Deploy de back-end em Spring utilizando Heroku

O que é o Heroku?

O Heroku é uma plataforma de cloud que oferece “Platform as a Service”, ou seja, ele permite que você hospede suas aplicações em um ambiente facilmente escalável e com suporte a várias tecnologias. Ele tem um plano free, que é indicado para testes, e opções pagas com mais funcionalidades e suporte.

O que veremos por aqui?

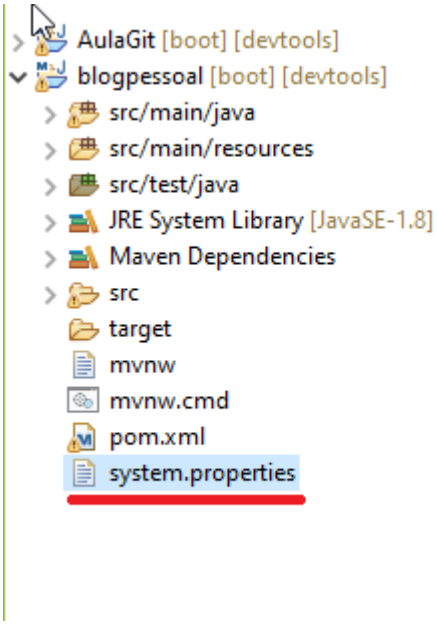
Esse documento é um passo a passo para você subir (deploy) o sua API criada em SPRING gratuitamente para o Heroku, que é uma aplicação de hospedagem de site na web, isso irá gerar um link de acesso a sua página que poderá ser acessadoem qualquer lugar. Para realizar esse deploy vamos precisar fazer algumas modificações em nosso projeto e principalmente já **criar uma conta no Heroku** através desse endereço: https://www.heroku.com.

#01 Passo

Crie a conta no heroku

#02 Passo

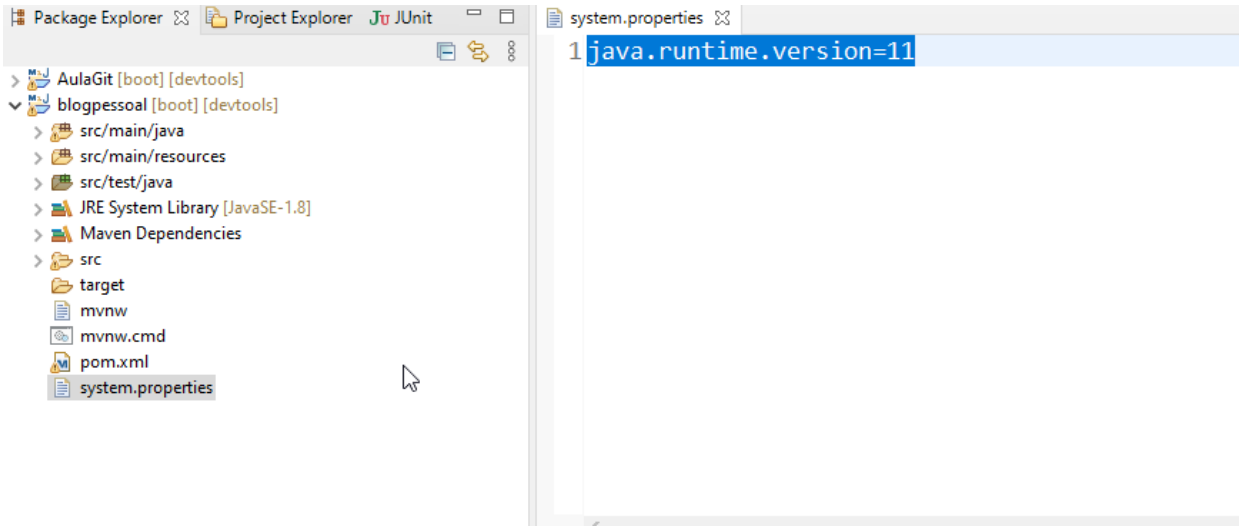
Crie um arquivo `system.properties` na raiz do projeto.



coloque a seguinte informação:

