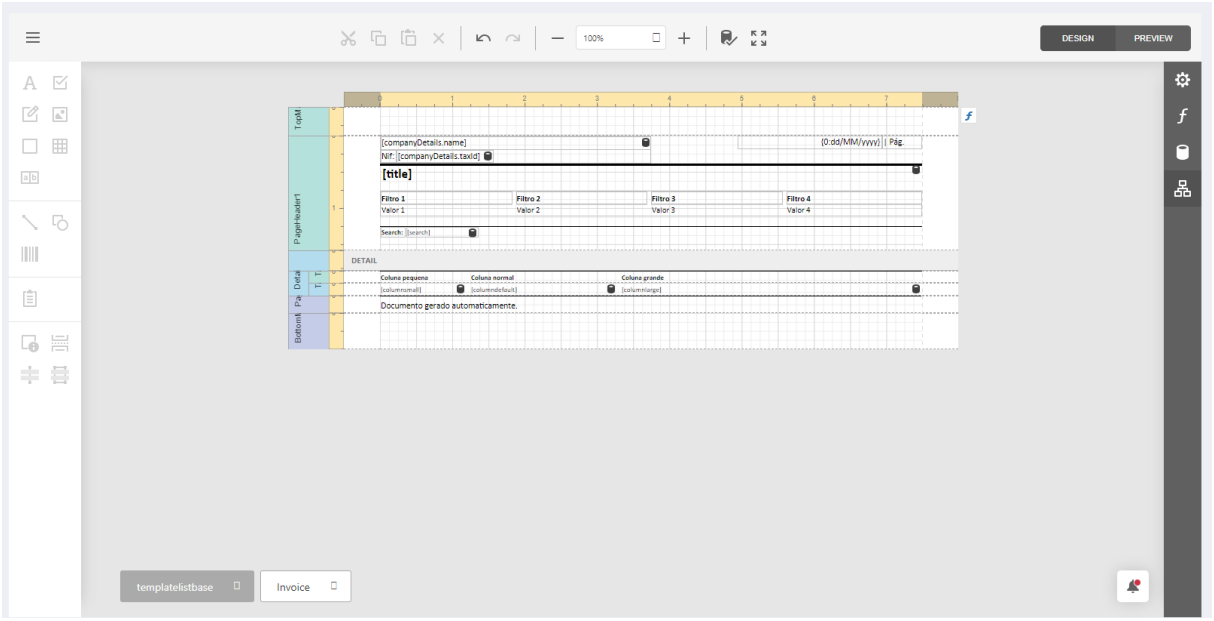


Figura 32: Representação no *designer* do *templatelistbase*

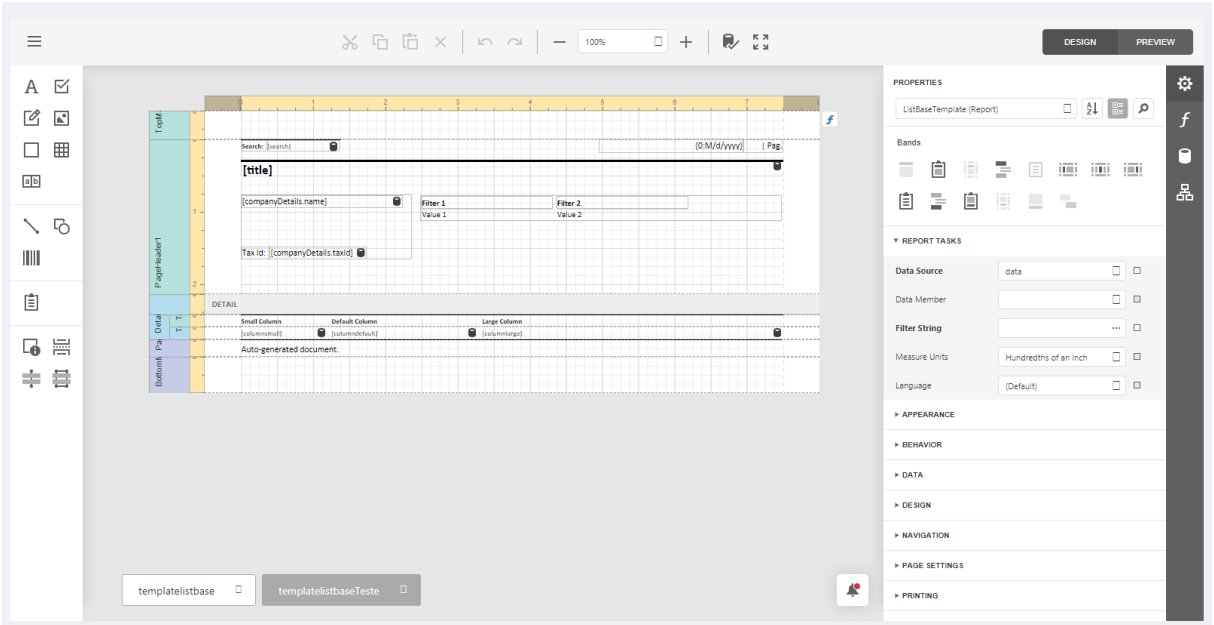
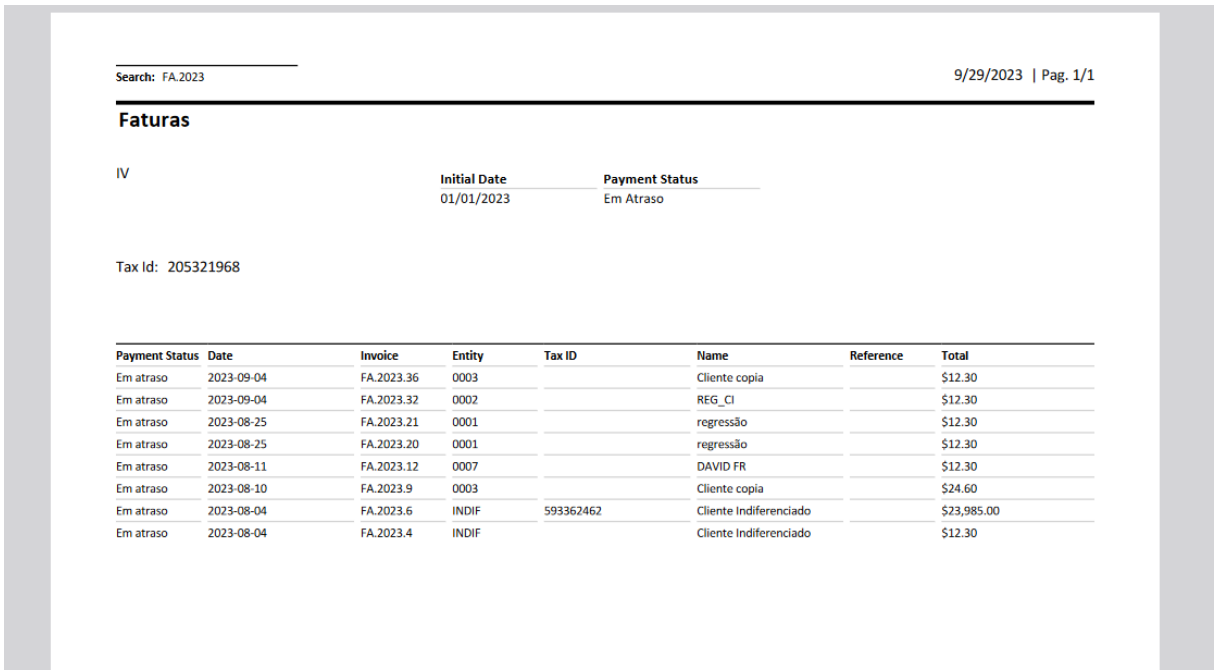


Figura 33: Representação no *designer* do *templatelistbaseteste*



Search: FA.2023 9/29/2023 | Pag. 1/1

Faturas

IV

Initial Date	Payment Status
01/01/2023	Em Atraso

Tax Id: 205321968

Payment Status	Date	Invoice	Entity	Tax ID	Name	Reference	Total
Em atraso	2023-09-04	FA.2023.36	0003		Cliente copia		$12.30
Em atraso	2023-09-04	FA.2023.32	0002		REG_CI		$12.30
Em atraso	2023-08-25	FA.2023.21	0001		regressão		$12.30
Em atraso	2023-08-25	FA.2023.20	0001		regressão		$12.30
Em atraso	2023-08-11	FA.2023.12	0007		DAVID FR		$12.30
Em atraso	2023-08-10	FA.2023.9	0003		Cliente copia		$24.60
Em atraso	2023-08-04	FA.2023.6	INDIF	593362462	Cliente Indiferenciado		$23,985.00
Em atraso	2023-08-04	FA.2023.4	INDIF		Cliente Indiferenciado		$12.30

Figura 34: *Report* da listagem de faturas com *templatelistbaseteste*