FIAP GRADUAÇÃO

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Arquiteturas Disruptivas, IoT, IA e Big Data

PROF. ANTONIO SELVATICI

Versão 5– Março/2019

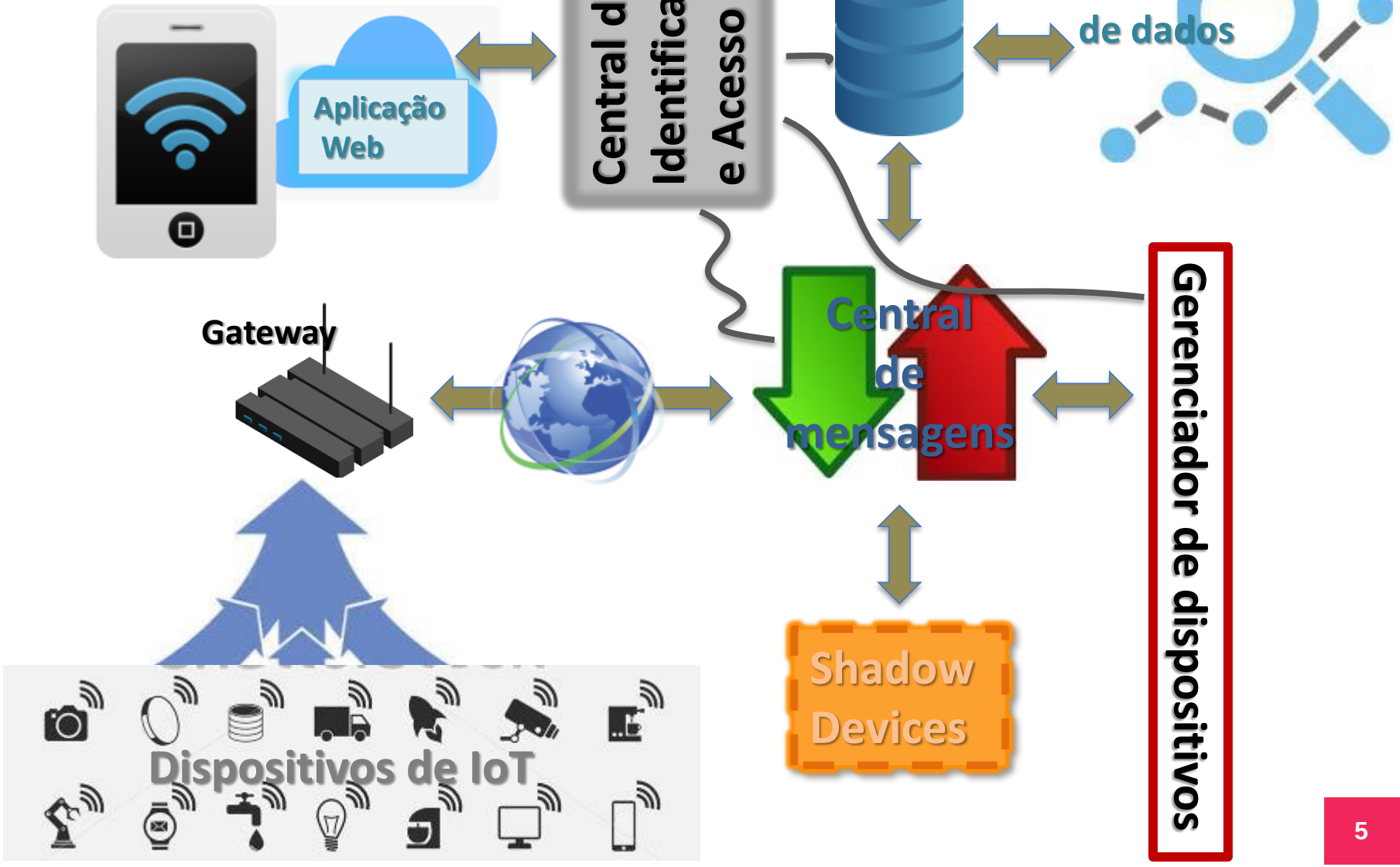
SHORT BIO

É engenheiro eletrônico formado pelo Instituto Tecnológico de Aeronáutica (ITA), com mestrado e doutorado pela Escola Politécnica (USP), e passagem pela Georgia Institute of Technology em Atlanta (EUA). Desde 2002, atua na indústria em projetos nas áreas de robótica, visão computacional e internet das coisas, aliando teoria e prática no desenvolvimento de soluções baseadas em Machine Learning, processamento paralelo e modelos probabilísticos. Desenvolveu projetos para Avibrás, IPT, CESP e Systax.