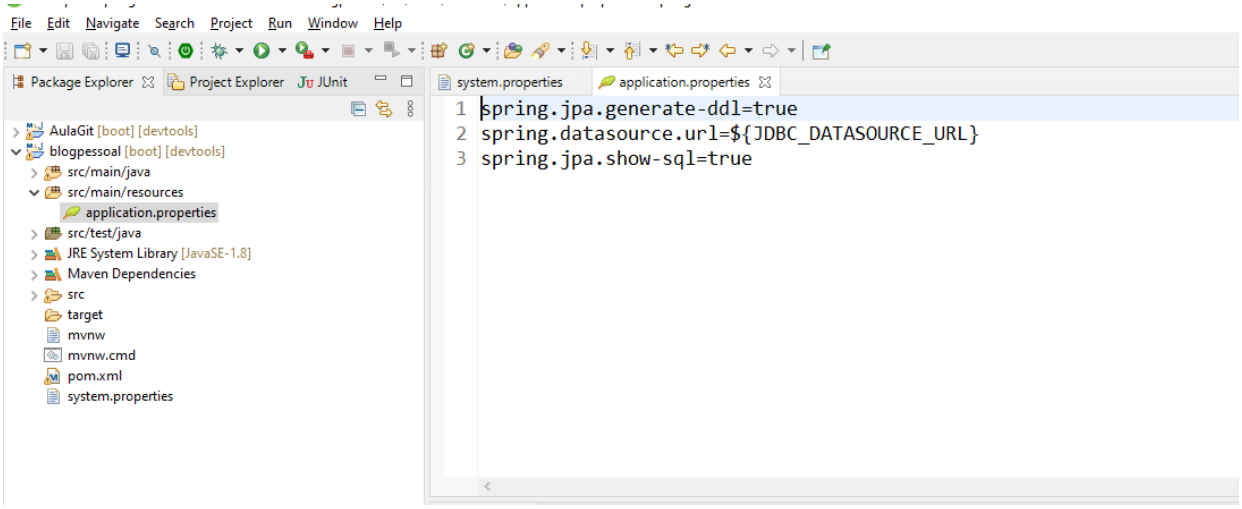
```
java.runtime.version=11
```

indique a versão do java do seu projeto.



#03 Passo

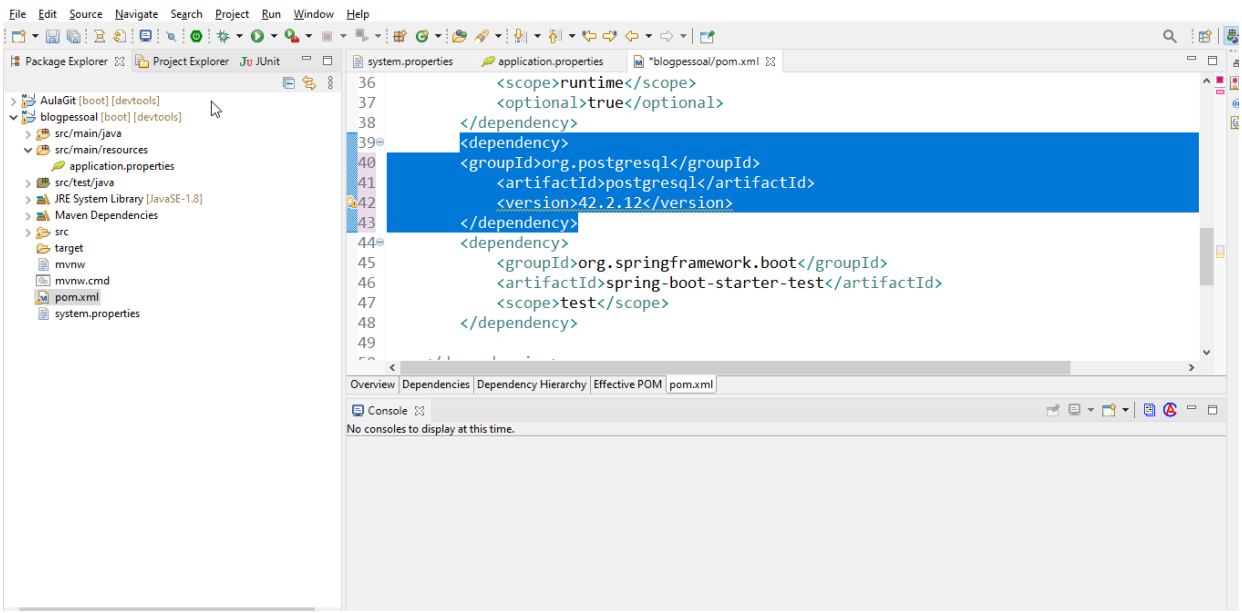
Substitua o conteúdo do arquivo aplication.properties, para:




```
spring.jpa.generate-ddl=true
spring.datasource.url=${JDBC_DATASOURCE_URL}
spring.jpa.show-sql=true
```

