

INSTITUTO SUPERIOR TÉCNICO

ARQUITECTURA DE SISTEMAS DE INTERNET

Web Service - Gestão de ocupação de salas no IST

Autores:

José Dias 75847
Ricardo Miranda 75757

Professor:

Prof. João Silva

30 de Janeiro de 2017

1 Introdução

O Instituto Superior Técnico, enquanto instituição de ensino superior, disponibiliza aos seus alunos salas que estes podem ocupar em horário livre. Este projecto destina-se ao desenvolvimento de um *Web Service* de controlo de entradas e saídas de salas. Para além do objectivo de implementação das funcionalidades, pretende-se também que o projecto desenvolvido tenha uma API REST bem definida no servidor. Isto aumenta a interoperabilidade do *Web Service*, permitindo que o cliente seja desenvolvido em várias plataformas.

2 Arquitectura

O sistema é constituído por um servidor centralizado, executado no Google App Engine, e por um conjunto de módulos que a ele se conectam. Os clientes desenvolvidos em plataformas diferentes de *browser* (representados como Client X) podem conectar-se ao sistema apesar de não terem sido implementados. Isto deve-se à implementação de uma API REST adequada, bastando apenas que suportem bibliotecas de HTTP e String (formato JSON). Para armazenar a tabela de *users* (estudantes) registados no sistema e de salas disponíveis para ocupação foi usado o Google DataStore NDB. A interface que foi implementada para um cliente num *browser* realiza os pedidos ao servidor através de JavaScript e usando AJAX (biblioteca xhttp). O pedido pode ser concretizado através do pedido HTTP ou adicionalmente acompanhado do envio de uma *string* no formato JSON. No servidor, foi utilizada a ferramenta Bottle do Python que permite associar um método a um URI e a um tipo de pedido REST (GET, POST, PUT e DELETE). Deste modo, é possível associar cada função do sistema a um caminho como representado na figura 3. O servidor, para responder a um pedido e verificar a validade dos dados executa *queries* na base de dados. Quando se trata de um pedido GET, o servidor responde com o envio de uma string no formato JSON. No cliente, a resposta é recebida em modo assíncrono pelo AJAX do JavaScript.

Quando um administrador pretende adicionar salas deverá ver a lista de todos os espaços existentes no Técnico. Para tal, o servidor realiza um conjunto de pedidos HTTP (método GET) através da API FenixEdu.

A figura 1 ilustra a arquitectura deste projecto.

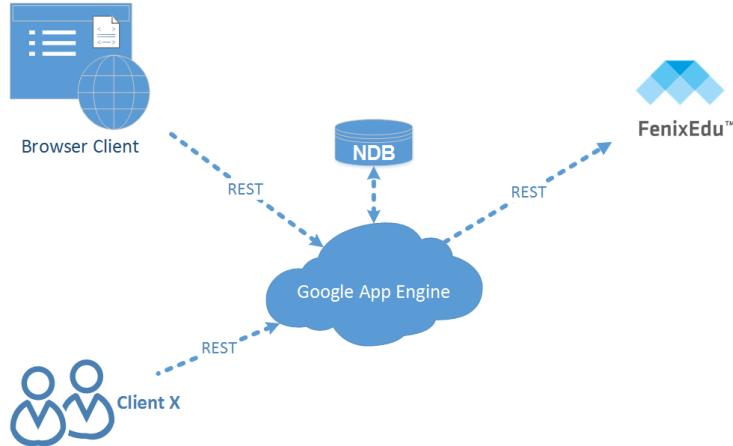


Figura 1: Arquitetura do Web Service desenvolvido.

Cada estudante regista-se uma única vez na base de dados com um nome de utilizador. A esse estudante é atribuído um identificador único que é utilizado para a sua identificação na comunicação com o sistema. Cada estudante pode consultar a lista de espaços disponíveis, verificar o nome de utilizador dos alunos que estão a ocupar um determinado espaço e ocupá-lo, caso em que terá de fazer o *Check In*. A partir do momento em que um estudante faz *Check In*, o seu nome de utilizador constará na lista de alunos que a ocupar essa sala. Sendo que, ao realizar *check out* será retirado dessa lista. Se um estudante estiver numa sala e fizer *check in* num outro espaço, esse aluno será automaticamente retirado da sala em que estava inscrito. Um estudante pode procurar um amigo no sistema, recebendo informação sobre se esse nome está registado, se se encontra dentro de uma sala e em caso afirmativo, o nome da sala.

