APS 1: Criando uma interface REST

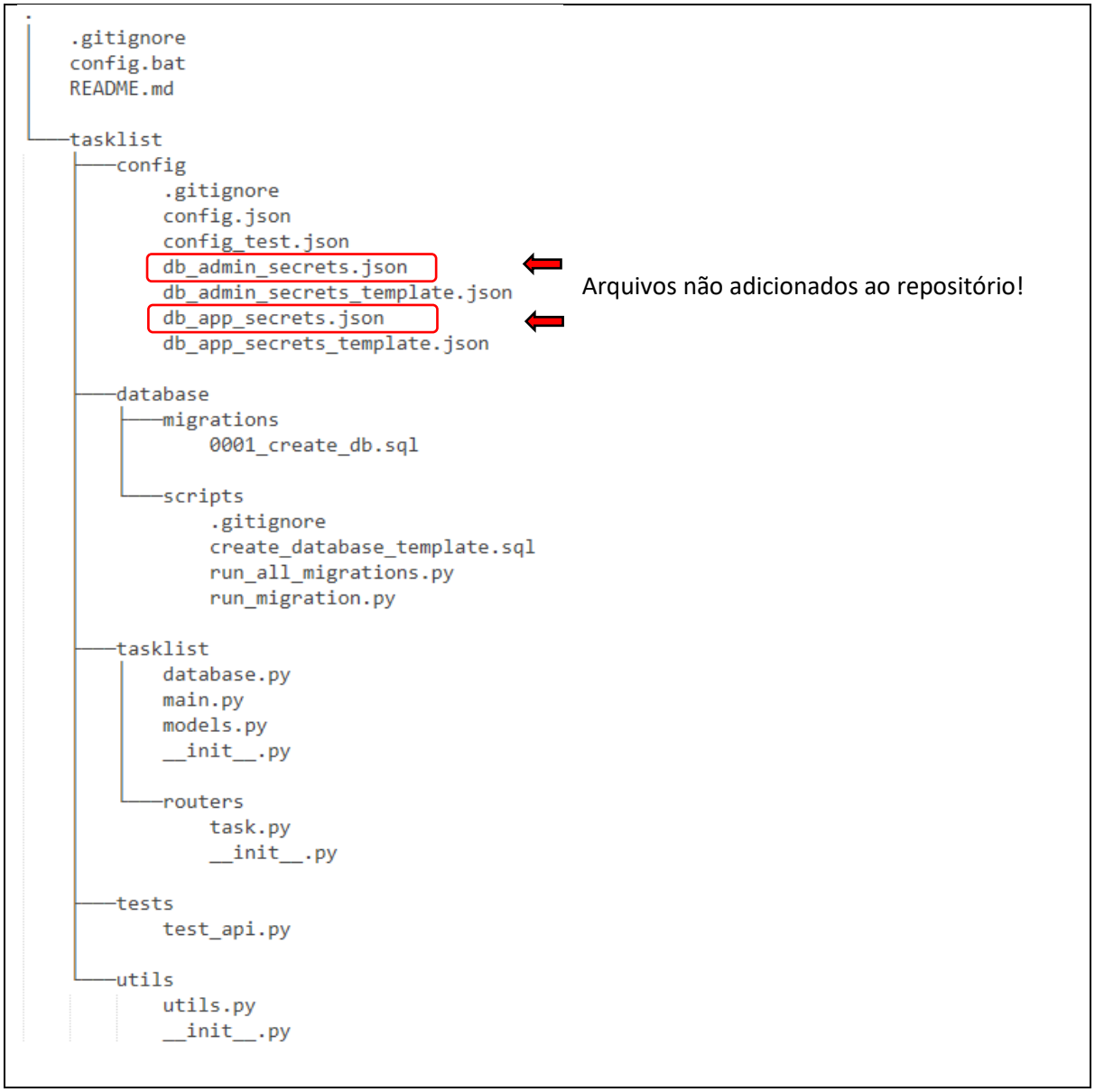
Megadados – Projeto 1

Nesta APS vamos fazer o seguinte:

- Eu fiz várias mudanças na estrutura do projeto e criei a primeira integração com o banco de dados – isso já vai pronto para vocês como exemplo
- Vocês vão construir novas funcionalidades: usuários.

