



**Faculdade de Ciências e Tecnologia**

**Universidade de Coimbra - DEI**

# **ES Project 2 - A three tier application on the cloud**

Mestrado em Engenharia Informática

autores: João Batanete 2009113460, Hugo Silveiro 2013136248

# Introdução

Nesta secção iremos fazer uma breve descrição das tecnologias utilizadas ao longo do projeto.

## **SQLAlchemy**

SQLAlchemy é um ORM(object-relational mapper) para a linguagem Python. Tal como outros ORM's(tais como JPA para Java), introduz uma abstração na criação e manuseio de bases de dados, facilitando a sua utilização em relação à utilização de queries manualmente. Para tal, oferece um paradigma orientado a objetos na representação das tabelas e das suas relações.

À semelhança de JPA, utiliza o paradigma *Data mapper* na sua implementação, pelo que os objetos(entidades) são independentes da base de dados em si, sendo representados de forma mais ligeira e desacoplada, e manuseados através de *getters* e *setters* ou métodos semelhantes. De forma resumida, tratamos cada tabela de uma forma mais semelhante a um Objeto normal de OOP do que como uma tabela real de uma base de dados.

Outro paradigma utilizado em alguns ORM's é o *Active Mapper*, em que cada objeto possui as suas operações de CRUD implementadas nele mesmo, e estas são efetuadas aquando da sua chamada automaticamente.

Neste projeto esta tecnologia foi utilizada para implementar e manusear a cada de dados da aplicação.

## **Flask**

O Flask é uma “micro” *web framework* Python. O termo “micro” é devido ao facto de que Flask não inclui muitas das extensões que *frameworks* do género(tais como Django) possuem, tais como um ORM próprio ou validação de formulários HTML submetidos(neste projeto estas duas funcionalidades foram disponibilizadas pelo SQLAlchemy e a OpenAPI, respetivamente). A implementação base apenas possibilita a implementação da camada de negócio de uma aplicação, pelo que geralmente são utilizadas outras ferramentas para as demais camadas.

A *framework* é no entanto geralmente tida como mais simplista que outras do género, e pode ter muitas destas funcionalidades incorporadas através das muitas extensões e *plugins* existentes para Flask.

Neste projeto esta tecnologia foi utilizada para a camada de negócio(*server*) da aplicação.

## **OpenAPI**

A OpenAPI, conhecida anteriormente como Swagger, define uma especificação para a implementação de APIs REST, tornando mais simples o processo de compreensão dos métodos presentes na mesma, quer pela máquina, quer pelo ser humano. Pode ser comparada a outras tecnologias já existentes, tais como WSDL para os WebServices SOAP.

Embora a sua implementação possa ser útil no caso de aplicações maiores, ou no caso do programador utilizar métodos REST de uma perspectiva *Black Box*(sem acesso ao código fonte destes), introduz, no nosso entender, *overhead* desnecessário no caso de projetos mais pequenos. No entanto, neste projeto foi utilizada para fins pedagógicos, à semelhança da utilização do JavaEE na disciplina de Integração de Sistemas.

Neste projeto esta tecnologia é utilizada para a especificação dos métodos REST implementados no Flask, e chamados a partir da camada de apresentação.

## **React.js**

React é uma biblioteca de JavaScript utilizada para criar interfaces web *client-side*, em conjunto com HTML. Através do seu uso é possível construir páginas web interativas, que se reconfiguram sem a necessidade de carregamento de outra página. Foi desenvolvida pelo Facebook, e é utilizada no próprio website da empresa, bem como outros como Instagram, Netflix e Walmart.

Os componentes React utilizam o conceito de lifecycle, em que são tratados eventos tais como o componente se ter acabado de montar(`componentDidMount`), bem como em intervalos de tempo definidos. Utiliza também o conceito de “estado” de um componente, em que cada vez que este estado é alterado(por recurso ao método `setState`) o componente é renderizado de novo com as suas novas propriedades. Isto possibilita, entre outras possibilidades, atualizar os dados presentes na página em tempo real através de chamadas Ajax, recorrendo a eventos temporais em intervalos periódicos.

Um componente pode igualmente conter componentes “filho” contidos nele mesmo, que representam outros elementos HTML dentro dele. Este modelo possibilita um forte desacoplamento entre os elementos da página, e facilita a implementação de páginas web intuitivas e atrativas para o utilizador, de uma forma muito mais facilitada em relação a JavaScript “puro”, sem recorrer a uma biblioteca semelhante.

Neste projeto esta tecnologia é utilizada na camada de apresentação, em conjunto com páginas HTML obtidas do site Bootstrap. Para as chamadas aos métodos REST, foram utilizadas chamadas Ajax.