PROF. ANTONIO SELVATICI
profantonio.selvatici@fiap.com.br

INTERNET DAS COISAS

Arquitetura proposta



Aplicações de IoT

- As aplicações de IoT são aplicações com a capacidade de interagir com dispositivos e gateways através das plataformas de IoT
  - Comumente o protocolo usado para comunicar com o dispositivo através das plataforma é o MQTT
- A construção dessas aplicações em geral obedece à arquitetura multi camadas, incluindo
  - Camada de negócio
  - Camada de acesso aos dados
  - Camada de interface com aplicações cliente (API)
- Enquanto a construção de aplicações web é objeto de outras disciplinas, aqui vamos complementar com a construção de APIs RESTful em Java e Node-RED

API RESTful com Jersey

ReST – Representational State Transfer

- Do Wikipedia:
  - A **Representational State Transfer** (REST), em português Transferência de Estado Representacional, é um estilo de arquitetura que define um conjunto de restrições e propriedades baseados em HTTP.
- Os web services compatíveis com REST permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web usando um conjunto uniforme e predefinido de operações sem estado
  - As operações mais comuns correspondem aos métodos GET, POST, PUT e DELETE
  - Os formatos de dados mais usados para a representação de recursos são JSON, XML e HTML
  - Em IoT, um recurso pode se referir a um dispositivo ou a uma funcionalidade desse dispositivo

Acesso aos recursos

- Uma API que obedeça aos princípios do REST é chamada API RESTful, e deve ter as seguintes características:
  - Client/Server
  - Stateless: não há variáveis de estado que afetem as respostas à chamada
  - Possibilidade de cache de recursos
  - Representação uniforme, padronizada, de recursos
- Os recursos são em geral expressos usando URIs (*Uniform Resource Identifiers*) padronizadas. Por exemplo:
  - http://myapi.com/store/costumer/id/53 (identifica o cliente com ID 53)
  - https://my_iot_api.com/deviceType/zigbee/deviceId/754/led (identifica o LED do device tipo “zigbee” com ID 754

Programando a Aplicação Web – RESTful API

- Numa arquitetura simplificada de IoT, uma única aplicação Web se encarrega de gerenciar e controlar o acesso aos dispositivos e comandar o armazenamento em banco de dados, além de fornece uma API, geralmente RESTful, para que programas aplicativos possam ter acesso às regras de negócio
- Essa aplicação precisa ter três pontas:
  - A ponta do cliente do banco de dados
  - A ponta do cliente MQTT
  - A ponta do servidor REST
- Via de regra, essa aplicação web é desenvolvida usando linguagens de programação flexíveis, como Java, Ruby, ou, como tem se tornado preferência no mundo de IoT, Node.js
- No entanto, para aplicações simples, podemos usar também o próprio Node-RED para programá-la