Um administrador tem também um número de identificação (0). Este pode adicionar novas salas à lista de espaços que os alunos podem ocupar, ver a lotação actual de cada sala, assim como, os alunos que as ocupam. Para disponibilizar ao administrador a lista dos espaços de todos os Campus do Técnico é utilizada a API do Fenix, disponível publicamente em <http://fenixedu.org/dev/api/>. A lista é apresentada de forma hierárquica: campus, edifícios, pisos e salas. A informação sobre cada *layer* é obtida através de um pedido HTTP, GET, com URI /SPACES/ID.

3 REST

3.1 Conceito REST

REST denominado por **R**epresentational **S**tate **T**ransfer é um modelo de arquitectura para o desenvolvimento de *Web Services*. REST integra o modelo *Resource API* e apresenta-se como sendo uma alternativa aos modelos *RPC API* e *Message API*. Um sistema REST terá que obedecer algumas restrições inerentes ao próprio modelo REST:

- **Client-server**, este modelo pode ser caracterizado principalmente pela separação de responsabilidades. O cliente não tem que conhecer a estrutura de armazenamento ou de funcionamento do servidor. Esta separação permite que o cliente utilize um serviço sem conhecer características específicas das tarefas que pretende realizar. A escalabilidade e a portabilidade (da *user-interface*) apresentam-se como algumas das vantagens deste modelo *client-server*;
- **Stateless**, quando o cliente faz um pedido, esse pedido deverá conter toda a informação necessária para que o servidor consiga interpretar a tarefa requisitada pelo cliente. Não deverá existir qualquer auxílio no processamento do pedido através de informações guardadas de sessões anteriores;
- **Cacheable**, esta característica pretende melhorar a eficiência, a escalabilidade e a rapidez de todo o sistema. Ao identificar uma resposta (a um pedido) como *cachable*, é permitido que essa informação seja utilizada novamente em resposta equivalentes. No entanto a informação em *cache* poderá ser diferente da informação mais recente, pelo que será necessário um processamento adicional para actualizar essa informação;
- **Uniform interface**, este princípio simplifica a arquitectura do sistema através da implementação de uma interface uniforme entre os diversos componentes do sistema. Este princípio é alcançado através de quatro métodos: Identification of resources; Manipulation of resources through representations; Self-descriptive messages; Hypermedia as the engine of application state (HATEOAS).
- **Layered system**, a estruturação do sistema em camadas implica que cada camada só “conhece” as camadas imediatamente adjacentes. Esta

criação de fronteiras permite que as camadas sejam desenvolvidas isoladamente. Um possível cenário resultante desta estruturação ocorre quando um cliente se conecta com um servidor. O cliente não sabe se se está a ligar ao *end server* ou a um servidor intermediário;

- **Code on demand**, mecanismo através do qual um serviço consegue estender algumas funcionalidades no cliente. O cliente faz o *download* e execução de código na forma de *applets* ou *scripts*.

Um serviço que viole uma das características referidas acima não pode ser considerado com um serviço REST.

3.2 Implementação

O desenvolvimento de *web service* APIs considerando o modelo REST e que sejam implementadas segundo o protocolo HTTP podem ser caracterizadas por:

- URL, por exemplo `http://www.ist.com/alameda/north/`
- Internet media type, que define o tipo de dados ou o formato da informação
- Métodos HTTP, por exemplo GET ou POST.

A REST API deste projecto foi desenvolvida de acordo com as características apresentadas em 3.1 e as particularidades da implementação do modelo REST no protocolo HTTP.