## AWS Elastic Beanstalk

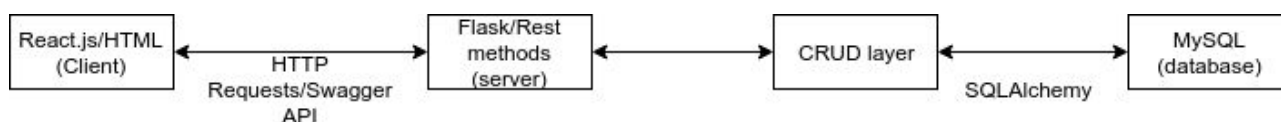
Elastic Beanstalk é um dos vários serviços disponibilizados pela *cloud* AWS. Permite hospedar aplicações ou serviços na *cloud* de forma simplificada, oferecendo um nível de abstração por cima do sistema operativo e do ambiente utilizado.

Possibilita aos *developers* ter controlo total sobre vários aspetos do sistema, tais como a plataforma ou tecnologia utilizada(neste caso Python 3), os recursos utilizados e o balanceamento de carga. Esta última opção pode ser implementada recorrendo à funcionalidade auto-scaling, que irá iniciar ou suspender instâncias do sistema consoante o necessário ao funcionamento normal deste, com limites e políticas de balanceamento definidas pelo utilizador.

Neste projeto o serviço foi utilizado para hospedar a aplicação e para efetuar testes de balanceamento de carga.

## Arquitetura

O sistema encontra-se estruturado da seguinte forma:



A camada de apresentação comunica com a de negócio(server) através de chamadas aos métodos REST(em praticamente todos os casos via AJAX). A camada de negócio lê os argumentos necessários a partir do JSON e executa chamadas ao layer de operações CRUD. Este, por sua vez, realiza operações na base de dados usando o ORM SQLAlchemy e devolve(quando necessário) os resultados à componente de negócio convertidos para JSON, que por sua vez também irá devolver o resultado em JSON ao cliente(bem como o código de sucesso/falha da operação).

Assim, é garantido o desacoplamento total de todos os componentes da aplicação, uma vez que nem a camada de negócio nem o cliente manuseiam entidades SQLAlchemy ou interagem com a base de dados diretamente.

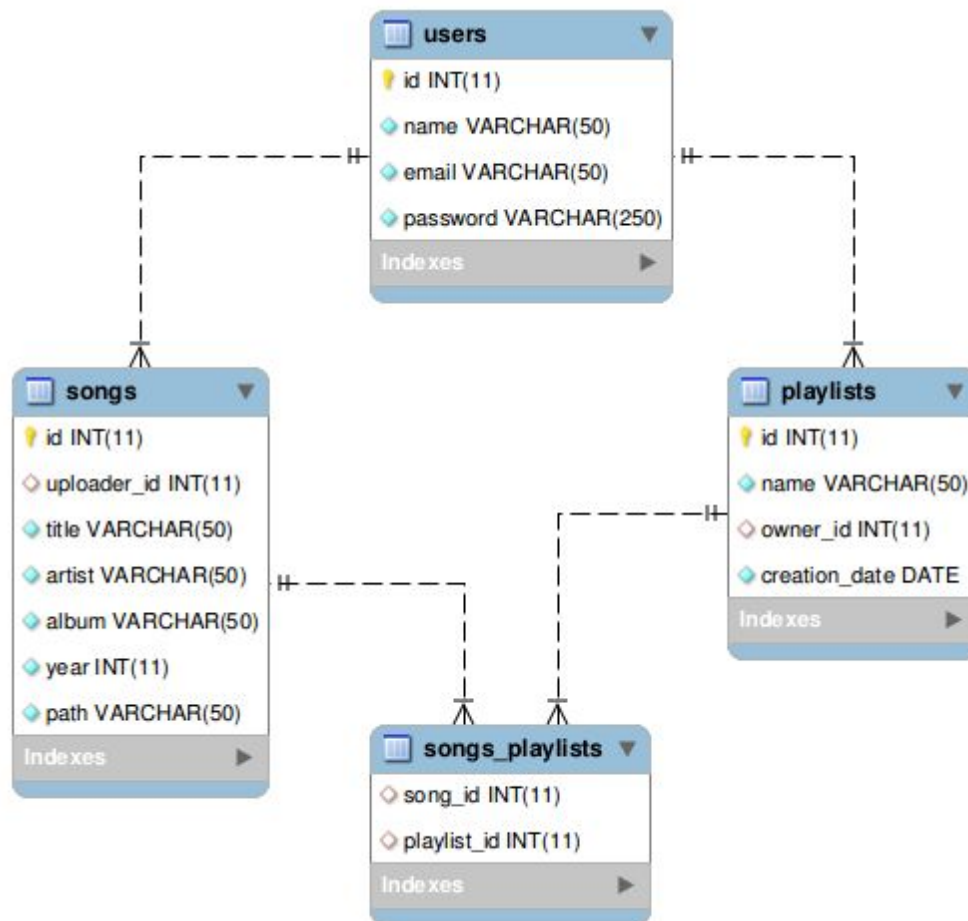
Nas secções seguintes iremos descrever o funcionamento de cada componente com um nível maior de detalhe.

## Componentes do sistema

Nesta secção será descrito o funcionamento e as decisões tomadas ao longo da implementação das várias componentes do projeto.

