LI3 - Relatório da Fase I - Grupo 12

Humberto Gomes (A104348)
José Lopes (A104541)
José Matos (A100612)

novembro de 2023

Resumo

Este relatório tem como intuito explicar a estrutura do nosso trabalho prático para a UC de LI3. Como o foco principal desta UC é a modularização e o encapsulamento do código, este documento descreve como tal for conseguido, justificando as nossas decisões a este nível. O objetivo do projeto para a 1.ª fase é o *parsing* e validação de um *dataset* contendo utilizadores, voos, passageiros em voos e reservas de hotéis, sobre o qual serão executadas *queries*, das quais implementámos seis, fornecendo informação sobre a base de dados. O modo de organização e processamento destes dados também é descrito.

1 Estrutura do trabalho

Devido à elevada complexidade e ao elevado número de módulos neste projeto, após um diagrama que de dependências completo, este documento separa-o em diversas secções lógicas, que têm o seu funcionamento descrito uma a uma. Mesmo assim, para reduzir a complexidade visual, não incluímos todas as relações de dependência, mas apenas as mais relevantes. Segue-se a nossa convenção gráfica:

- Um retângulo com cantos arredondados representa uma estrutura de dados;
- Um retângulo sem cantos arredondados representa um módulo cuja tarefa principal é a execução de código. Mesmo assim, um destes módulos pode conter estruturas de dados auxiliares (por exemplo, uma gramática definida no módulo de um *parser*);
- $A \to B$ significa que o módulo A depende do módulo B. $A \dashrightarrow B$ representa uma falsa dependência, por exemplo, a necessidade de se saber da existência de um nome de um tipo opaco.