#03 Passo

Abra o **pom.xml** e substitua a dependência do MySql por essa dependência:




```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.12</version>
</dependency>
```

#04 Passo

Pronto, agora só precisamos abrir a **pasta que contém o arquivo pom.xml** no terminal de sua escolha e digite os seguintes comandos para criar um repositório no git:

git init
git add .
git commit -m “mensagem”

#05 Passo

Agora precisamos configurar o Heroku no terminal, porém antes de iniciar precisamos instalar o heroku através do pacote npm:

```
npm i -g heroku
```

Agora é só fazer o login no heroku e continuar as configurações:

```
heroku login
```

Após a execução desse comando será aberta em seu navegador uma página da Heroku com um botão para você logar, clique nele, volte para o terminal e prossiga com as configurações.

```
heroku create nomedoprojeto  //mesmo nome usado nos endpoints
```

Esse comando serve para criar o seu projeto na Heroku.

```
heroku addons:create heroku-postgresql:hobby-dev
```

Esse comando serve para criar o banco de dados do seu projeto na Heroku.

#06 Passo

Para finalizar só precisamos digitar o seguinte comando, ainda no terminal:

```
git push heroku master
```

No próprio terminal irá aparecer a url que você precisa entrar para abrir o projeto no navegador, mas normalmente a url é https://nomedoprojeto.heroku.com.

Conceitos gerais

KISS

Qualquer um deve ser capaz de usar sua API sem precisar da documentação.

- Use termos padrão, concretos e comuns, nunca termos ou siglas de negócio específicos da sua empresa.
- Não deve existir mais de uma maneira de se obter um mesmo resultado.
- A API é projetada para seus clientes (desenvolvedores), não para os seus dados.
- Foque nos principais casos de uso, e deixe os casos excepcionais para depois.

```
GET /orders, GET /users, GET /products, etc.
```

CURL

Compartilhe vários exemplos de cURL, que facilitem o uso de copy/paste.

```
CURL –X POST \
-H "Accept: application/json" \
-H "Authorization: Bearer at-80003004-19a8-46a2-908e-33d4057128e7" \
-d '{"state":"running"}' \
https://api.fakecompany.com/v1/users/007/orders?client_id=API_KEY_003
```

Granularidade média de recursos

Use granularidade média, nem fina, nem grossa
Recursos não devem ser aninhados em mais de 2 níveis:

```
GET /users/007
```

```
{ "id":"007",
  "firstName":"James",
  "name":"Bond",
  "address":{
      "street":"H.Ferry Rd.",
      "city":{"name":"London"}
  }
}
```

Considere usar esses cinco subdomínios

- API produção - `https://api.fakecompany.com`
- API teste - `https://api.sandbox.fakecompany.com`
- Developer portal - `https://developers.fakecompany.com`
- OAuth2 produção - `https://oauth2.fakecompany.com`
- OAuth2 teste - `https://oauth2.sandbox.fakecompany.com`

Segurança: OAuth2 & HTTPS

Você deve usar **OAuth2** para gerenciar a autorização

- OAuth2 se encaixa em 99% dos requisitos e topologias dos clientes, não reinvente a roda!

