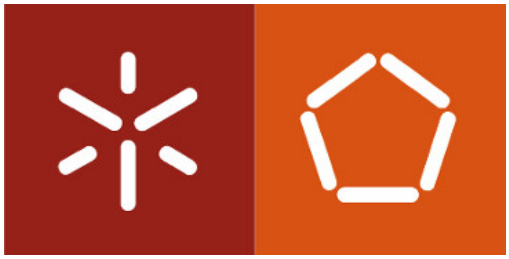

Universidade do Minho

Mestrado integrado em Engenharia Informática

F02 - Interoperabilidade com REST e SOAP

Grupo 5

Vasco António Lopes Ramos, PG42852

Interoperabilidade Semântica

4º Ano, 2º Semestre

Departamento de Informática

29 de abril de 2021

Índice

Lista de Figuras

Lista de Códigos

1 Introdução

1.1 Contextualização

No âmbito da UC de *Interoperabilidade Semântica* foi-nos proposta a realização de uma segunda ficha de exercícios para explorar a aplicação de interoperabilidade semântica através de serviços web REST com JSON para a estrutura de dados de comunicação. De forma mais explícita, o objetivo deste trabalho era desenvolver um ambiente com 2 sistemas completamente separados que comunicam as suas ações entre si através de serviços web: REST e SOAP.

Este trabalho foi desenvolvido pelos dois elementos do grupo (Grupo 05), ao longo de 2 semanas, em que na primeira semana se começou por desenvolver as APIs e serviços de comunicação; na segunda semana procedeu-se ao desenvolvimento dos clientes web (interfaces) de modo a permitir interação com os serviços, bem como a escrita do presente relatório.

Ao nível da divisão de trabalho: eu, Vasco Ramos, fiquei responsável pelo desenvolvimento de tudo o que estava relacionado com o serviço 1 e a minha colega, Carolina Marques, ficou responsável pelo serviço 2.

1.2 Estrutura do relatório

Este relatório encontra-se dividido em 5 partes:

No primeiro capítulo é feita uma breve introdução a este trabalho, e é explicada a estrutura deste relatório.

No segundo capítulo é feita uma análise sobre a arquitetura geral do sistema implementado, bem como as tecnologias usadas para as diferentes camadas.

No terceiro capítulo são abordados tópicos relevantes sobre decisões e estruturas de implementação relativas às principais camadas: base de dados, serviços e respetivos clientes, bem como o que fazer para migrar de REST para SOAP.

No quarto capítulo procede-se à exposição e demonstração dos resultados obtidos, bem como uma discussão dos mesmos.

Por fim, no último capítulo é feito um breve resumo do que foi possível implementar e realizar ao longo do trabalho respetivo a este relatório, bem como uma breve análise conclusiva sobre o que se retirou do mesmo.

2 Arquitetura

A arquitetura foi feita de forma a que cada serviço realiza as suas operações internamente, através do seu cliente *WEB*, comunicando com a sua base de dados. A garantia de sincronização de dados é feita através da comunicação com a *REST API* do outro serviço que permite a realização de todas as ações necessárias para tal.

Deste modo, cada serviço é constituído pela plataforma de comunicação com a Base de Dados e um servidor aplicacional em **Node.js**, que disponibiliza o cliente *WEB* para interação "local"com o serviço e a *REST API* para permitir ao outro serviço não só consultar informação com sincronizar e atualizar informação.

A figura 1 apresenta uma visão genérica de como os vários componentes se relacionam.