Nova estrutura do código

O diagrama abaixo ilustra a estrutura nova do código (INCLUINDO ARQUIVOS NÃO COMMITTED NO DIRETÓRIO config!):



Nesta primeira parte do handout vamos discutir algumas das mudanças realizadas, e nas partes subsequentes deste documento vamos detalhar os desenvolvimentos mais significativos.

Scripts de configuração do ambiente de desenvolvimento

Na raiz do projeto temos um arquivo `config.bat` que serve para alterar a variável de ambiente PYTHONPATH para localizar os pacotes Python do projeto. É prática comum criar *scripts* que configuram a area de trabalho do desenvolvedor, e estes podem ser executados diretamente ao iniciar uma sessão de trabalho (por exemplo no arquivo `.bashrc` de quem tem Linux ou Mac), ou manualmente.

Portanto, rode esse script (ou crie o seu próprio) para começar o desenvolvimento.

NOTA IMPORTANTE: normalmente esses scripts de configuração do ambiente NÃO SÃO incluidos no repositório (pois cada um tem o seu), no máximo um *template* para novos desenvolvedores pode ser incluido. É melhor neste caso manter um repositório separado, pessoal, para seus arquivos de configuração. Apenas inclui o meu arquivo de configuração aqui como exemplo.

Algumas equipes incluem os *scripts* de configuração de ambiente, contudo: nestes casos vale a pena ter um diretório separado onde todos os desenvolvedores mantem seus arquivos de configuração.

A prática para o *deploy* da aplicação, contudo, é diferente. Vamos voltar a esse ponto em uma etapa futura do projeto.

Criando o banco de dados

Agora já temos a API funcionando, está na hora de introduzir o banco de dados. Poderíamos ter feito as coisas na ordem contrária também, sem problema – somente não o fizemos porque ainda não tínhamos explorado bem os bancos de dados SQL.

Por que não ORM?

A unica razão para não usar um ORM aqui (Django, SQLAlchemy, etc) é conhecer o funcionamento interno de um CRUD. Sabendo como funciona por dentro, fica mais fácil entender as decisões de projeto destes ORM populares: afinal, o que eles fazem é meramente automatizar alguns (não todos) dos procedimentos que estamos construindo aqui. Mas para projetos futuros, recomendo usar um ORM sim!

Criando as contas de admin e de aplicação

Vamos criar duas contas no banco de dados, e dois bancos de dados. Veja o script abaixo (tasklist/database/scripts/create_database_template.sql):

```
DROP DATABASE IF EXISTS tasklist;
CREATE DATABASE tasklist;

DROP DATABASE IF EXISTS tasklist_test;
CREATE DATABASE tasklist_test;

DROP USER IF EXISTS tasklist_admin@localhost;
CREATE USER tasklist_admin@localhost IDENTIFIED BY "senha super dificil";
GRANT ALL ON tasklist.* TO tasklist_admin@localhost;
GRANT ALL ON tasklist_test.* TO tasklist_admin@localhost;

DROP USER IF EXISTS tasklist_app@localhost;
CREATE USER tasklist_app@localhost IDENTIFIED BY "senha impossivel";
GRANT SELECT, INSERT, UPDATE, DELETE ON tasklist.* TO tasklist_app@localhost;
GRANT SELECT, INSERT, UPDATE, DELETE ON tasklist_test.* TO tasklist_app@localhost;

COMMIT
```

Este script deverá ser executado pelo administrador do sistema de gerenciamento de banco de dados. As senhas nunca deverão ser adicionadas (*committed*) ao repositório – este arquivo é só um template!

Os bancos de dados são:

- `tasklist`: banco de dados de produção – este é o banco de dados que será usado quando o serviço estiver funcionando
- `tasklist_test`: banco de dados de teste

As contas são:

- `tasklist_admin`: Conta de desenvolvedor, permite fazer qualquer alteração nos bancos de dados de produção e de teste
- `tasklist_app`: Conta de aplicação – é com essas credenciais que nosso serviço vai rodar, para garantir que não fará nenhuma alteração drástica do banco de dados de produção.