Use **HTTPS** para todas as chamadas da API/OAuth2

URLs

Nomes

Você deve usar nomes (pronomes), não verbos (ao contrário de SOAP-RPC)

```
GET /orders not /getAllOrders
```

Plurais

Use nomes no plural, e não no singular, para gerenciar dois tipos diferentes de recursos:

- Coleções de recursos: `/users`
- Instância de um recurso : `/users/007`

Mantenha a consistência

`GET /users/007` e não `GET /user/007`

Consistência da caixa

Escolha entre snake_case ou camelCase para atributos e parâmetros, mas mantenha a consistência

```
GET /orders?id_user=007
POST/orders {"id_user":"007"}
```

ou

```
GET /orders?idUser=007
POST/orders {"idUser":"007"}
```

Se você tem mais de uma palavra na URL, use spinal-case (alguns servidores ignoram a caixa)

```
POST /specific-orders
```

Versionamento

Use versionamento obrigatório na URL, no escopo mais alto (major versions)
Você deve suportar no máximo duas versões ao mesmo tempo (apps nativas precisam de um longo ciclo)

```
GET /v1/orders
```

Estrutura hierárquica

Leve a hierarquia natural dos recursos para a URL para sugerir a estrutura(agregação ou composição)
Ex.: um pedido contém produtos

```
GET /orders/1234/products/1
```

Operações CRUD: Use verbos HTTP para operações CRUD (**C**reate/**R**ead/**U**pdate/**D**elete).

Verbo HTTP	Coleção: /orders	Instância: /orders/{id}
GET	Lê a lista de pedidos. 200 OK.	Lê os detalhes de um pedido específico. 200 OK.
POST	Cria um novo pedido. 201 Created.	-
PUT	-	Update completo: 200 OK./ Cria um pedido específico: 201 Created.
PATCH	-	Update parcial. 200 OK.
DELETE	-	Deleta um pedido. 204 OK.

POST é usado para **C**reate (criar) uma instância numa coleção. O ID não é fornecido, e o endereço do novo recurso é retornado no header "Location".

```
POST /orders {"state":"running", "id_user":"007"}
201 Created
Location: https://api.fakecompany.com/orders/1234
```

Mas lembre-se que se o ID for fornecido pelo cliente, PUT será usado para **C**reate (criar o recurso).

```
PUT /orders/1234
201 Created
```

PUT é usado para **U**pdates, para realizar uma substituição completa do recurso.

```
PUT /orders/1234 {"state":"paid", "id_user":"007"}
200 Ok
```

PATCH é frequentemente usado para **U**pdates parciais, substituindo alguns atributos.

```
PATCH /orders/1234 {"state":"paid"}
200 Ok
```

GET é usado para **R**ead (ler) a coleção.

```
GET /orders
200 Ok
[{"id":"1234", "state":"paid"}
 {"id":"5678", "state":"running"}]
```

GET é usado para **R**ead (ler) uma instância.

```
GET /orders/1234
200 Ok
{"id":"1234", "state":"paid"}
```

Query strings

Busca

Você pode usar o "Google way" para fazer uma busca global em múltiplos recursos.

```
GET /search?q=running+paid
```

Filtros

Use '?' para filtrar recursos

```
GET /orders?state=payed&id_user=007
```

ou (múltiplas URIs podem referenciar o mesmo recurso)

```
GET /users/007/orders?state=paied
```

Paginação

Você pode usar um parâmetro *range (faixa de valores) na* query. A paginação é obrigatória: uma paginação default tem que ser definida, por exemplo: *range=0-25*.

O response deve conter os headers: Link, Content-Range, Accept-Range.

Note que a paginação pode causar comportamento estranho se muitos recursos forem adicionados.

```
/orders?range=48-55
206 Partial Content
Content-Range: 48-55/971
Accept-Range: order 10
Link : <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first",
<https://api.fakecompany.com/v1/orders?range=40-47>; rel="prev",
<https://api.fakecompany.com/v1/orders?range=56-64>; rel="next",
<https://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

Responses parciais

Use partial responses para que os desenvolvedores possam escolher qual informação eles precisam, para otimizar banda (fundamental para desenvolvimento mobile).