### Camada de dados(SQLAlchemy/MySQL) e CRUD:

Para a criação e manuseio da base de dados foi utilizado o ORM SQLAlchemy para Python. O esquema da base de dados criada encontra-se ilustrado abaixo.



### **Decisões-chave tomadas na implementação deste componente:**

- As entidades do ORM são manuseadas a partir da camada CRUD, que devolve os resultados das queries na forma de dicionários Python: Não são retornadas entidades diretamente para manter o isolamento dos componentes.

- As pesquisas são realizadas, quando possível, utilizando um get pelo id da tabela correspondente, fazendo assim uso dos índices associados para melhorar a performance das pesquisas.
- Embora isso não tenha sido realizado(devido a não ser o foco do projeto), a performance de queries envolvendo mais que uma coluna poderia ser também melhorada criando mais índices na base de dados. No entanto, isto iria também tornar as operações de criação e apagamento mais lentas, pelo que numa aplicação real teria de se estudar o trade-off a realizar para um volume considerável de clientes.
- Para as operações de Create, Delete e Update, é verificado se é possível realizar a operação pretendida através de blocos try/catch. Caso tal não seja possível(exemplo: tentar adicionar uma música a uma playlist, quando a mesma já se encontra contida nela), é feito rollback na base de dados e o método retornará False à camada de negócio.

### **Camada de negócio(Flask/OpenAPI):**

Nesta camada foi realizada a especificação e implementação dos métodos REST chamados pelo cliente.

Cada método recebe dados em JSON(com a exceção dos métodos para realizar upload de músicas e o de *login*, que utilizam forms) e devolve o output esperado pelo cliente.

Em situações em que o cliente não tem permissão para chamar o método(exemplo: tenta apagar uma playlist que não é dele) é enviado um código de erro 403. Caso os recursos a que pretendemos aceder não estejam disponíveis(exemplo: chamar um método com os parâmetros errados), é devolvido um código 404.

O servidor implementa um sistema de autenticação recorrendo a *sessions* Flask. Quando o utilizador efetua o login, estas irão ficar criadas até ele realizar o logout. Tentativas de aceder ao menú principal da aplicação(ou a qualquer método REST tirando o de login ou de criação de conta) sem estar logado resultam num redirecionamento para a página de *login*. Estas sessions são algo difíceis de configurar em conjunto com o swagger, mas permitiram a incorporação mais fácil da aplicação no AWS.

Nota: A lista de métodos REST disponíveis e a sua especificação encontra-se descrita no ficheiro swagger.yaml, na pasta swagger.

### **Carregamento de ficheiros:**

O armazenamento dos ficheiros foi efetuado recorrendo a um Bucket S3 do Amazon Web Services.

### **Encriptação de passwords:**

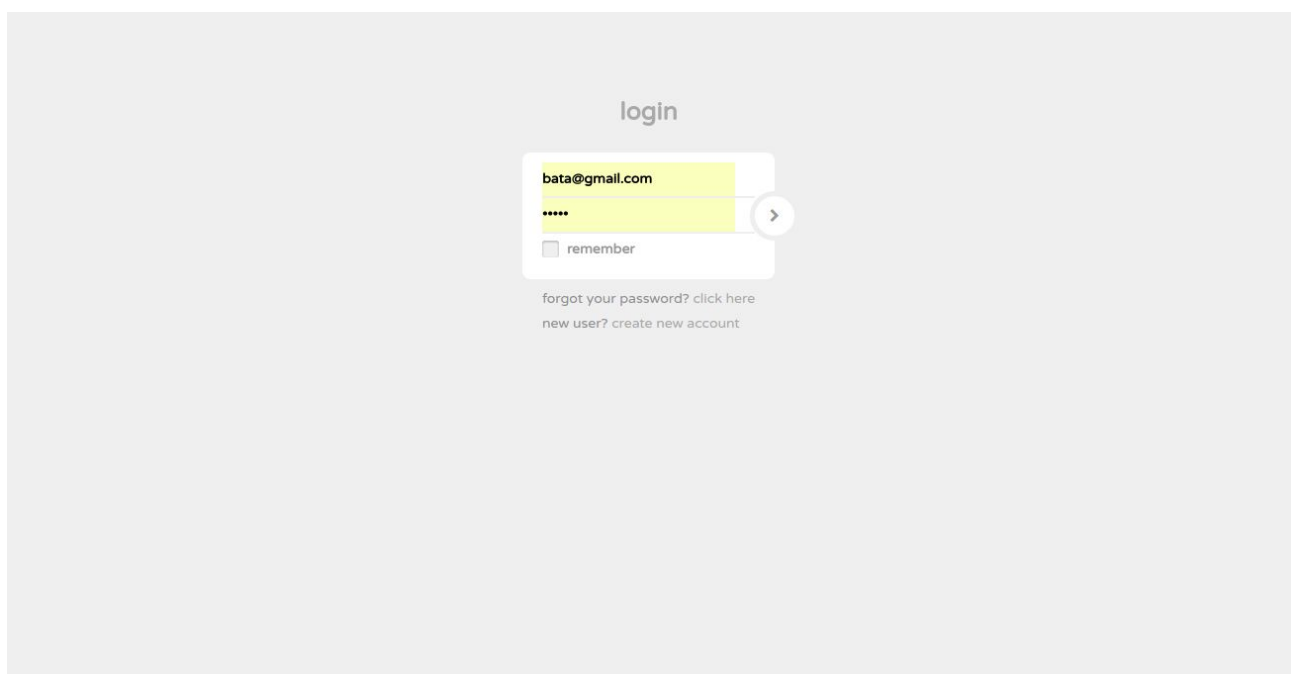
Para a encriptação de passwords na base de dados, foi utilizado o algoritmo SHA-1. Embora este algoritmo já se encontre algo ultrapassado quando comparado a alternativas como MD5 e SHA-256, sendo este um projeto de natureza académica o grupo considerou-o suficiente.