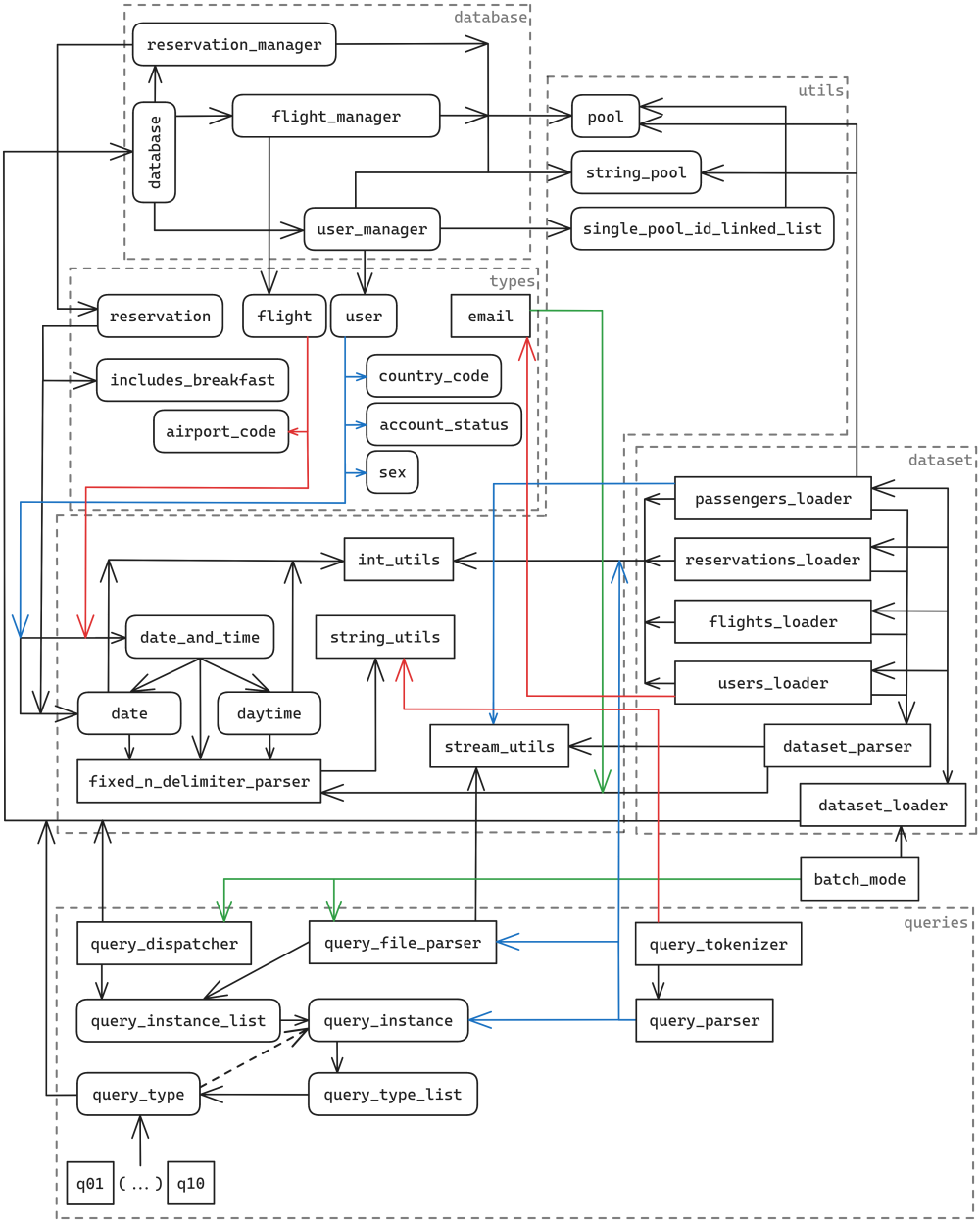

Figura 1: Diagrama de dependências de toda a aplicação. Cores são utilizadas apenas para facilitar a leitura.

1.1 *Parsing*

O nosso sistema de *parsing* começa com *tokenizadores*, que separam *strings* e ficheiros por um delimitador (`string_utils` e `stream_utils`, respetivamente). Sobre estes é construído o caso mais específico do `fixed_n_delimiter_parser`, que espera um número pré-determinado de *tokens*, chamando um *callback* diferente para cada um; uma gramática genérica, definível pelo programador, define estes *callbacks*. Este *parser* é utilizado para a validação de *emails* (`email`) e para o *parsing* de datas (`date`), horas de um dia (`daytime`), combinações de datas e horas (`date_and_time`), e linhas de um *dataset*. `int_utils` é um *parser* de valores inteiros, semelhante às funções `atoi` e `strtol`, mas com deteção de erros mais rigorosa.