Acção	HTTP	URL
Registrar estudante	POST	<code>/register</code>
Adicionar uma sala	POST	<code>/user/0/search/addroom</code>
Encontrar um amigo	GET	<code>/find/<friend_username></code>
Listar Salas	GET	<code>/user/<user_id>/listrooms</code>
Listar Estudantes	GET	<code>/user/<user_id>/liststudents/room/<room_id></code>
Fazer CheckIn/Out	POST	<code>/user/<user_id>/<room_id>/<in_or_out></code>

Tabela 1: REST API

O servidor envia a resposta aos comandos REST da tabela 1 em formato JSON. Sempre que ocorre uma mensagem de erro, existe um campo ErrorCode seguido de uma *string* com a indicação do erro que ocorreu (utilizador inexistente, formatação errada, etc). Quando se trata de uma resposta que inclui um vector de vários elementos, como a lista de salas ou a lista de estudantes, e esta está vazia, retorna-se o vector vazio ([]). Um exemplo da resposta do servidor a um pedido da lista de salas pode ser observado na figura 2. No lado do cliente, o JavaScript interpreta o formato recebido verificando primeiramente a condição de erro e posteriormente a de vector vazio.

```
# AVAILABLE ROOM LISTING
@bottle.get('/user/<user_id>/listrooms')
def list_rooms(user_id):
    if user_id == '0':
        rooms = RoomR.query().fetch()
        if not rooms:
            ret = {"containedRooms": []}
        else:
            array = []
            for r in rooms:
                students = StudentS.query(StudentS.room_id == r.id)
                array.append({"name": r.name, "id": r.id, "count": students.count()})
            ret = {"containedRooms": array}
    else:
        find_user = StudentS.query(StudentS.id == int(user_id)).count()
        if (find_user):
            rooms = RoomR.query().fetch()
            if not rooms:
                ret = {"containedRooms": []}
            else:
                array = []
                for r in rooms:
                    array.append({"name": r.name, "id": r.id})
                ret = {"containedRooms": array}
        else:
            ret = {"ErrorCode": "User ID not found!"}

    return json.dumps(ret, sort_keys=True)
```

Figura 2: Método implementado para a listagem de salas.

```

# HOMEPAGE
@bottle.route('/init')
def init(): ...

# STUDENT REGISTER PAGE
@bottle.get('/register')
def student(): ...

# STUDENT REGISTER RESULT PAGE
@bottle.post('/register')
def register(): ...

# USER HOMEPAGE - ADMIN (ID: 0) OR STUDENT
@bottle.route('/user/<user_id>')
def user_home(user_id): ...

# AVAILABLE ROOM LISTING
@bottle.get('/user/<user_id>/listrooms')
def list_rooms(user_id): ...

# FIND A FRIEND METHOD
@bottle.get('/find/<friend_username>')
def search_friend(friend_username): ...

# CHECK IN | CHECK OUT PAGE
@bottle.route('/user/<user_id>/<room_id>')
def check_room(user_id, room_id): ...

# ROOM OCCUPANCY INFO - LIST STUDENTS
@bottle.get('/user/<user_id>/liststudents/room/<room_id>')
def list_students(user_id, room_id): ...

# CHECK IN | CHECK OUT METHOD
@bottle.post('/user/<user_id>/<room_id>/<in_or_out>')
def check_in_or_out(user_id, room_id, in_or_out): ...

# FOR ADMIN TO SEARCH FOR ROOMS IN TECNICO (W/ FENIX API)
@bottle.get('/user/0/search/<room_id>')
def search(room_id): ...

# ADD A ROOM METHOD
@bottle.post('/user/0/search/addroom')
def add_room(): ...

# Print information error for request to non-defined URIs
@bottle.error(404)
def error_404(error): ...

```

Figura 3: Métodos implementados.

4 Classes e Estruturas

Como referido em 2 o Google DataStore NDB é responsável pelo armazenamento e gestão da informação necessária ao funcionamento de todo o *web service*. Na figura 4 é possível observar as classes implementadas. A classe *StudentS* compreende:

- *username* - Elemento do tipo *String* que contém o nome de um estudante;
- *id* - Elemento do tipo *Integer* que contém o identificador de um estudante;
- *room_id* - Elemento do tipo *String* que contém o identificador da sala em que o aluno está.

Já a classe *RoomS* é constituída por:

- *name* - Elemento do tipo *String* que contém o nome de uma sala;
- *id* - Elemento do tipo *String* que contém o identificador de uma sala.

```
class StudentS(ndb.Model):
    username = ndb.StringProperty()
    id = ndb.IntegerProperty()
    room_id = ndb.StringProperty()

class RoomR(ndb.Model):
    name = ndb.StringProperty()
    id = ndb.StringProperty()
```

Figura 4: Classes utilizadas.