### **Camada de apresentação(React.js/HTML):**

Como já foi mencionado, a camada de apresentação foi implementada usando React.js.

A aplicação possui três páginas Web. Nesta secção iremos fazer uma descrição mais detalhada de cada uma.

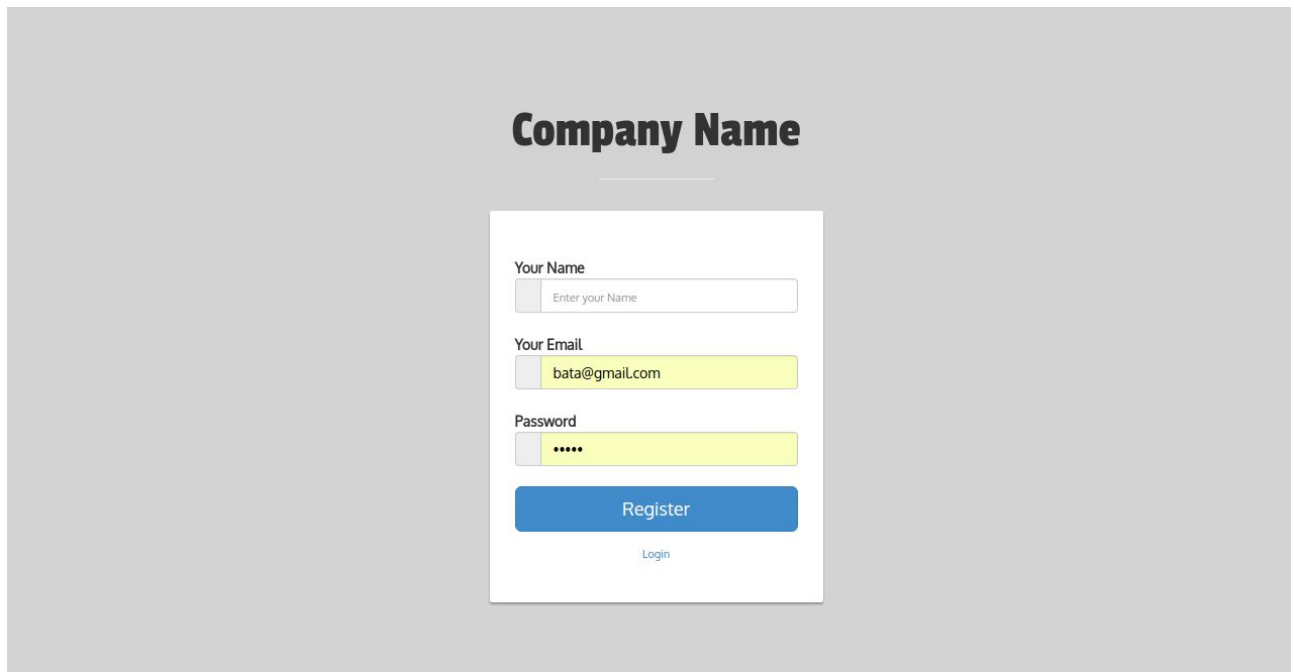
### **Menu de Login(login.html)**



Este menu é utilizado para nos autenticarmos na aplicação. Faz uso de um formulário HTML que chama o método login do flask. Caso o utilizador carregue no botão “new user” é reencaminhado para o menu de criação de conta.

Uma mensagem de erro é apresentada caso as credenciais de início de sessão estejam incorretas. De notar que o campo “username” se refere ao email do utilizador, e não ao nome.