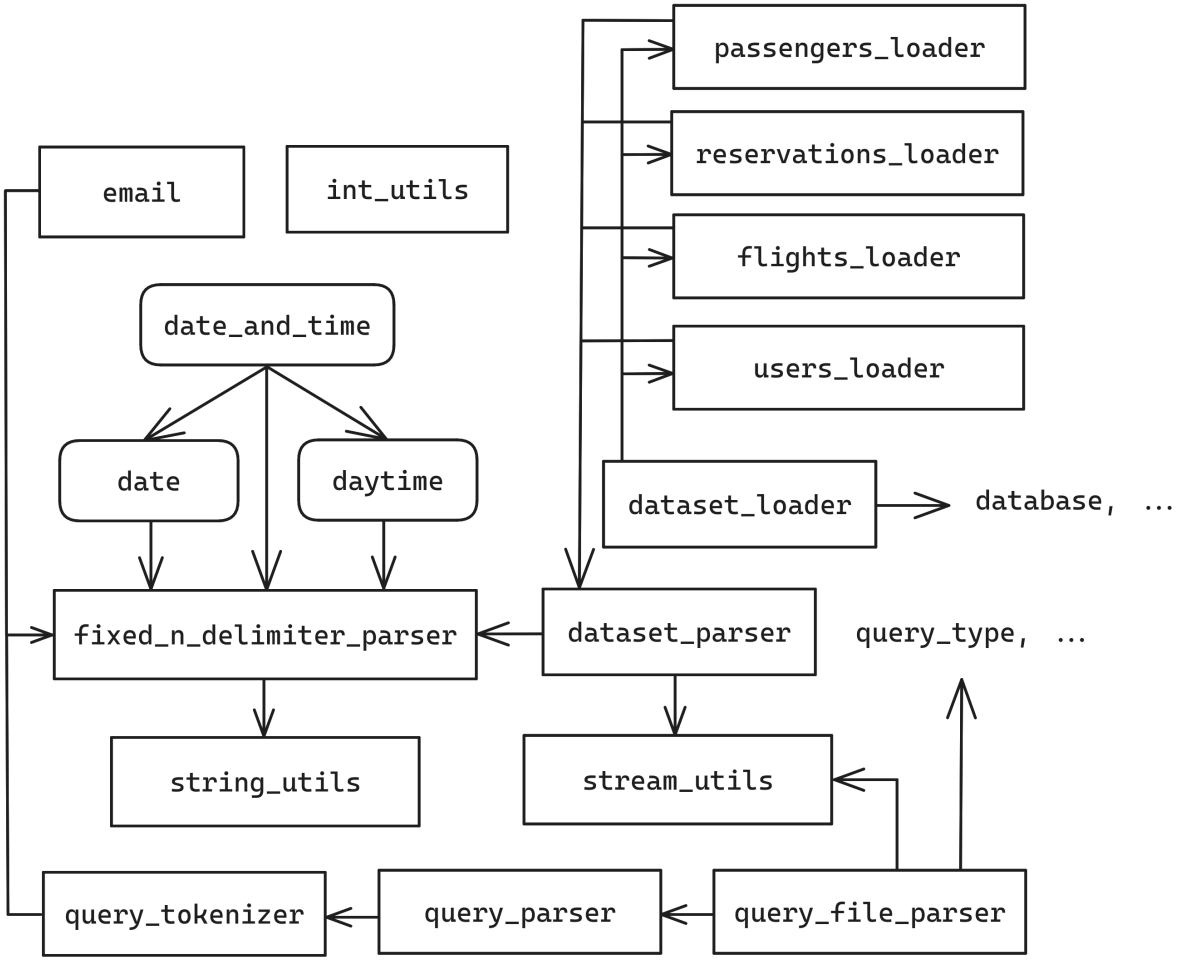

Figura 2: Diagrama de dependências do subsistema de *parsing*

Para a análise de um ficheiro de *dataset*, o `dataset_parser`, é responsável por separar o ficheiro por linhas, excluir a primeira linha (o *header* da tabela CSV), e *tokenizar* cada linha, chamando os *callbacks* adequados na sua gramática também customizável. O `dataset_loader`, não propriamente encaixado neste secção de *parsing*, é responsável por abrir cada ficheiro do *dataset*, interagir com os *parsers* adequados (`*_loader`), que adicionam elementos à base de dados, enquanto o `dataset_loader` regista os erros reportados nos ficheiros de erro.

O *parsing* de *queries* requer um *tokenizador* adicional para lidar com aspas (`query_tokenizer`), um *parser* para determinar o tipo de uma *query* e se os seus argumentos são válidos (`query_parser`), e um *parser* de um ficheiro com uma *query* por linha (`query_file_parser`).

1.2 Entidades

O nosso projeto define três entidades (`user_t`, `reservation_t` e `flight_t`), correspondentes às definidas no enunciado do trabalho, juntamente com módulos de estruturas de dados auxiliares. Os campos destas entidades formam um subconjunto dos campos definidos no enunciado do