5 User-Interface

A figura 5, representa a página de apresentação do projecto. Inicialmente, não existe nenhum estudante registado nem nenhuma sala disponível, razão pela qual a lista das salas na área do administrador na figura 7 está vazia.



Figura 5: Homepage do projecto.

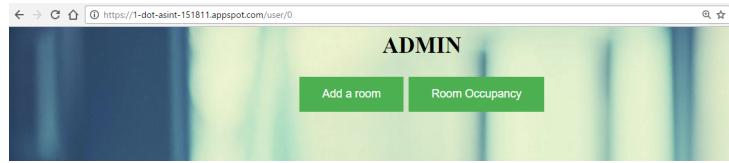


Figura 6: Homepage Admin.

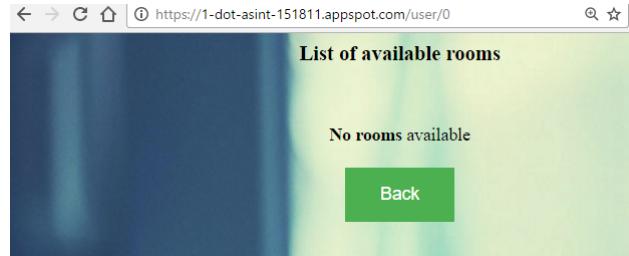


Figura 7: Lista de salas disponíveis [admin].

Para adicionar uma sala, o administrador clica no botão indicado para tal (figura 6) e é redirecionado para uma página que disponibiliza a lista de Campus do Técnico (figura 8). Neste momento começa a utilização da API do Técnico com um pedido GET /SPACES, seguido, de pedidos GET /SPACES/ID, consoante o campus escolhido. As figuras 8, 9, 10, 11 mostram a sequência da escolha Alameda->Torre Norte->-1 onde por fim é escolhida a sala E3 (figura 12a).

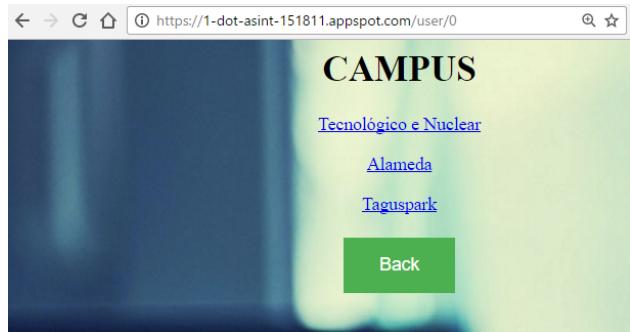


Figura 8: Procurar sala através da API do técnico: selecionar Campus.

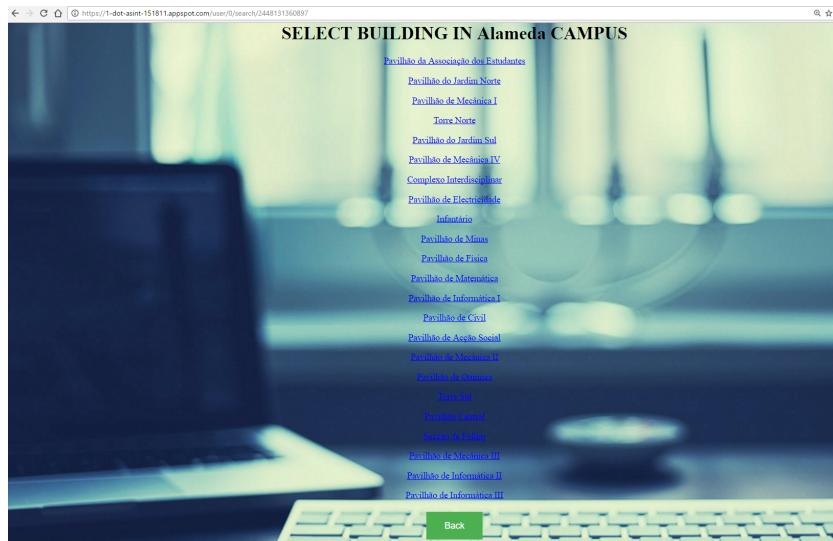


Figura 9: Procurar sala através da API do técnico: edifícios dentro do Campus Alameda.