## Menu de registo(create\_account.html)

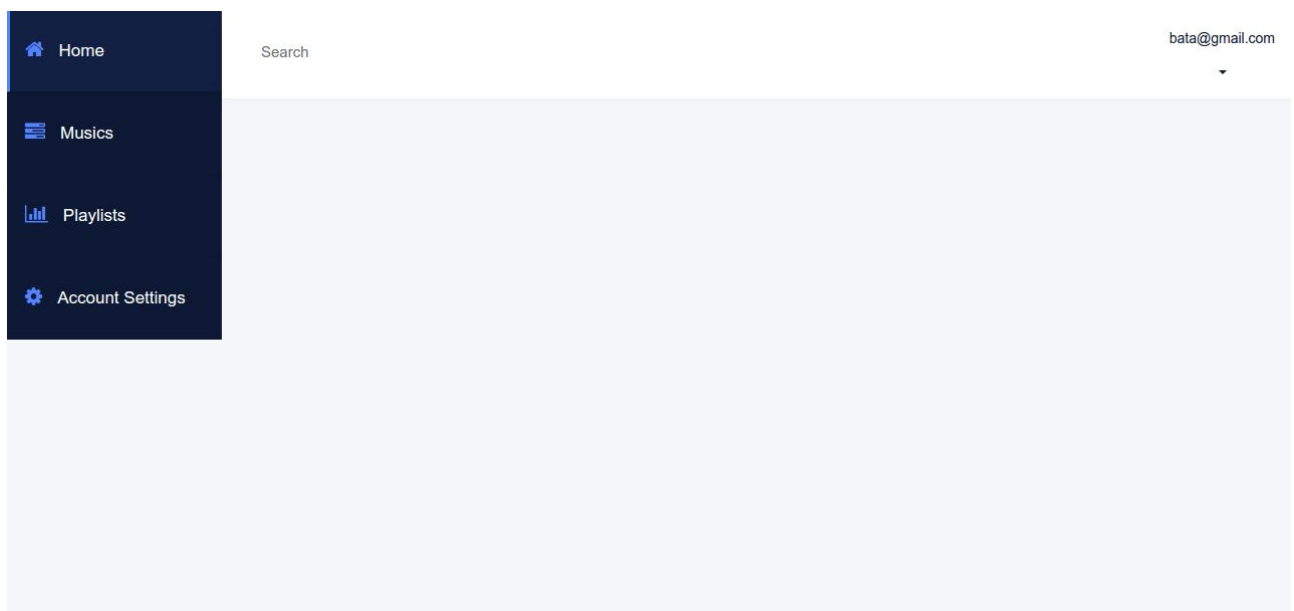


The registration form is centered on a light gray background. It features a title 'Company Name' in bold black text. Below the title is a white form box containing three input fields: 'Your Name' with a placeholder 'Enter your Name', 'Your Email' with the value 'bata@gmail.com', and 'Password' with masked characters '\*\*\*\*\*'. A blue 'Register' button is positioned below the password field, and a blue 'Login' link is located at the bottom of the form box.

Este menu é utilizado para criar novas contas. Utiliza uma chamada AJAX ao método `create_account` do servidor. Embora utilize um componente React, nesta página ainda não é feito uso das funcionalidades da biblioteca, e poderia ter sido implementada facilmente com um form, à semelhança do login.

Uma mensagem de erro é apresentada no browser caso uma conta com o email indicado já exista.

## Menu principal(menu.html)



Nesta página são realizadas a maior parte das operações da aplicação. O menu apenas é acessível através do menu de login, ou caso o utilizador já possua uma sessão ativa.

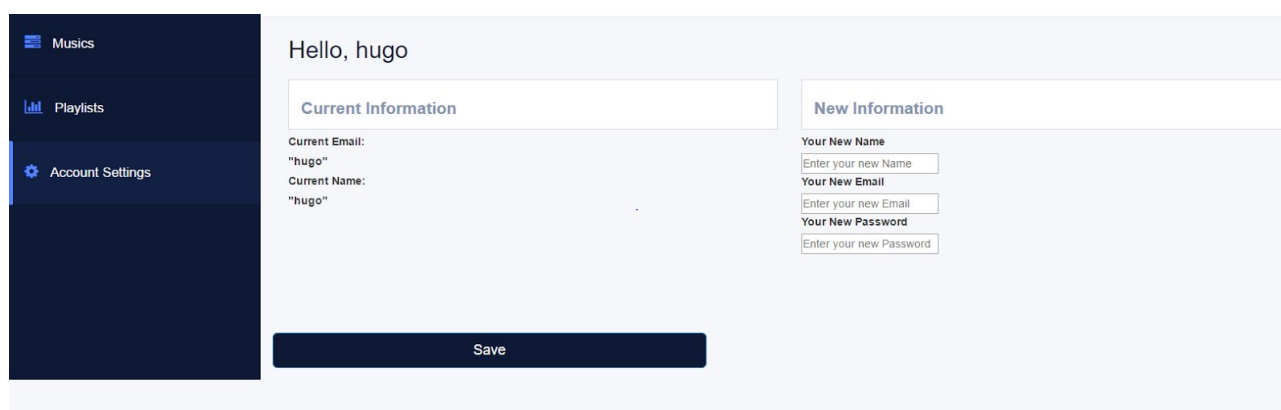


O menú possui 3 submenus, que são acedidos através da barra lateral presente na imagem anterior. Em qualquer altura é possível realizar o logout clicando na seta no canto superior direito do ecrã. Isto é efetuado através de um form que chama o método `logout` do servidor e reencaminha o utilizador para a página de login.