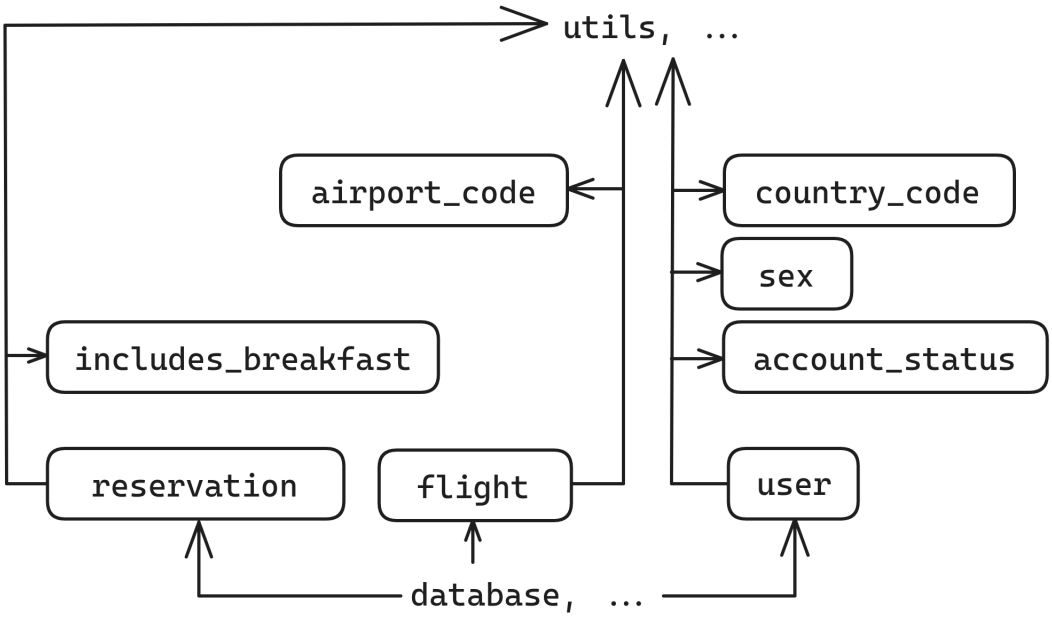

Figura 3: Diagrama de dependências das entidades da aplicação

trabalho (não armazenamos campos nunca pedidos por *queries*). A única exceção ocorre nos voos, onde adicionamos um campo relativo ao número de passageiros, devido à sua grande utilidade tanto para a validação do *dataset* como para a execução de *queries*.

1.3 Catálogos

Antes de definirmos uma base de dados, começámos por definir estruturas que permitem melhorar a eficiência espacial e a velocidade das alocações na aplicação: um alocador em `pool` para objetos todos do mesmo tamanho, e uma `string_pool` para *strings*. Definimos também uma `single_pool_id_linked_list`, uma implementação de uma lista ligada na qual que várias listas podem partilhar a mesma `pool` para armazenamento de nodos, contribuindo para um menor uso de memória em *overheads* de alocação.

No módulo `database`, definimos uma estrutura de dados que contém os três *managers* na aplicação: o `reservation_manager`, que gere reservas, o `flight_manager`, que gere voos, e o `user_manager`, que gere utilizadores. Este último liga também cada utilizador aos identificadores de voos e reservas a ele associados. Todos estes *managers* consistem numa `pool` para alocação de entidades, uma `string_pool` para alocação de *strings*, e uma tabela de *hash*, para a associação de identificadores de entidades às entidades em si. O `user_manager` surge como exceção, onde cada identificador de um `user` se encontra também associado a listas ligadas com os IDs dos voos e reservas associados a esse utilizador.