Credenciais de acesso

No diretório tasklist/config temos vários arquivos contendo credenciais de acesso:

- config.json: Contem a informação do *hostname* do sistema de gerenciamento de banco de dados, e o nome do banco de dados de produção. Esta informação não é secreta.
- config_test.json: o mesmo que o config.json, mas para o banco de dados de teste.
- db_admin_secrets.json: *username* e senha da conta de desenvolvedor. ESTA INFORMAÇÃO NÃO SERÁ ADICIONADA AO REPOSITÓRIO.
- db_admin_secrets_template.json: template para o arquivo anterior, esta sim fica no repositório.
- db_app_secrets.json: *username* e senha da conta de aplicação. ESTA INFORMAÇÃO NÃO SERÁ ADICIONADA AO REPOSITÓRIO.
- db_app_secrets_template.json: template para o arquivo anterior, esta sim fica no repositório.

Criando o banco de dados

Nosso banco de dados é muito simples, consiste em uma tabela só no momento.

Como lidar com o ID da tarefa? No código de referência da etapa passada eu defini o identificador da tarefa como um UUID Versão 4. Esta definição apresenta a vantagem de que o identificador é único mesmo entre servidores separados – facilitando uma possível mudança de escala do projeto para uma escala global! Porém, como desvantagem temos o fato de que o UUID vai ocupar de 16 bytes (se armazenado de modo eficiente) a 36 bytes (se armazenado diretamente como uma *string* de 36 caracteres).

O MySQL tem uma função `UUID()` que gera uma *string* de 36 caracteres contendo um UUID. Poderíamos armazenar diretamente esse UUID como um `VARCHAR(36)`, mas isso é perda de espaço de armazenamento. Temos as funções UUID_TO_BIN() e BIN_TO_UUID() para converter entre a representação *string* e a representação eficiente de 16 bytes, que pode ser armazenada em uma coluna do tipo `BINARY(16)`. Para conhecer mais sobre essas operações, consulte o manual do MySQL. O site https://www.mysqltutorial.org/mysql-uuid/ tambem apresenta uma discussão a respeito deste assunto.

Migrações

Após a criação das contas de acesso e dos bancos de dados (vazios, por enquanto), todas as alterações de banco de dados devem ser configuradas na forma de um *script* SQL que será armazenado no diretório tasklist/database/migrations. Isso inclui a criação de tabelas, e toda e qualquer modificação. Veja neste diretório o *script* de criação da nossa tabela inicial.

Observe que não usamos o comando "`USE tasklist`" nesse *script*. Isto é intencional: vamos deixar a especificação do banco de dados para o momento da criação da conexão, no código *Python*. Desta forma podemos usar os mesmos *scripts* de migração para criar nosso banco de dados de produção e de teste.

Em alguns projetos os *scripts* de migração são executados pelos próprios desenvolvedores. Em outros times, pode ser decidido que os *scripts* devem ser sempre executados pelo administrador do banco de dados, que será responsável por analisar cada *script* e julgar se o *script* está correto e não terá um impacto excessivo no desempenho do servidor (lembre-se de que o servidor possivelmente está compartilhado com outros projetos)

Testes

Os testes foram movidos para o diretório tasklist/tests. Estude o arquivo test_api.py. Ele contém agora código para rodar todos os *migrations* para criar a base de dados de teste.

Dependency injection

Observe como funciona o mecanismo de injeção de dependencia do FastAPI. As dependencias que foram declaradas com a classe "Depends" podem ser substituidas usando o dicionário app.dependency_overrides, onde a dependencia a ser substituida é a chave, e a dependencia substituta é o valor. Assim podemos rodar o nosso serviço e testá-lo, mas conectando no banco de dados de teste ao invés do de produção.

Atividade

Agora é com vocês. Implemente as seguintes *features*:

- Criar/Ler/Atualizar/Remover usuário. Não precisa de senha, este projeto é só um exercício.
- Modificar as tarefas para adicionar o usuário responsável pela tarefa.