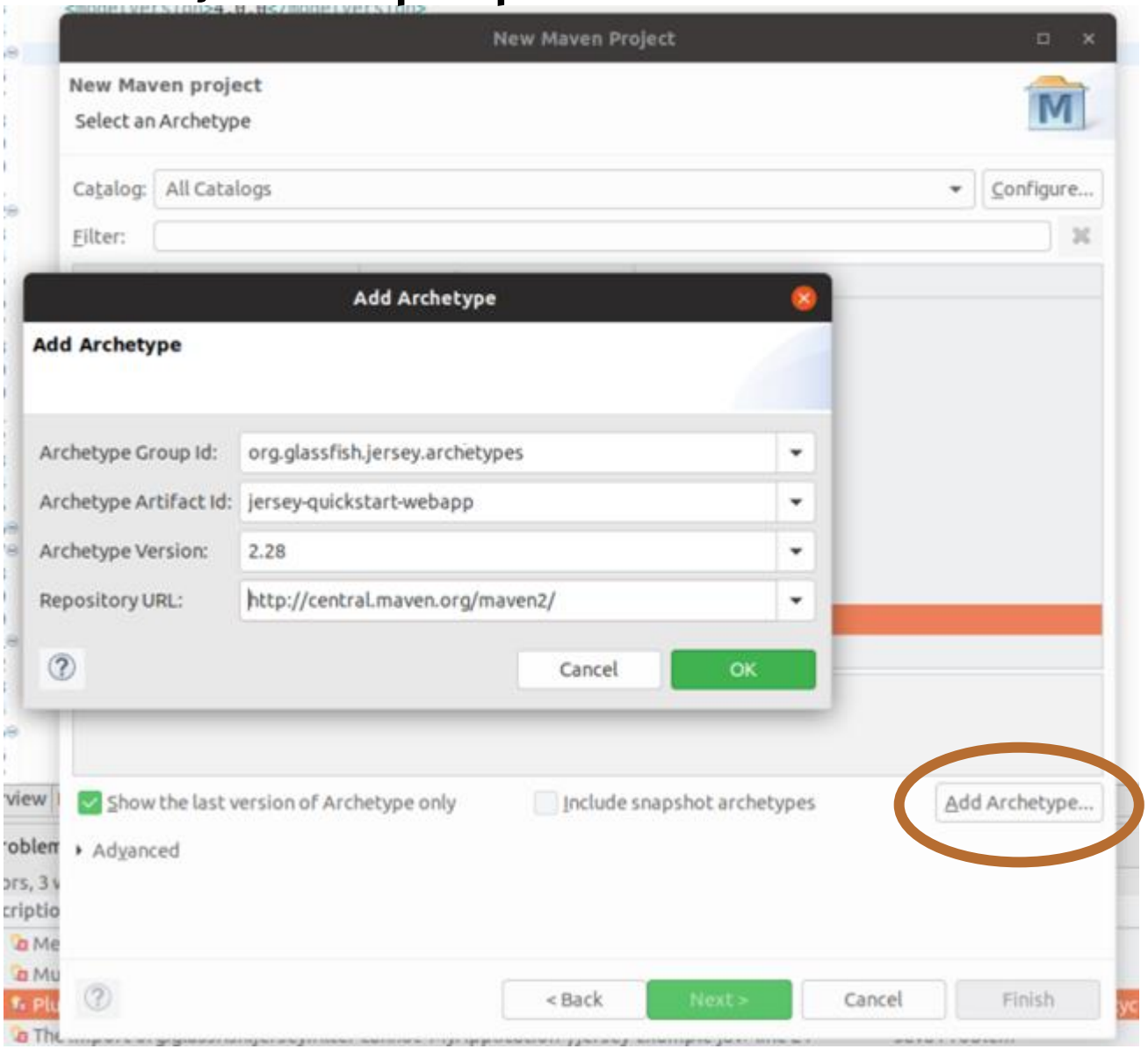
API RESTful com Jersey

- O Java EE (Enterprise Edition) é composto por uma série de especificações (Java Specification Requests – JSR) que definem as diversas funcionalidades padronizadas previstas, como JPA, JSF, etc.
- Uma dessas funcionalidades é a construção de uma API RESTful entre aplicações, que está definida dentro do padrão **JAX-RS** (https://projects.eclipse.org/projects/ee4j.jaxrs)
  - Enquanto a JAX-RS é apenas uma especificação da API que deve implementar as funções requisitadas, precisamos do software que de fato implementa esses requisitos.
- Jersey (https://jersey.github.io/) é o projeto que consiste na implementação padrão da JAX-RS, e que facilita a criação de APIs RESTful por meio de simples anotações (versão 2.1 da JAX-RS).