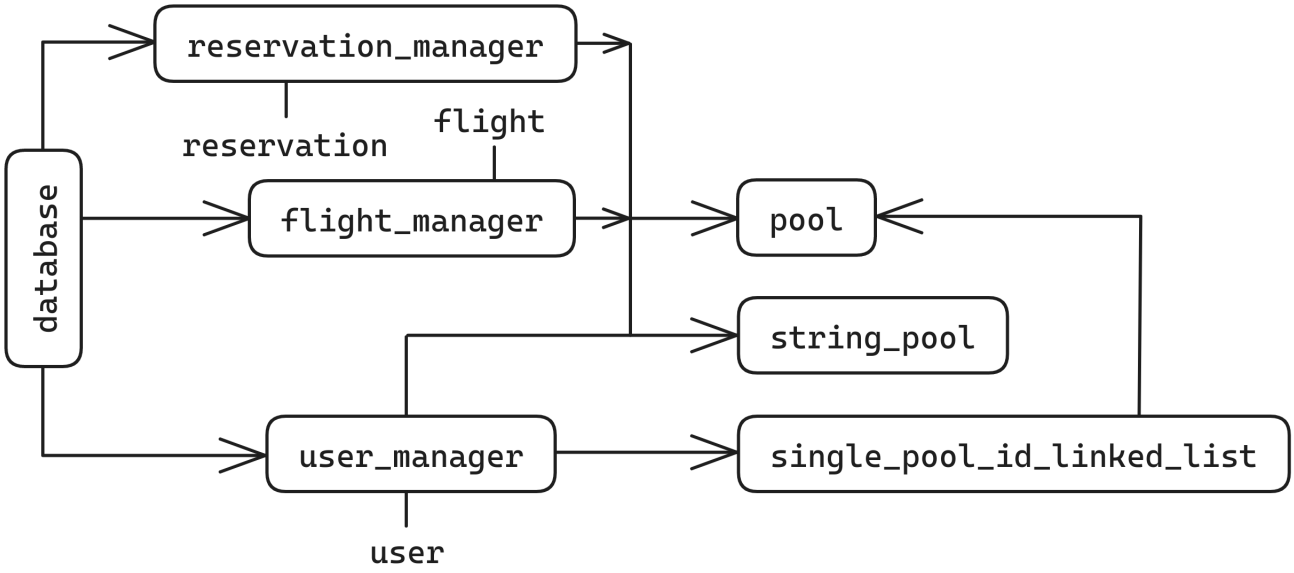

Figura 4: Diagrama de dependências dos catálogos na aplicação

1.4 *Queries*

O *parsing* de *queries* já foi descrito na secção *Parsing*. No sistema de *queries*, começámos por definir um `query_type`, um conjunto de *callbacks* que define um tipo de *query*, de modo a se simular polimorfismo em C. Definir uma *query* resume-se a a definir uma função para cada *callback*, e adicioná-la à `query_type_list`, a lista de todas as *queries* conhecidas.

Antes de enumerar as *queries* implementadas, devemos mencionar como geramos dados estatísticos para uma *query*. Ao contrário do sugerido no enunciado no projeto, não utilizamos um módulo para estatísticas, dado que implementá-lo seria uma quebra do modularidade da aplicação: implementar uma nova *query* implicaria edições consideráveis a vários módulos. Sendo assim, optámos por um modelo em que dados estatísticos são gerados por cada tipo (número) de *query*, e são partilhados por todas as *queries* do mesmo tipo. Por exemplo, torna-se possível fazer uma única iteração da base de dados para todas as *queries* do tipo 3, em vez de uma iteração para cada *query* deste tipo. Assim, temos uma solução mais modular, mas com pior desempenho do que um módulo de estatísticas globais (que faria uma única iteração pelos dados, gerando dados usados por todas as *queries*).

Estas são as *queries* que implementámos nesta primeira fase:

- **Q01** - Consulta de uma entidade pelo seu identificador. A sua implementação foi trivial: começa-se por determinar o tipo da entidade com base no formato do seu identificador, processo seguido da consulta direta do *manager* correto na base de dados.
- **Q02** - Listagem de voos e / ou reservas de um utilizador. Também de implementação tri-