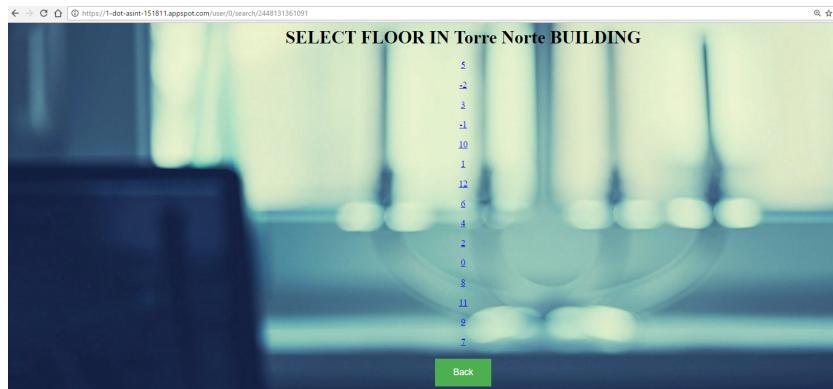


Figura 10: Procurar sala através da API do técnico: pisos entro do edifício Torre Norte.

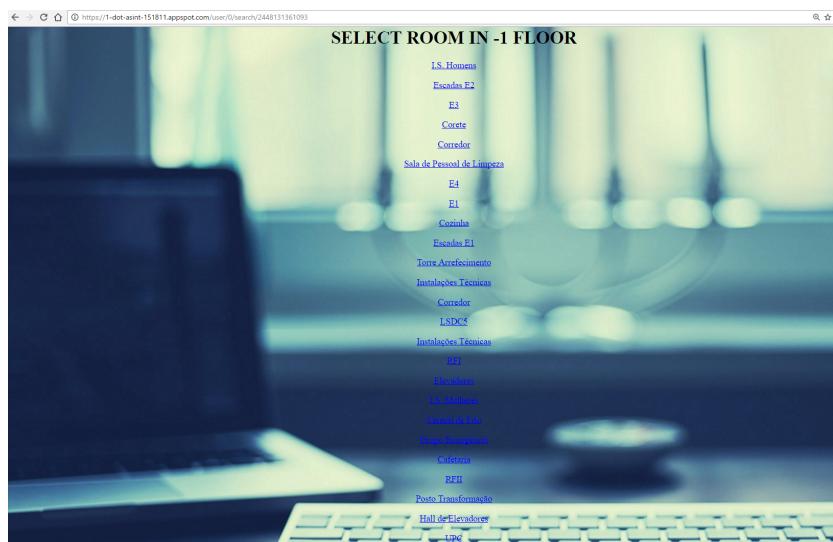
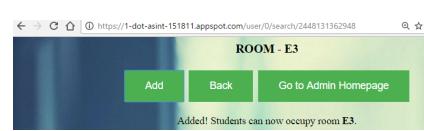


Figura 11: Procurar sala através da API do técnico: salas no piso -1 do edifício Torre Norte.



(a) Página de uma sala



(b) Resultado da disponibilização de uma sala

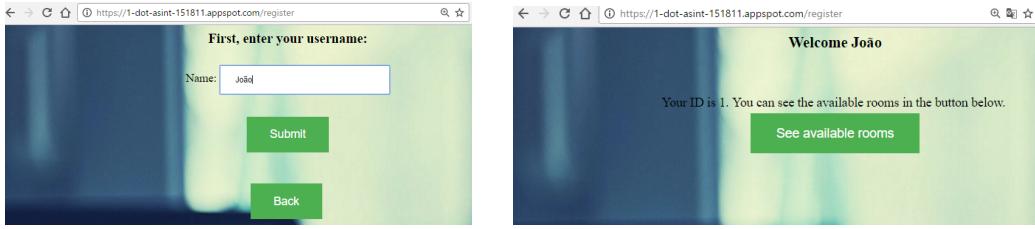
Figura 12: Disponibilização de uma sala

Na lista de salas, tanto para o estudante como para o administrador (figura 13), passa agora a constar a sala E3 adicionada anteriormente.

List of available rooms	
Room Name	Occupancy
E3	0
Back	

Figura 13: Lista de salas e respectiva ocupação [admin].

Quando um estudante utiliza pela primeira vez o sistema, começa por se registar na página da figura 14a. Se o nome já constar na base de dados, é devolvida uma mensagem de erro. Caso contrário, o estudante é informado do seu número de identificação no sistema e é tornado visível um botão que permite ao estudante seguir para a página onde poderá consultar a lista de salas disponíveis.