Em seguida iremos detalhar o conteúdo e utilização de cada submenu.

### Submenu “Account Settings”

No caso do submenu referente ao “Account Settings” o utilizador tem acesso ao seu nome e email actual assim como a campos para proceder à alteração destes parâmetros. Caso decida enveredar pela modificação dos parâmetros associados à conta é invocado o método `edit_account` que vai alterar os dados na base de dados.

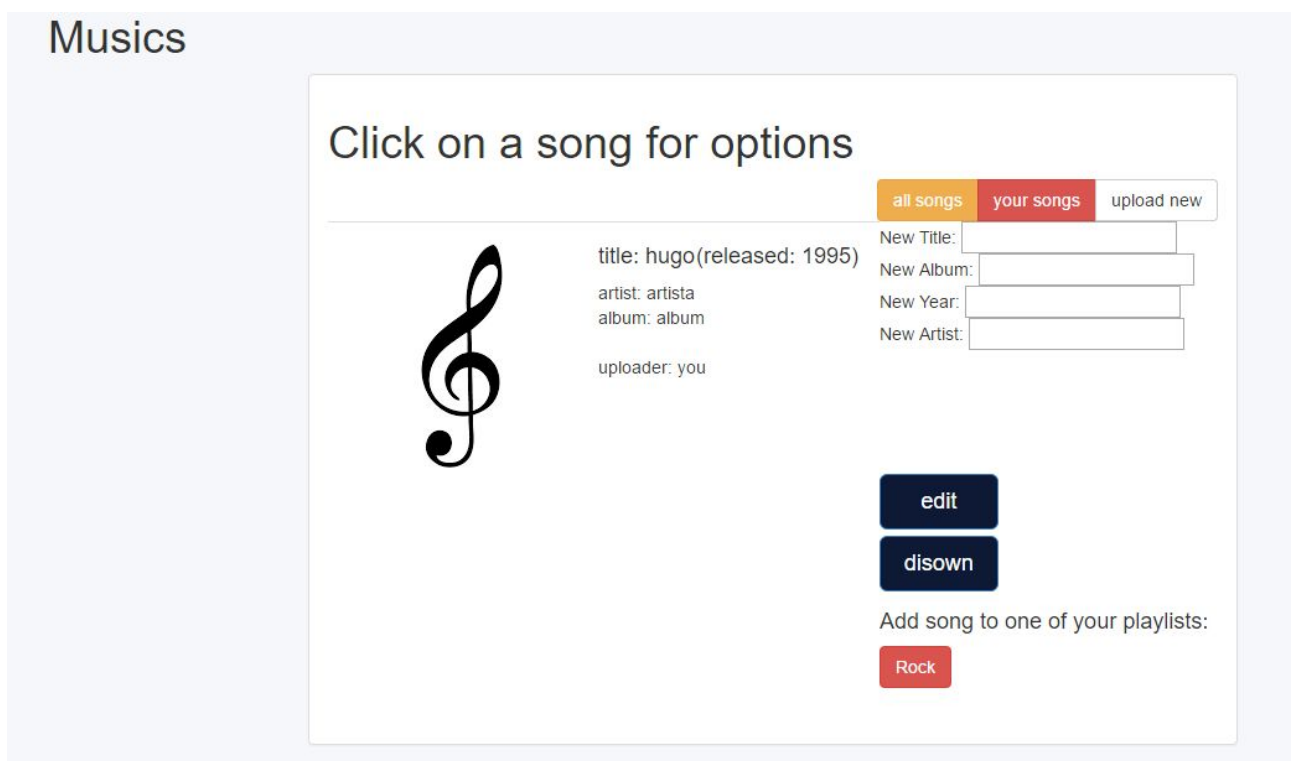
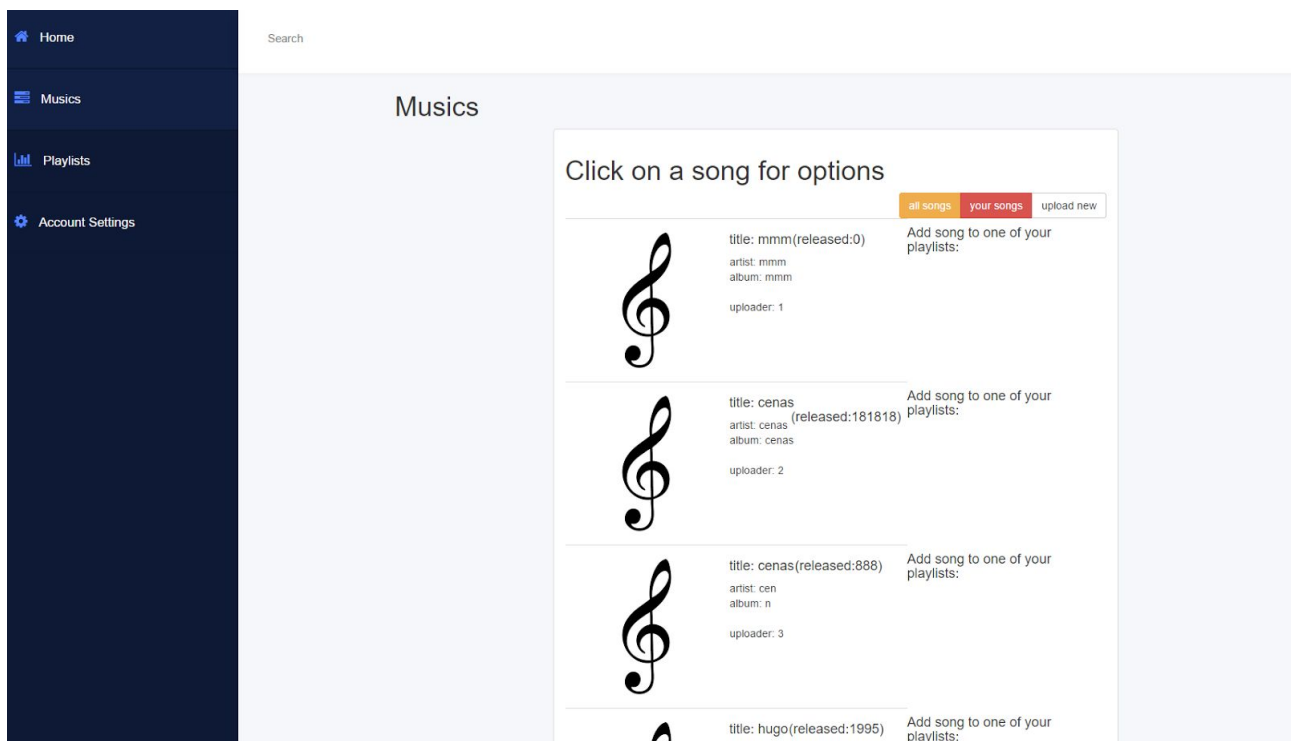