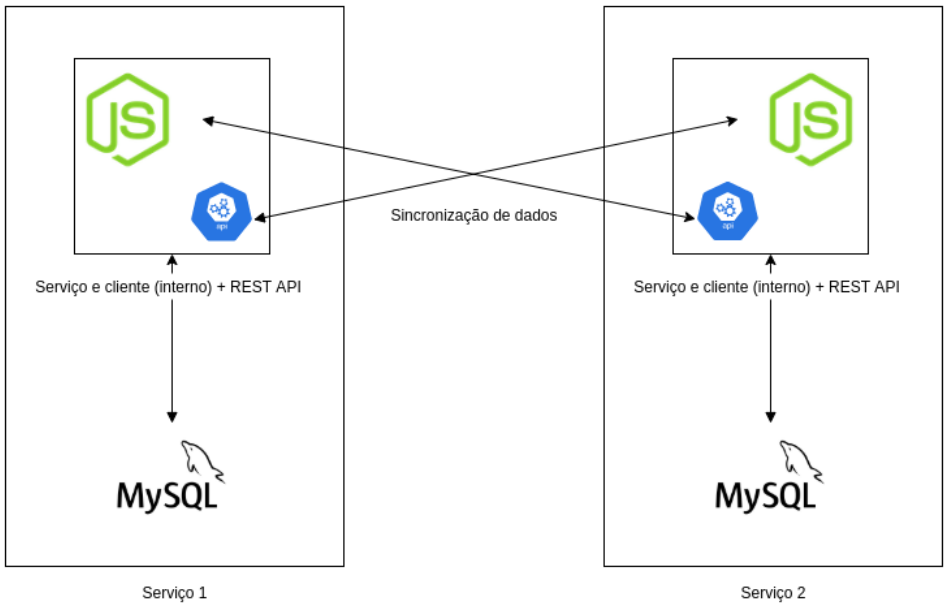

Figura 1: Visão genérica da arquitetura

2.1 Escolha de Tecnologias

2.1.1 Bases de Dados

Para as bases de dados que suportam os serviços desenvolvidos decidiu-se usar **MySQL**, por ser simples e rápido de usar, bem como pelo contacto prévio e familiaridade com este motor de base de dados.

2.1.2 Clientes (serviços)

Os clientes (e respetivos serviços) foram desenvolvidos em **Node.js** com a integração do **Pug** para as interfaces dos clientes.

2.1.3 Comunicação entre serviços

Ao nível da comunicação entre os dois serviços usou-se serviços web REST, através de JSON para transmissão de dados.

3 Implementação

3.1 Bases de Dados

Pegando no que tinha sido feito na primeira ficha, ao nível das bases de dados, para além de usar duas bases de dados separadas para cada serviço, a nomenclatura das tabelas e respetivas colunas de cada base de dados difere, bem como a própria organização dos dados, que não é exatamente igual nas duas bases de dados.

3.1.1 Serviço 1

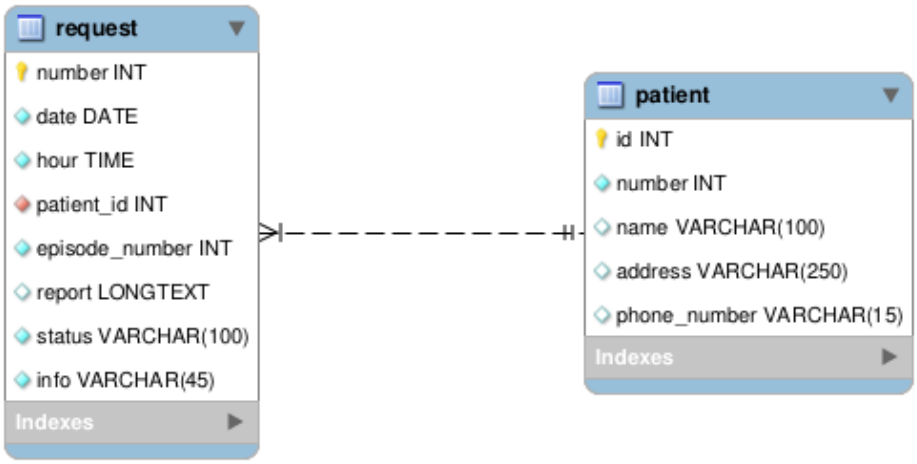

Figura 2: Serviço 1 - Esquema da Base de Dados

Como se pode ver na figura 2, o serviço 1 utiliza uma base de dados com o nome de ***medical_exams***. A informação dos pacientes é guardada na tabela ***patient***, sendo que os seus valores podem, ou não ser atualizados consoante a utilização do programa. A informação relativa aos pedidos de exames é guardada na tabela ***request***.