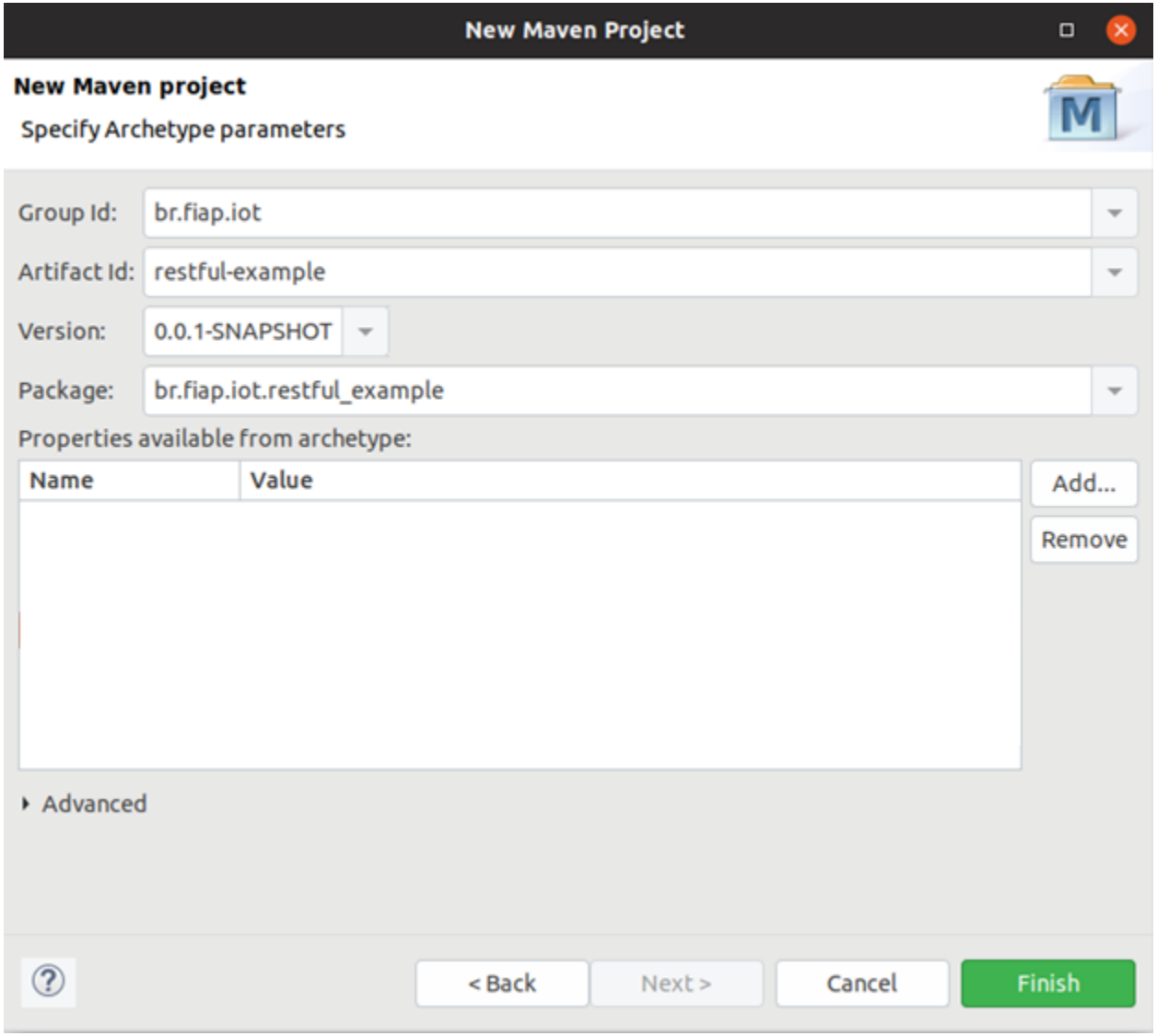
Preparação do projeto no Maven

- Enquanto um servidor de aplicações mais robusto como Glassfish e JBoss geralmente já possui sua implementação da JAX-RS, vamos considerar a implantação no servidor Tomcat, que vem “pelado”.
- Para criar um novo projeto Maven no Eclipse, podemos usar um arquétipo (*arquetype*) que já traga a configuração básica necessária para o projeto de API usando o Jersey.
  - Arquivo -> Novo -> Projeto...
  - Dentro da pasta “Maven”, escolher “Projeto Maven”
  - Usar a localização padrão, e clicar em “Next” para escolher o arquétipo

Janelas de seleção de arquétipo



Configuração do projeto

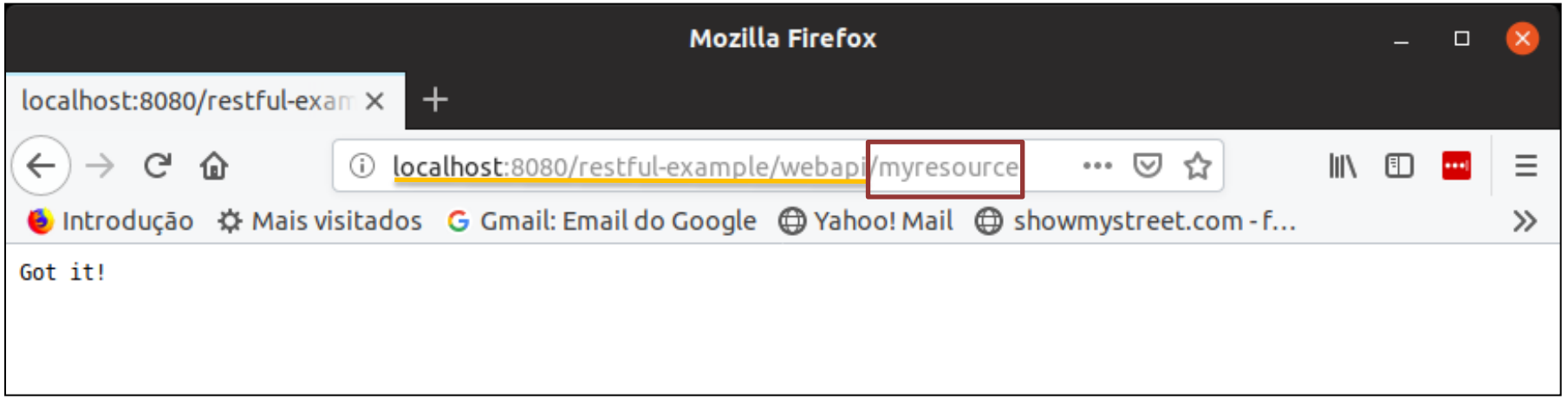


Meu primeiro projeto Jersey

restful-example/pom.xml | MyResource.java

```
1  package br.fiap.iot.restful_example;
2
3  import javax.ws.rs.GET;
7
8  /**
9   * Root resource (exposed at "myresource" path)
10  */
11 @Path("myresource")
12 public class MyResource {
13
14     /**
15      * Method handling HTTP GET requests. The returned object will be sent
16      * to the client as "text/plain" media type.
17      *
18      * @return String that will be returned as a text/plain response.
19      */
20     @GET
21     @Produces(MediaType.TEXT_PLAIN)
22     public String getIt() {
23         return "Got it!";
24     }
25 }
26
```