```
GET /users/007?fields=firstname,name,address(street)
200 OK
{ "id":"007",
  "firstname":"James",
  "name":"Bond",
   address:{"street":"Horsen Ferry Road"}
}
```

Ordenação

Use ?sort =atribute1,atributeN para ordenar recursos. A ordenação crescente é default.

Use ?desc=atribute1,atributeN para ordenar em ordem decrescente.

```
GET /restaurants?sort=rating,reviews,name;desc=rate,reviews
```

Palavras reservadas na URL: first, last, count

Use /first para obter o 1º elemento

```
GET /orders/first
200 OK
{"id":"1234", "state":"paid"}
```

Use /last para obter o último recurso de uma coleção

```
GET /orders/last
200 OK
{"id":"5678", "state":"running"}
```

Use /count para obter o tamanho atual da coleção

```
GET /orders/count
200 OK
{"2"}
```

Outros conceitos importantes

Negociação de conteúdo

A negociação de conteúdo é conduzida somente da forma RESTful pura. Os clientes requisitam o tipo do conteúdo, no header Accept, na ordem de preferência. O formato default é JSON.

`Accept: application/json, text/plain` e não `/orders.json`

I18N (internacionalização)

Use o padrão ISO8601 para Date/Time/Timestamp : `1978-05-10T06:06:06+00:00` ou `1978-05-10`

Adicione suporte para outras linguas.

`Accept-Language: fr-CA, fr-FR` e não `?language=fr`

Requisições cross-origin

Use o padrão CORS para suportar requisições de browsers para a API REST (js SPA, etc.).

Mas se você vai suportar Internet Explorer 7/8/9, você deve considerar endpoints específicos para adicionar suporte JSONP.

- Todos os requests serão enviados com método GET!
- A negociação de conteúdo não pode ser feita no header Accept com JSONP.
- O payload não pode ser usado para enviar dados.

```
POST /orders       e também  /orders.jsonp?method=POST&callback=foo
GET /orders        e também  /orders.jsonp?callback=foo
GET /orders/1234   e também  /orders/1234.jsonp?callback=foo
PUT /orders/1234   e também  /orders/1234.jsonp?method=PUT&callback=foo
```

Atenção: um web crawler pode facilmente causar problemas na sua aplicação com um parâmetro *method*. Certifique-se de pedir sempre um *access_token* OAuth2, e também o *client_id* OAuth2.

HATEOAS

Sua API deve oferecer links Hypermedia para ser totalmente *discoverable*. Mas tenha em mente que a maioria dos usuários não vão usar esses hyperlinks por enquanto, e vão precisar da documentação da API e de fazer copy/paste com os exemplos de chamadas.

Então, cada chamada para a API deve retornar no header *Link* todos os possíveis estados da aplicação a partir do estado atual, além dele mesmo.

Você pode usar a notação de Link da RFC5988 para implementar o HATEOAS:

```
GET /users/007
< 200 Ok
< { "id":"007", "firstname":"James",...}
< Link : <https://api.fakecompany.com/v1/users>; rel="self"; method:"GET",
<https://api.fakecompany.com/v1/addresses/42>; rel="addresses"; method:"GET",
<https://api.fakecompany.com/v1/orders/1234>; rel="orders"; method:"GET"
```

Cenários "Sem recursos"

Em alguns poucos casos precisamos considerar operações ou serviços ao invés de recursos.

Você pode usar um request POST com um verbo no final da URI