3.1.2 Serviço 2

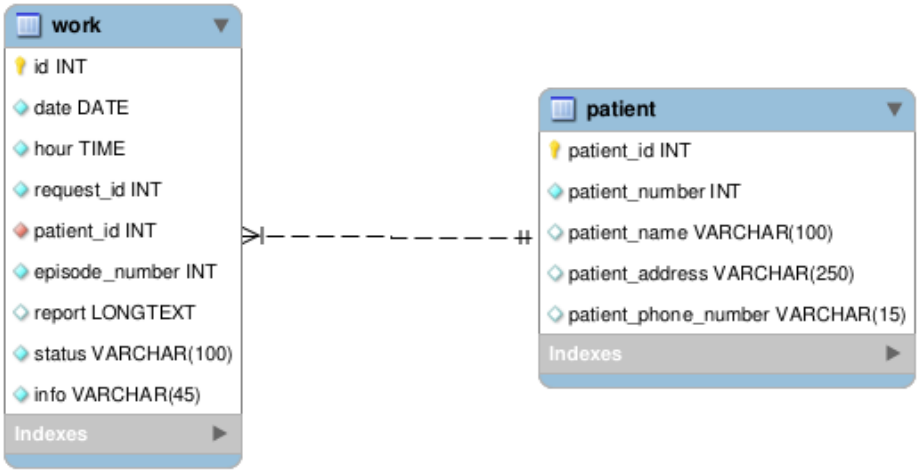

Figura 3: Serviço 2 - Esquema da Base de Dados

Tal como mostra a figura 3, o serviço 2 utiliza uma base de dados com o nome de ***medical_work_list***. A informação dos pacientes é guardada na tabela ***patient***, sendo que os seus valores podem, ou não ser atualizados consoante a utilização do programa. Por fim, a informação relativa aos pedidos de exames é guardada na tabela ***work***.

3.2 Serviços

Os serviços, tal como referido na secção 2.1, foram desenvolvidos em *Node.js* e cada um dos servidores aplicacionais se subdivide em serviço (REST API) e cliente (interface web). Esta secção vai tratar da REST API. Cada um dos servidores aplicacionais corre numa porta diferente, mais especificamente o Serviço 1 está a correr na porta 3000 e o Serviço 2 na porta 3001. Cada uma das APIs suportam as ações necessárias para manter a "sincronização"de informação entre os dois serviços. O serviço 1 suporta as seguintes ações:

- Listar todos os pedidos;
- Filtrar os pedidos por data;
- Editar o estado de um pedido;

- Adicionar o relatório de um pedido completado.

O serviço 2 suporta as seguintes ações:

- Listar todos os pacientes;
- Criar pacientes;
- Listar todos os pedidos;
- Listar os pedidos de 'hoje';
- Criar um pedido;
- Cancelar um pedido.

3.3 Clientes

Os clientes, tal como referido na secção 2.1, foram desenvolvidos em *Node.js* com o motor de templating *Pug* e contactam diretamente com middleware de backend do seu serviço 1 transportam as ações para o serviço 2 através da respetiva REST API.

```
router.get("/filter", function (req, res) {
  request.list().then((data) => {
    res.render("index", { view: "filter", data: data });
  });
});

router.get("/addPatient", function (req, res) {
  res.render("index", { view: "addPatient" });
});

router.post("/addRequest", function (req, res) {
  let [patient_id, patient_number] = req.body.patient_id.split(";");
  req.body.patient_id = patient_id;
  req.body.info = req.body.info.replace(/\r/g, "");
  request
    .insert(req.body)
    .then((data) => {
      req.body.request_id = data.insertId;
      req.body.patient_id = patient_number;
```

```
      axios
        .post(`http://localhost:3001/api/requests`, req.body)
          .then(() => {
            res.redirect("/");
          })
          .catch((error) => {
            res.render("error", { message: "Failed to submit new patient to service 2",
              error: error });
          });
      })
    .catch((error) => {
      res.render("error", { message: "Failed to submit new request", error: error });
    });
});
```

Código 1: Exemplo de código no cliente do Serviço 1

3.4 Migração de REST para SOAP

3.4.1 Desenvolvimento de servidores SOAP que permita as mesmas operações que as REST APIs

Como foi descrito acima, todo o sistema foi desenvolvido com o auxílio de serviços REST. Contudo, existem outro tipo de serviços web, também, largamente utilizados: os serviços SOAP. Este tipo de serviços constitui-se numa especificação de um protocolo de comunicação através de ficheiros XML.