Acesso ao webservice



Meu primeiro projeto Jersey

- O arquétipo dá conta de importar os pacotes e configurações necessários
- O webservice estará disponível em um arquivo WAR, com toda a estrutura de diretórios necessária pra exportar um Servlet rodando no container do Tomcat
- Anotações importantes:
  - **@Path**: anota a classe, indicando qual é o caminho do recurso dentro do servidor
  - **@GET, @POST, @PUT, @DELETE**, etc.: anota o método da classe, indicando qual é o comando HTTP que aquele método irá tratar
  - **@Produces**: indica o tipo de resposta que será produzido pelo método
    - Por padrão, não há suporte à produção de JSON
    - Para ativar esse suporte, devemos usar o POM para habilitar a dependência `org.glassfish.jersey.media:jersey-media-json-binding`
  - **@Consumes**: indica os tipos de mídia que podem ser consumidos. Em caso de omissão, todos os tipos de mídia são aceitos

restful-example.pom

restful-example/pom.xml | MyResource.java

```
36              </dependency>
37          </dependencies>
38      </dependencyManagement>
39
40      <dependencies>
41          <dependency>
42              <groupId>org.glassfish.jersey.containers</groupId>
43              <artifactId>jersey-container-servlet-core</artifactId>
44              <!-- use the following artifactId if you don't need servlet 2.x compatibility -->
45              <!-- artifactId>jersey-container-servlet</artifactId -->
46          </dependency>
47          <dependency>
48              <groupId>org.glassfish.jersey.inject</groupId>
49              <artifactId>jersey-hk2</artifactId>
50          </dependency>
51          <!-- uncomment this to get JSON support
52          <dependency>
53              <groupId>org.glassfish.jersey.media</groupId>
54              <artifactId>jersey-media-json-binding</artifactId>
55          </dependency>
56          -->
57      </dependencies>
58      <properties>
59          <jersey.version>2.28</jersey.version>
60          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
61      </properties>
62  </project>
```

Criando o recurso MyJsonResource

restful-example/pom.xml | MyResource.java | MyJsonResource.java

```
1  package br.fiap.iot.restful_example;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5  import javax.ws.rs.Produces;
6  import javax.ws.rs.core.MediaType;
7
8  @Path("myjsonresource")
9  public class MyJsonResource {
10
11     /**
12      * Method handling HTTP GET requests. The returned object will be sent
13      * to the client as "application/json" media type.
14      *
15      * @return String that will be returned as a application/json response.
16      */
17     @GET
18     @Produces(MediaType.APPLICATION_JSON)
19     public String getIt() {
20         return "{\"say\": \"Hi!\"}";
21     }
22 }
23
```

Qual é a vantagem????

- Ter que devolver um JSON artesanal não traz muita vantagem sobre devolver um texto simples
- Melhor seria se esse texto JSON fosse criado a partir da serialização de um objeto de recurso preexistente, por exemplo:
  - `webapi/cliente/id/53`: devolveria a informação cadastrada sobre o cliente de ID 53
- Problemas:
  - Geração automática do JSON: resolvido pelo próprio Jersey
  - Tratamento da URI com parâmetro: Jersey permite que parâmetros de caminho sejam identificados e tratados dentro dos métodos através da anotação **@PathParam**

Classe Cliente.java

```
package br.fiap.iot.restful_example;
import java.io.Serializable;
import java.util.Date;

public class Cliente implements Serializable {

     private static final long serialVersionUID =
     7290176776484525708L;

     private long id;
     private Date nascimento;
     private String nome;
     private int cpf;

     public long getId() { return id; }
     public void setId(long id) { this.id = id; }
     public Date getNascimento() { return nascimento; }
     public void setNascimento(Date nascimento) {
          this.nascimento = nascimento;
     }
     public int getCpf() { return cpf; }
     public void setCpf(int cpf) { this.cpf = cpf; }
     public String getNome() { return nome; }
     public void setNome(String nome) { this.nome = nome; }
}
```