```
POST /emails/42/send
POST /calculator/sum  [1,2,3,5,8,13,21]
POST /convert?from=EUR&to=USD&amount=42
```

Mas você deve considerar usar recursos RESTful antes de partir para uma solução com verbos.

Status codes HTTP

Você deve usar status codes HTTP adequados.
Você pode usar a notação padrão OAuth2: http://tools.ietf.org/html/rfc6749#page-45
Mantenha a simplicidade: não tente usar todos os status codes HTTP, mas apenas os 12 principais.

Verbo HTTP	Status code HTTP	Descrição
SUCCESS	200 OK.	Código de sucesso básico. Funciona para na maioria dos casos. Especialmente usado no sucesso do primeiro request GET, ou update com PUT/PATCH.
	201 Created.	Indica que o recurso foi criado. Resposta típica a um request PUT ou POST.
	202 Accepted.	Indica que o request foi aceito para processamento. Resposta típica a uma chamada para processamento assíncrono (para melhor UX e boa performance).
	204 No Content.	O request funcionou, mas não há conteúdo a retornar. Resposta comum para um DELETE com sucesso.
	206 Partial Content.	O recurso retornado está incompleto. Usado normalmente em recursos paginados.
CLIENT ERROR	400 Bad request.	Erro geral para qualquer request (caso não se encaixe nos demais). A boa prática é ter dois tipos de erros: erro no request, e condição de erro na aplicação. Exemplo de erro no request: GET /users?payed=1 < 400 Bad Request < {"error": "invalid_request", "error_description": "There is no 'payed' property on users."} Exemplo de condição de erro na aplicação: POST /users {"name": "John Doe"} < 400 Bad Request < {"error": "invalid_user", "error_description": "A user must have an email adress"}
	401 Unauthorized.	Eu não conheço o seu id. Diga-me quem você é, e eu vejo sua autorização. GET /users/42/orders < 401 Unauthorized < {"error": "no_credentials", "error_description": "This resource is under permission, you must be authenticated with the right rights to have access to it"}
	403 Forbidden.	Você foi autenticado corretamente, mas não tem privilégios suficientes. GET /users/42/orders < 403 Forbidden < {"error": "not_allowed", "error_description": "You're not allowed to perform this request"}
	404 Not Found.	O recurso que você pediu não existe. GET /users/999999 < 400 Not Found < {"error": "not_found", "error_description": "The user with the id '999999' doesn't exist"}
	405 Method not allowed.	Você chamou um método que não faz sentido nesse recurso, ou o usuário não tem permissão de fazer essa chamada. POST /users/8000 < 405 Method Not Allowed < {"error":"method_does_not_make_sense", "error_description":"How would you even post a person?"}
	406 Not Acceptable.	Nenhum formato se encaixa no Header Accept-* do seu request. Por exemplo, você pediu o recurso em formato XML, mas ele só está disponível em JSON. Isso também funciona para I18N (internacionalização). GET /users Accept: text/xml Accept-Language: fr-fr < 406 Not Acceptable < Content-Type: application/json < {"error": "not_acceptable", "available_languages":["us-en", "de", "kr-ko"]}
SERVER ERROR	500 Internal server Error.	A requisição está correta, mas ocorreu um erro de execução. O cliente não tem muito o que fazer sobre isso, então apenas retorne um status 500. GET /users < 500 Internal server error < Content-Type: application/json < {"error":"server_error", "error_description":"Oops! Something went wrong... "}

AVISO

Esse Guia de Referência não pretende ser totalmente preciso. Os conceitos de projeto aqui expostos são resultado de projetos anteriores com REST. Leia nosso blog http://blog.octo.com/pt-br, e sinta-se à vontade para comentar/desafiar esse guia de APIs. Nós esperamos poder compartilhar mais experiências com você.

Mais uma coisa

Nós o encorajamos a ser pragmático, para o benefício dos seus clientes: ***Desenvolvedores de aplicativos***.

Fontes

Design Beautiful REST + JSON APIs
> http://www.slideshare.net/stormpath/rest-jsonapis

Web API Design: Crafting Interfaces that Developers Love
> https://pages.apigee.com/web-api-design-website-h-ebook-registration.html

HTTP API Design Guide
> https://github.com/interagent/http-api-design

RESTful Web APIs
> http://shop.oreilly.com/product/0636920028468.do

OCTO Technology

ACREDITAMOS QUE A TECNOLOGIA *da informação*
TRANSFORMA NOSSA SOCIEDADE
SABEMOS QUE AS *grandes realizações*
SÃO FRUTOS DO *compartilhamento de conhecimento*
E DO PRAZER DE +RABALHAR EM EQUIPE
NÓS *buscamos* SEMPRE AS
MELHORES *formas* DE TRABALHAR