Para esta transição, decidiu-se utilizar a framework de desenvolvimento web Spring, baseada em Java, para o desenvolvimento dos servidores. Para tal, primeiro seria necessário criar os respetivos projetos de Spring (Boot) e garantir que estes possuem a dependência wsdl4j.

O segunda passo será criar o domínio (métodos e parâmetros) para o nosso serviço (a restante exemplificação será feita apenas para o Serviço 1, sendo que o processo seria semelhante para o Serviço 2). Para tal, cria-se um esquema XML (ficheiro do tipo XSD), que o Spring irá tratar de exportar automaticamente como um WSDL. Um exemplo desse ficheiro a ser criado encontra-se no anexo A.

De seguida, criar as respetivas classes, que podem ser criadas manualmente

ou com recursos a *plugins* que criam as classes, consoante a especificação anterior. Após criar as classes, o próximo passo é criar os *endpoints* para efetivamente implementar as ações necessárias, tal como se pode ver no anexo B.

Deste modo, temos uma perspetiva e um plano de como estruturar e implementar os nossos serviços em **SOAP**, ao invés de **REST**.

3.4.2 Consumo dos serviços

Para o consumo dos serviços SOAP na aplicação Node.js, serão necessárias algumas alterações: desde logo a introdução de um *package* que auxilie o processo de consumir esses dados - request. O próximo passo é definir os envelopes SOAP necessários para consumir os dados, sendo o seguinte exemplo para o cancelamento de um pedido:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:gs="http://service1">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:completeRequest>
         <gs:reqId>1</gs:reqId>
      </gs:GetUsers>
   </soapenv:Body>
</soapenv:Envelope>
```

Código 2: Exemplo do XML para um pedido SOAP ao Serviço 1

Tendo isto, basta então fazer o pedido e o seu parsing, tal como se pode ver no seguinte exemplo, que mostra uma possível estratégia para tal:

```
const  request = require("request");
const xml2js = require("xml2js");

let xml =
‘<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:gs="http://service1">
   <soapenv:Header/>
   <soapenv:Body>
      <gs:completeRequest>
         <gs:reqId>1</gs:reqId>
      </gs:GetUsers>
```

```
    </soapenv:Body>
</soapenv:Envelope>‘;

var options = {
  url: 'http://localhost:8080/service1?wsdl', // apenas exemplificativo
  method: 'POST',
  body: xml,
  headers: {
    'Content-Type':'text/xml;charset=utf-8',
    'Accept-Encoding': 'gzip,deflate',
    'Content-Length':xml.length
  }
};