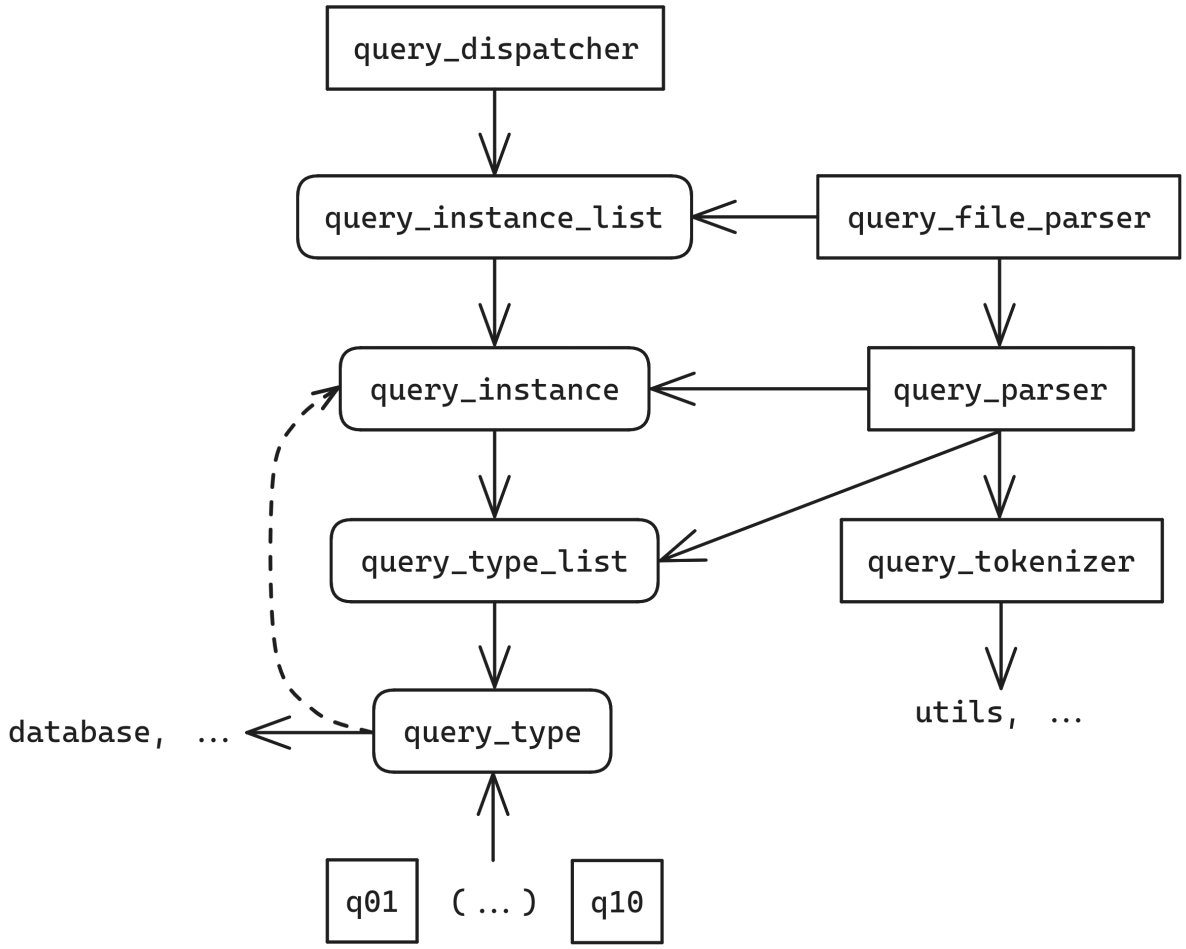

Figura 5: Diagrama de dependências do subsistema de *queries*

vial, esta *query* pede a lista de reservas / voos (ou ambas) de um utilizador ao `user_manager` (consulta direta), seguida da consulta também direta do *manager* adequado para cada voo / reserva, para a obtenção de informação sobre data do evento.

- **Q03** - Apresentação da classificação média de um hotel. Numa única iteração estatística pelas reservas, calcula-se a média de todos os hotéis mencionados nas *queries* de tipo 3. A execução de cada *query* resume-se então à consulta direta dos dados estatísticos, formados por uma tabela de *hash* que associa o identificador de um hotel à sua média.

- **Q04** - Listagem das reservas de um hotel. Tal como a Q03, faz-se uma única iteração estatística da lista de reservas, mas em vez de se calcular uma média com cada reserva, adiciona-se a reserva a um *array* associado a esse hotel através de uma tabela de *hash*. Após se ordenar cada *array*, pode-se proceder à execução de cada *query*, uma consulta direta dos dados estatísticos.

- **Q06** - Listagem dos N aeroportos com mais passageiros num dado ano. Com uma única iteração estatística pelo `flight_manager`, forma-se uma tabela de *hash*, que associa anos a outras tabelas de *hash*, que por sua vez associam aeroportos a números de passageiros. Cada uma destas segundas tabelas de *hash* é convertida para um *array* ordenado de pares

aeroporto-passageiros. Assim, a execução de uma *query* limita-se à escolha do *array* de pares correto e à apresentação dos seus primeiros N elementos.

- **Q09** - Listagem de todos os utilizadores cujo nome tenha como prefixo o argumento desta *query*. De momento, devido ao breve prazo de entrega, implementámos esta *query* ineficientemente, com uma iteração do gestor de utilizadores e filtragem dos nomes por cada *query* no ficheiro de *input*. Segue-se a ordernação e apresentação dos resultados. Pretendemos melhorar esta *query* na segunda fase, adicionando uma árvore binária de procura com os nomes dos utilizadores ao `user_manager`.

Retornando à descrição de cada módulo, uma `query_instance` refere-se à ocorrência de uma *query* (num ficheiro, por exemplo), e uma `query_instance_list` a uma lista destas. Por último, o `query_dispatcher` é o módulo responsável por executar uma lista de *queries* dada uma base de dados e os ficheiros de *output* de cada *query*.

2 Otimização do uso de memória

Dado que os *datasets* da 2.ª fase serão de maior dimensão, preocupámo-nos desde já com a quantidade de memória utilizada, de modo não sermos futuramente obrigados a reescrever partes significativas do nosso código.