Account Settings page layout:

- Left Sidebar:** Musics, Playlists, Account Settings (selected).
- Greeting:** Hello, hugo
- Current Information:**
  - Current Email: "hugo"
  - Current Name: "hugo"
- New Information:**
  - Your New Name:
  - Your New Email:
  - Your New Password:
- Save Button:** Save

### Submenu “Musics”

Em relação ao submmenu “Musics”, é apresentado ao utilizador uma listagem de todas as músicas da plataforma ou, em caso de escolha das músicas próprias (como se pode ver pela imagem), faz uma listagem das suas músicas. No caso de as músicas serem da autoria do utilizador que está com sessão activa, se clicar na música irão aparecer opções disponíveis para o utilizador relativas à mesma (como editar os detalhes, adicioná-la a uma playlist sua ou dar “disown” à música, isto é, retirar o rasto de quem deu upload à mesma. Para além destas listagens também é possível adicionar novas músicas ao sistema, preenchendo todos os detalhes pedidos no ato da submissão de uma música nova. Por fim também está embutido nesta página uma *search bar* para procurar por uma música/artista específico.



### Submenu Playlists

No submenu “Playlists”, é apresentado uma lista de playlist próprias (da autoria do utilizador que está logado) que podem ser ordenados tanto por nome como pela data de criação. Cada playlist irá mostrar as músicas que fazem parte da mesma e, caso se clique numa música, esta é eliminada da playlist. Também é possível fazer operações de edição da playlist assim como a operação de apagá-la da plataforma. Por fim, temos também um botão para a criação de uma nova Playlist, bastando para isso escrever o nome desejado

## Playlists

Playlist Name:

Create Playlist


Click on the music to remove

Ascending(name)

Descending(name)

Ascending(date)

Descending(date)



Name: Rock(2017-04-22)

New name:

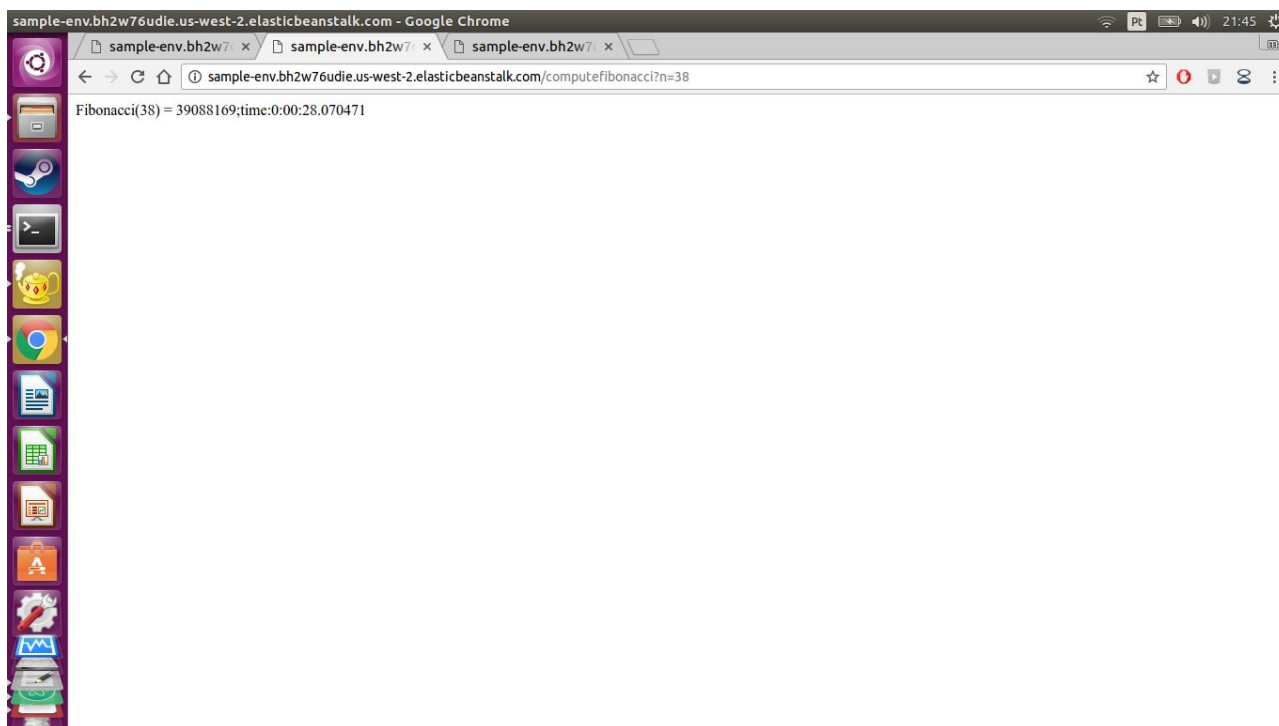
edit

delete

## Testes realizados ao balanceamento de carga do Elastic Beanstalk

Para realizar testes ao auto-scaling deste serviço utilizámos o método REST fornecido pelo professor que implementa uma versão lazy da sequência de Fibonacci. O método foi ligeiramente modificado para indicar no browser o tempo de execução. Para aumentar a carga, limitámo-nos a chamar o método pelo browser num número maior de separadores. Foi utilizado n com valor 38, que é um valor grande o suficiente para gerar carga no servidor durante um intervalo de tempo significativo, mas não para sobrecarregar o servidor para além do tempo da experiência.

O Beanstalk cria novas máquinas quando a carga de esforço do servidor se mantém alta durante um certo intervalo de tempo, e usando uma métrica estabelecida pelo utilizador. O grupo optou por usar a percentagem de utilização do CPU, e iniciar uma instância nova de cada vez que a carga média no CPU de todas as instâncias atualmente ativas ultrapassar 80%, sendo esta verificação feita a cada 10 segundos. Os testes foram realizados com o número de máquinas pretendido a correr, após ser forçada a sua execução com uma quantidade de carga suficiente.



Foi verificado que, para esta tarefa em específico, o Beanstalk não consegue tirar muito uso de paralelização de recursos. Tal como podemos ver na figura seguinte, não existe variação significativa no tempo de execução.

Número de separadores/tarefas	Número de máquinas a funcionar	Tempo médio de execução(por separador)
1	1	28.070471
1	2	27.984736
1	3	28.101372

No entanto, quando aumentamos o número de separadores para 3, o Beanstalk torna-se mais útil.

Número de separadores/tarefas	Número de máquinas a funcionar	Tempo médio de execução(por separador)
1	1	53.120654
2	2	43.784514
3	3	32.487131

Os resultados apontam para o escalonamento ser realizado em função do número de requests, em lugar de tentar paralelizar cada request individualmente. Isto torna o Elastic

Beastalk potencialmente mais apropriado para realizar o papel de servidor web, uma vez que estes servidores geralmente possuem tarefas menos complexas, embora em número potencialmente muito maior. Para aplicações em que o bottleneck se encontra num ponto específico difícil de paralelizar, os ganhos de performance serão, em princípio, menores.

## Referências

- slides da disciplina
- tutoriais do site AWS
- <https://en.wikipedia.org/wiki/SQLAlchemy>
- [https://en.wikipedia.org/wiki/Data\\_mapper\\_pattern](https://en.wikipedia.org/wiki/Data_mapper_pattern)
- [https://en.wikipedia.org/wiki/Active\\_record\\_pattern](https://en.wikipedia.org/wiki/Active_record_pattern)
- [https://en.wikipedia.org/wiki/Flask\\_\(web\\_framework\)](https://en.wikipedia.org/wiki/Flask_(web_framework))
- <https://github.com/OAI/OpenAPI-Specification>
- [https://en.wikipedia.org/wiki/React\\_\(JavaScript\\_library\)](https://en.wikipedia.org/wiki/React_(JavaScript_library))
- <http://bootsnipp.com/tags/login>