let callback = (error, response, body) => {
  if (!error && response.statusCode == 200) {
    console.log('Raw result', body);
    var parser = new xml2js.Parser({explicitArray: false, trim: true});
    parser.parseString(body, (err, result) => {
      console.log('JSON result', result);
    });
  };
  console.log('E', response.statusCode, response.statusMessage);
};
request(options, callback);
```

Código 3: Exemplo do XML para um pedido SOAP ao Serviço 1

A partir do momento em que se tem os dados em formato JSON, todo o processo se mantém, e as restantes alterações necessárias são mínimas, tais como: atualização da interpretação dos dados, visto que certamente a estrutura do JSON resultante do *parsing* é um pouco diferente da estrutura do JSON usado nos serviços REST.

4 Resultados

4.1 Apresentação

Com o desenvolvimento deste trabalho foi possível criar um cenário de interoperabilidade semântica em que dois serviços distintos comunicam de forma a completarem um macro-objetivo (a requisição e execução de exames médicos), cada um cumprindo com as suas responsabilidades e fazendo apenas aquilo que foi desenvolvido para fazer, delegando as restantes responsabilidades ao outro serviço.

4.1.1 Serviço 1

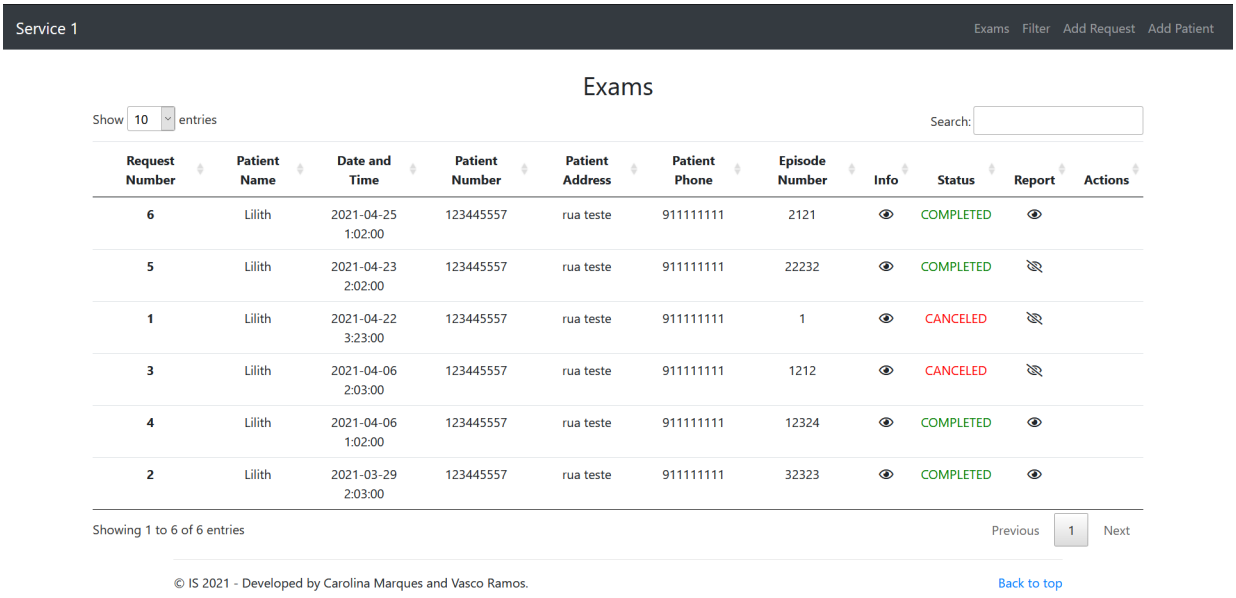

Figura 4: Serviço 1 - Listagem dos exames

O serviço 1 apresenta uma listagem com todos os exames e com as suas descrições (informações sobre o paciente e sobre o pedido em si).

Dependendo do estado do exame existe ainda outras possibilidades. No caso de CANCELED nada acontece. No caso de COMPLETED, consegue ver o relatório (caso este se encontre escrito) e no caso de TO BE EXECUTED, pode ainda cancelar o pedido.

Quando um pedido é cancelado ou completado, o serviço atualiza localmente o estado daquele pedido e comunica à API do outro serviço esta alteração de estado.

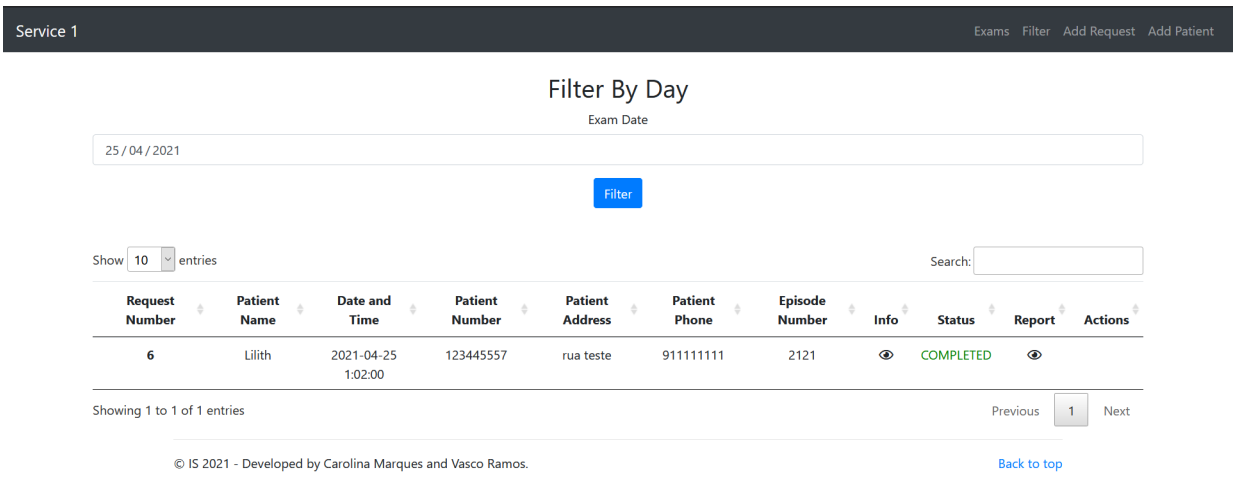

Figura 5: Filtrar Exames por dia

Este serviço permite ainda filtrar os exames dependendo do dia que seja especificado.