restful-example/pom.xml | MyResource.java | MyJsonResource.java | Cliente.java | *MyClienteResource.java

```java
package br.fiap.iot.restful_example;
import java.util.ArrayList;
@Path("cliente")
public class MyClienteResource {
    @GET @Path("{id: [1-9][0-9]*}")
    @Produces(MediaType.APPLICATION_JSON)
    public Cliente getCliente(@PathParam("id") long id) {
        //A classe Cliente poderia se tratar de uma entidade JPA recuperada do BD.
        Cliente cliente = new Cliente();
        cliente.setId(id); //O id é o mesmo procurado
        cliente.setNascimento(new Date()); //Data de agora
        cliente.setNome("Nome Aleatório");
        ////////////////////////////////////////////////////////////////////
        return cliente;
    }
    @GET @Path("all")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Cliente> getClientesList() {
        //Poderíamos estar recuperando a lista de clientes a partir do banco,
        // mas aqui a lista retorna vazia
        return new ArrayList<Cliente>();
    }
    @POST @Consumes(MediaType.APPLICATION_JSON)
    public Cliente cadastro(Cliente cliente) {
        //Salva os dados do cliente em algum lugar
        // e era o novo ID
        cliente.setId(1000);
        return cliente;
    }
}
```

Podemos testar enviando o JSON:
(atenção para o formato data/hora)

```
{
  "nome":"José Joaquim",
  "cpf": 1234,
  "nascimento":
    "1889-01-25T00:00:00.000Z"
}
```

Observações importantes

- O caminho `cliente` se trata do caminho raiz (***root path***) do recurso
- As anotações **`@Path`** internas ao recurso são relativas ao root path.
- Caminhos anotados com `{ }` são tratados como parâmetros de caminho, e podem ser recuperados no método através da anotação **`@PathParam`**
  - O uso da expressão regular no parâmetro de caminho é opcional, e serve para filtrar as URI que eventualmente sejam
- Dois métodos tratando o mesmo comando HTTP (por exemplo, GET) não podem ser ambíguos em termos de seus caminhos.
  - Os caminhos `cliente/{id}` e `cliente/all` são ambíguos, pois poderíamos ter `id <- "all"`
  - Já os caminhos `cliente/{id: [1-9][0-9]*}` e `cliente/all` não são ambíguos, pois `all` não casa com a expressão regular `"[1-9][0-9]*"`, que só admite números inteiros em base decimal não iniciados por zero.

Usando o MQTT em Java

- Para que nossa aplicação Web de IoT faça sentido, ela precisa poder receber e enviar informações aos dispositivos através da Central de Mensagens (MQTT Broker)
  - Para tanto, precisamos empregar um cliente MQTT!
- Aqui vamos usar o cliente do projeto Paho
  - https://www.eclipse.org/paho/
- O primeiro passo é incluir a dependência no arquivo POM:

```
<dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
    <version>1.2.1</version>
</dependency>
```

Conectando o cliente MQTT ao Broker

- 1. Criar o cliente, com client ID aleatório
  - String url = "tcp://iot.eclipse.org:1883“;
  - String clientId = UUID.randomUUID().toString();
  - MqttPersistence persist = new MemoryPersistence();
  - IMqttClient mqttClient = new MqttClient(url, clientId, persist);
- 2. Fazer a conexão, passando parâmetros de conexão
  - MqttConnectOptions options = new MqttConnectOptions();
  - options.setAutomaticReconnect(true);
  - options.setCleanSession(true);
  - options.setConnectionTimeout(10);
  - mqttClient.connect(options);

Publicando em um tópico

- A mensagem deve estar encapsulada em uma `MqttMessage`, que recebe um payload do tipo `byte[]`
  - Se a mensagem for uma String, podemos usar o método getBytes() para transformá-la em bytes de texto Unicode

```
if(mqttClient.isConnected()) {
    //payload é um byte array
    MqttMessage msg = new MqttMessage(payload);
    msg.setQos(0);
    msg.setRetained(true);
    mqttClient.publish(topic, msg);
}
```

Subscrevendo a um tópico

- Para subscrever a um tópico, precisamos cadastrar um “listener” que dispara sempre que chegar uma mensagem àquele tópico
- Se a mensagem for um texto, podemos recuperá-lo usando `toString()`

```
if(mqttClient.isConnected()) {
    mqttClient.subscribe(topic, new IMqttMessageListener{
        @Override
        public void messageArrived(String topic,
MqttMessage msg) throws Exception {
            //A mensagem está em msg
            System.out.println(msg.toString());
        }
    });
}
```