(a) Página de Inscrição

(b) Resultado de uma inscrição válida

Figura 14: Inscrição de um aluno

Se o administrador, além da E3, adicionar ainda as salas E1 e E2, a página principal do estudante será a representada na figura 15.

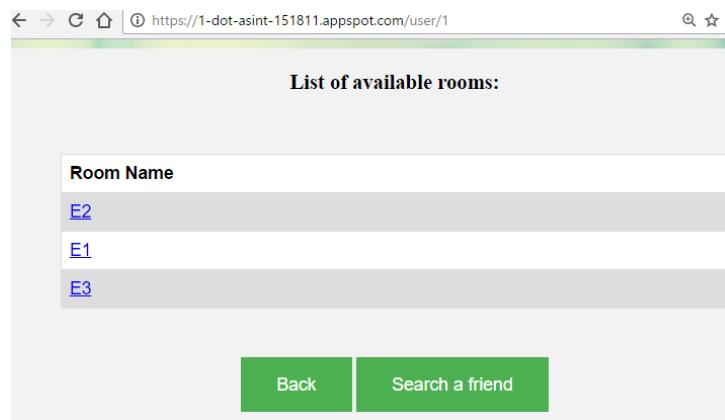


Figura 15: Lista de salas disponíveis [aluno].

Nesta página, se o estudante pretender encontrar um amigo, poderá clicar no botão *Search a friend*, onde na figura 16 se representa a <div> que foi tornada visível através de JavaScript pelo clique no botão.

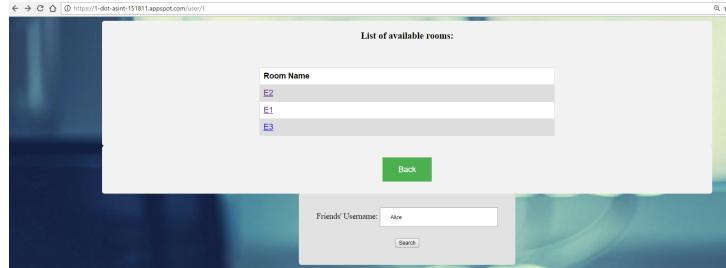


Figura 16: Lista de salas e secção *Search a friend*.

Depois de inserido o nome, o botão de procura desencadeia um pedido HTTP assíncrono através de AJAX, enviando um pedido GET para o URL `https://1-dot-asint-151811.appspot.com/user/<user_id>/listrooms`. O servidor Python, deverá receber o pedido através do Bottle e executar os *queries* na base de dados. Pode retornar as respostas representadas nas figuras 17a, 17b, 17c.

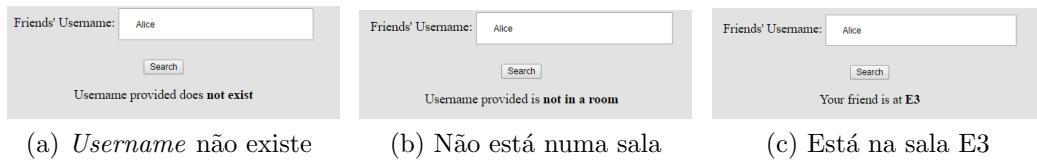


Figura 17: Resultados de *Search a friend*.

Quando o aluno seleciona uma sala na figura 15, por exemplo a sala E3, o resultado é visível na figura 18a. Caso o aluno pretenda entrar na sala E3 e seleccionar o botão *Check In* este aluno será inscrito nessa sala e sendo o resultado apresentado na figura 18b.



Figura 18: Página de uma sala [aluno]

Na figura 19a é apresentada a nova listagem de alunos presentes na sala E3, já após a inscrição do aluno. Caso o aluno pretenda sair da sala deverá clicar no botão *Check Out*. Se o aluno estiver na sala na qual selecionou *Check Out* deverá ver algo semelhante à figura 19b. Quando o aluno tenta fazer *check out* numa sala na qual não está inscrito, essa operação não é realizada.



(a) Lista de alunos numa sala e opções check in/out (b) Resultado de check out na sala E3

Figura 19: Página de uma sala [aluno] - continuaçāo.