2.1 Observação do *dataset* e das *queries*

Pudemos observar que alguns campos do *dataset* nunca precisavam de estar presentes no *output* de nenhuma *query* (ex: o *email* de um utilizador), pelo que era escusado o seu armazenamento na base de dados, sendo apenas necessária a sua validação durante o *parsing*. Ademais, por observação dos *datasets* em si, pudemos concluir que é possível armazenar certos campos usando tipos de dados de menor tamanho (ex: o identificador de um voo pode ser armazenado como um inteiro, em vez de uma *string*).

2.2 Tipos opacos

Um dos nossos obstáculos principais foi a forma como tipos opacos são implementados em C, que, devido à sua natureza de apontador, exigem uma alocação por instância, gerando significativa ineficiência em *overhead* de alocação. Esta secção descreve como, mantendo o encapsulamento dos tipos, conseguimos definir tipos de dados opacos sem estas limitações.

2.2.1 Estruturas de dados com menos de 8 bytes

Certas estruturas de dados, como datas, horas, códigos de aeroporto, *etc.* contêm menos de 8 bytes de informação. Assim, uma destas estruturas pode ser definida como um inteiro (com o número de bits adequado), evitando-se uma alocação por cada instância desta estrutura. Os métodos relativos a essa estrutura de dados usam uma `union` para aceder aos seus campos, como é visto no seguinte exemplo para datas:

include/utils/date.h:

```
typedef int32_t date_t;
```

src/utils/date.c:

```
typedef union {
    date_t date;

    struct {
        uint16_t year;
        uint8_t month, day;
    } fields;
} date_union_helper_t;
```

2.2.2 Alocação de entidades em *pools*

Como já descrito na secção Catálogos, o nosso projeto utiliza alocação em *pools* de modo a reduzir o *overhead* da alocação genérica de memória. Para tal ser possível com tipos opacos, cada entidade precisa de ter um método que devolve o seu tamanho, por exemplo, `user_sizeof`.

3 Destaques e aspetos a melhorar

O nosso grupo encontra-se satisfeito com o nível de modularidade e abstração que atingimos nesta primeira fase. Dada a estrutura modular do projeto, é importante que a interface de cada módulo apresente um comportamento bem documentado. Por isso, todas as estruturas de dados e métodos encontram-se fartamente documentados, incluindo até exemplos de utilização! Utilizando *Doxygen*, podemos formar páginas HTML com a documentação formatada, como visível na figura abaixo:

Por outro lado, temos noção de que é possível ainda melhorar a modularidade, com a construção de abstrações sobre algumas estruturas de dados da *glib*. No área do encapsulamento, apesar de termos definido *getters* e *setters*, fomos recentemente informados pelos docentes que

Figura 6: Índice da documentação do ficheiro `string_pool.h` na documentação *Doxygen*

deveríamos estar a criar cópias dos valores devolvidos, sendo o uso de `const`, a nossa solução atual, desaconselhado. Pretendemos implementar essa recomendação para a segunda fase de entrega.

Ademais, apesar de, à data de escrita deste relatório sermos, na plataforma de teste, entre os grupos com o trabalho completo, o que usa menos memória, ainda temos algumas ideias de como melhorar o seu uso para a próxima fase. Por último, as nossas ferramentas de desenvolvimento estão bastante completas, com um *Makefile* com geração automática de dependências (para diminuir tempos de compilação), e *scripts* para verificação de *memory leaks* (com supressões), formatação automática de código, e *profiling*.

4 Conclusão

Em suma, nesta primeira fase, procurámos criar alicerces sólidos para a estrutura da nossa aplicação, sobre os quais poderemos facilmente desenvolver a segunda fase, tirando proveito da extensibilidade que o desenvolvimento modular tem para oferecer. Estamos confiantes de que o *dataset* de maior dimensão da 2.ª fase será capaz de caber em memória com as nossas estruturas de dados, mas temos noção de que vamos ter um elevado crescimento no tempo de execução do programa: além do aumento (possivelmente linear) devido ao maior *dataset*, o desempenho diminuirá consideravelmente após implementarmos algumas melhorias ao encapsulamento, nomeadamente o uso de *clones* de estruturas de dados em *getters*.