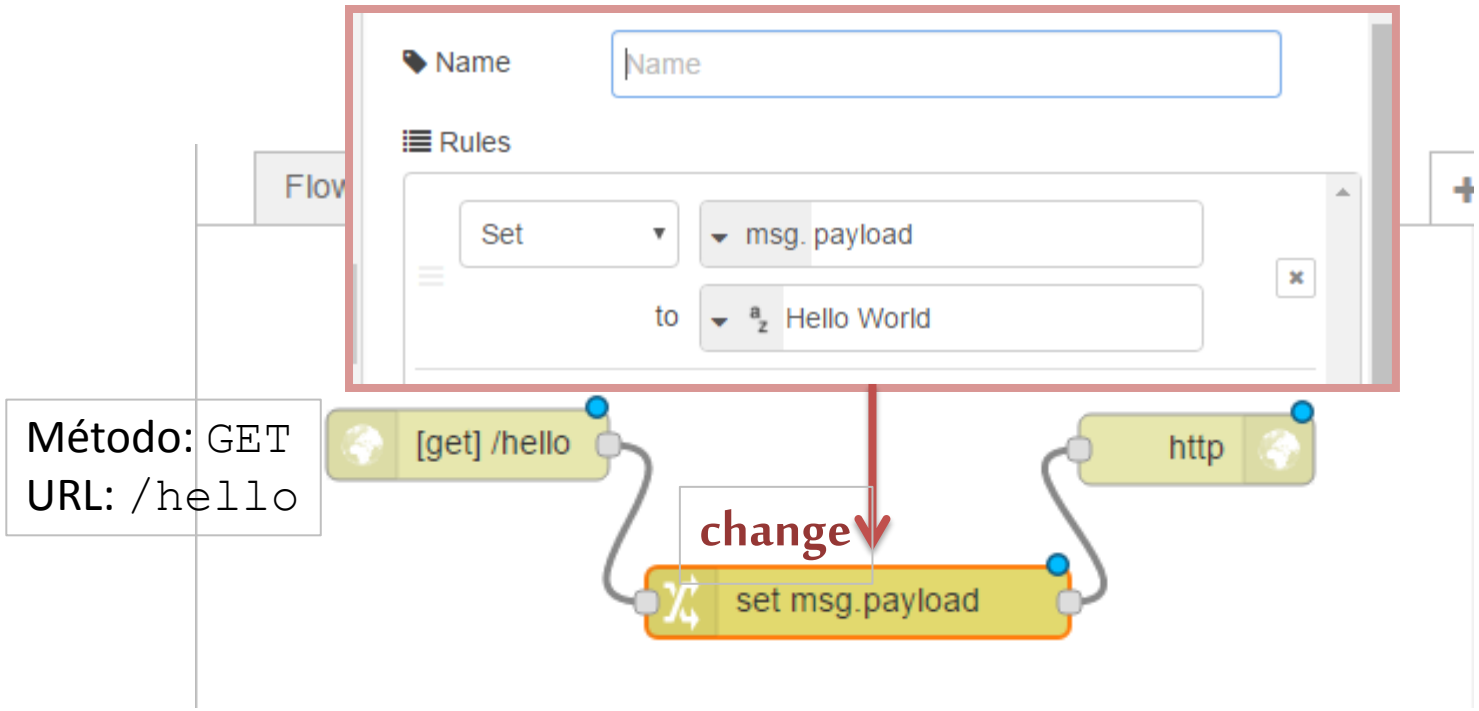
API RESTful com Node-RED

Servidor Web Node-RED

- O próprio Node-RED é um servidor web, que escuta em geral na porta 1880
- Podemos aproveitar esse mesmo servidor e criar URLs adicionais customizadas, definindo ainda o comando HTTP a ser executado (GET, POST, PUT, ou DELETE)
- Para criar um servidor simples, usamos o node “HTTP in” como fonte de dados, e finalizar o fluxo em um node “HTTP response”
- O corpo da resposta é definido pelo campo **msg.payload**, da mensagem **msg** recebida pelo node de saída HTTP, como exemplificado no próximo slide

Exemplo de servidor simples

- Servidor escutando em http://centralhub.mybluemix.net/hello
- O node "**change**" define o valor do campo de uma variável usando alguma regra



Servidor REST com JSON

- No caso de um servidor REST baseado em JSON, as respostas às requisições não são trechos textuais simples, mas possuem uma formatação específica
  - O formato da resposta deve ser especificado através do cabeçalho “Content-Type” da resposta HTTP
- Para ter acesso ao parâmetros de requisição e resposta, basta acessar, respectivamente, os campos `msg.req` e `msg.res` da mensagem resultante do node “HTTP in”
- Para indicar a resposta JSON e liberar as requisições *cross-site* (CORS) da nossa API, usamos o node “change” :
  - **Set:** `msg.headers`
  - **To** (JSON): `{"Content-Type":"application/json", "Access-Control-Allow-Origin":"*"}`

Objetos de requisição e resposta

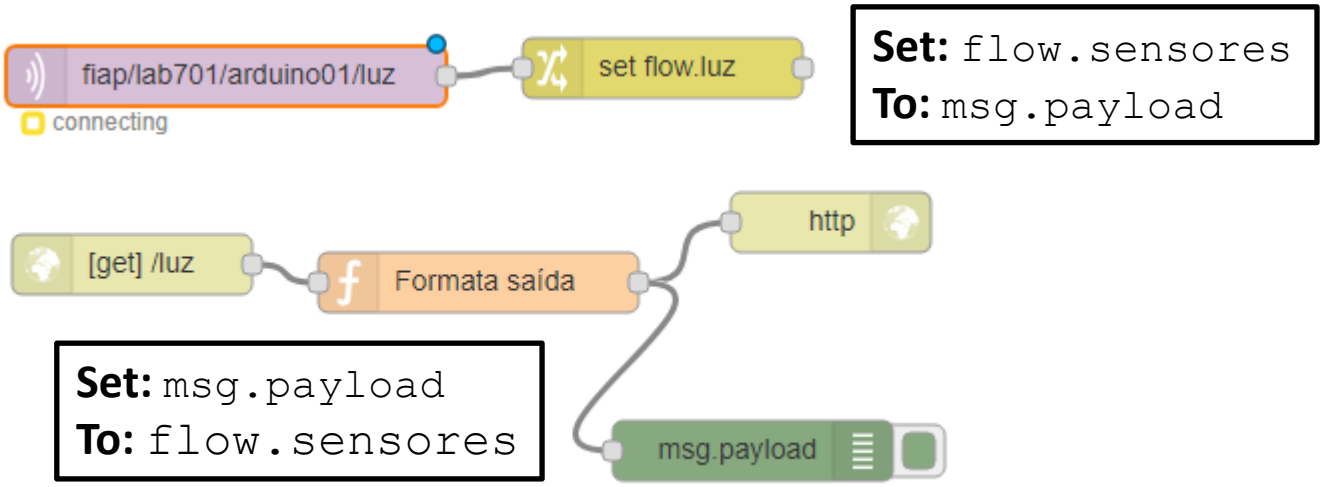
- Uma vez que Node-RED usa o pacote `express` do Node.js como servidor web, os campos `msg.req` e `msg.res` são objetos de requisição e resposta e respeitam a estrutura das classes `HTTPRequest` e `HTTPResponse` respectivamente.
- Os campos relevantes dos objetos de requisição e resposta encontram-se na documentação do próprio express . Alguns deles são:
  - **msg.req.path**: caminho do recurso requisitado no servidor
  - **msg.req.ip**: IP do cliente que realizou a requisição
  - **msg.req.body**: corpo da requisição (geralmente um JSON ou formulário URL-encoded)
  - **msg.req.headers**: cabeçalho da requisição HTTP
- Para definir os principais parâmetros da resposta, preenchemos diretamente os campos em msg:
  - **msg.payload**: corpo da resposta HTTP
  - **msg.headers**: cabeçalho da resposta HTTP
  - **msg.statusCode**: código de resposta HTTP

Simples servidor para informar o último valor lido do sensor

- É necessário tratar dois eventos que estão fora de sincronia: a chegada de dados do Arduino via tópico MQTT e chegada de requisição HTTP do cliente.
- Como sincronizar esses eventos?
  - Armazenar o dado recebido do MQTT em uma variável, e enviar o valor dessa variável quando da requisição HTTP
- Como trabalhar com variáveis no Node-RED?
  - Um node pode armazenar e recuperar informações através de *contexts*, que funcionam como dicionários contendo valores de propriedades
- Há três níveis de *contexts* que podem ser usados no Node-RED:
  - **Local**: pode ser acessado dentro do próprio node
  - **Flow:** é compartilhado por todos os nodes da mesma aba de edição
  - **Global:** é compartilhado por todos os nodes do servidor

Disponibilizando a luminosidade através da URL /luz

- Cada vez que a uma mensagem é recebida do MQTT, ela é armazenada dentro do *context* **flow** na propriedade "luz"
- Cada vez que é feita uma requisição HTTP GET na URL `/luz`, é retornado um JSON com o valor da luminosidade
- Para escrever ou ler o valor de uma propriedade dentro de um context, em uma function, usamos a notação `get/set`



REFERÊNCIAS

1. Wikipedia: REST. https://pt.wikipedia.org/wiki/REST
2. https://www.restapitutorial.com/
3. https://restfulapi.net/
4. https://www.devmedia.com.br/java-restful-como-criar-uma-aplicacao-com-jersey-e-mysql/31681
5. https://jersey.github.io/documentation/latest/user-guide.html
6. https://www.baeldung.com/java-mqtt-client
7. IBM Emerging Technologies. **Node-Red.** url: http://nodered.org
8. IBM IoT Platform documentation. url: https://console.bluemix.net/docs/services/IoT

FIAP