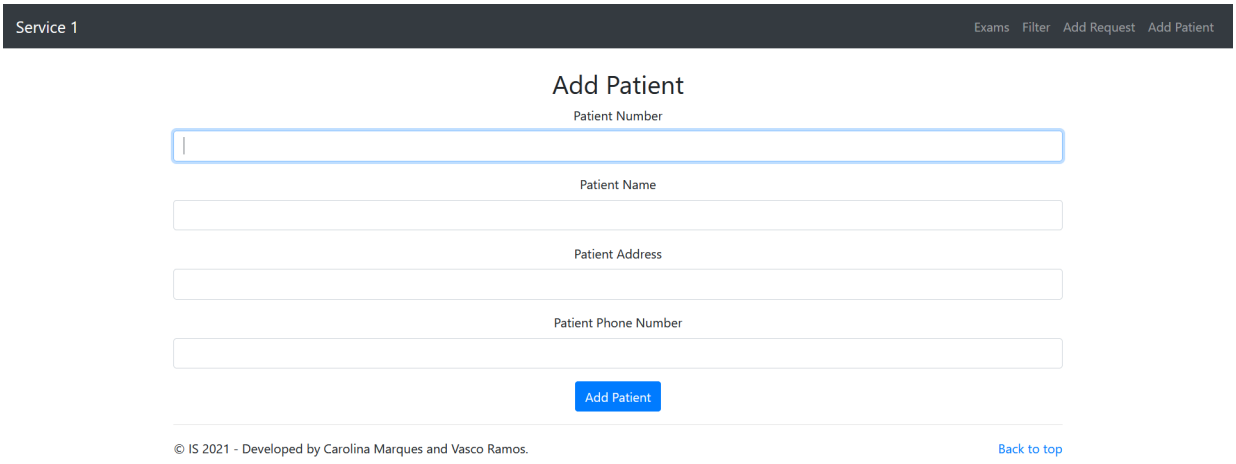

Figura 6: Registo de paciente no Serviço 1

Para se fazer um pedido, o serviço necessita de receber um paciente (Figura 6). Quando um paciente é registado, o serviço regista-o localmente e tal é comunicado

ao outro serviço, através da respetiva API. Estando o paciente criado, podemos agora criar o pedido.

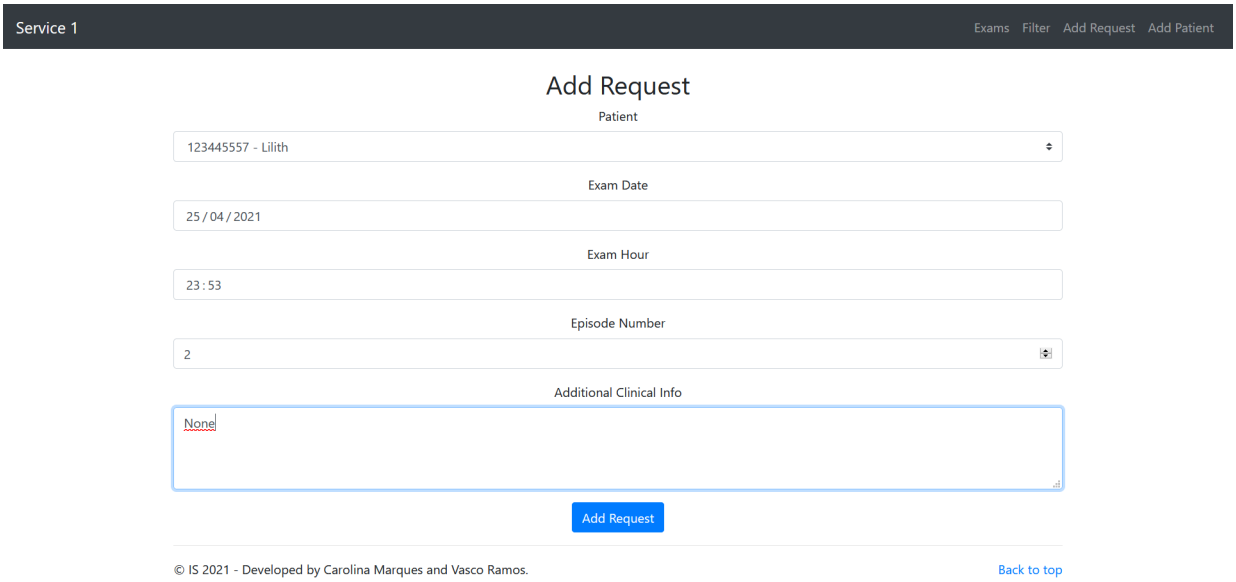

Figura 7: Criação de um Pedido com o Paciente Lilith

4.1.2 Serviço 2

Após a criação do pedido, este fica disponível nos dois serviços. Portanto, como se vê na Figura 8, o exame 7 encontra-se para ser executado e a possibilidade de cancelar este ou completar o mesmo está disponível.

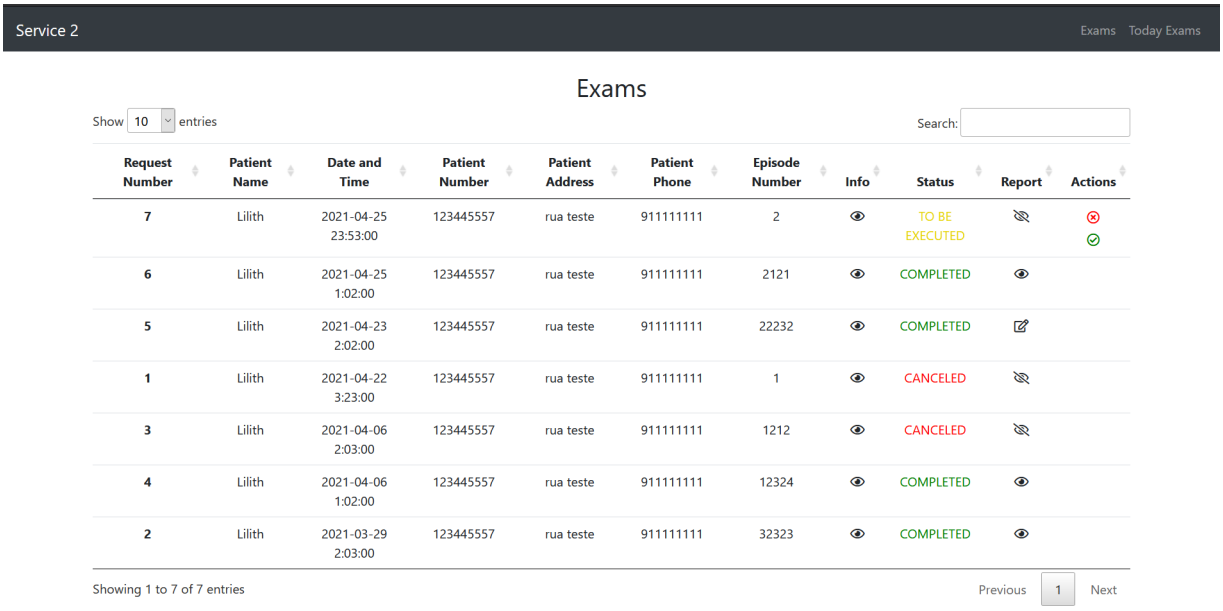

Figura 8: Home Page do Serviço 2

No caso de cancelamento, o processo encontra-se terminado. Já no caso de completo, podemos escrever o relatório para este exame. A escrita de um relatório neste serviço guarda na base de dados local o relatório e comunica esta escrita ao outro serviço.

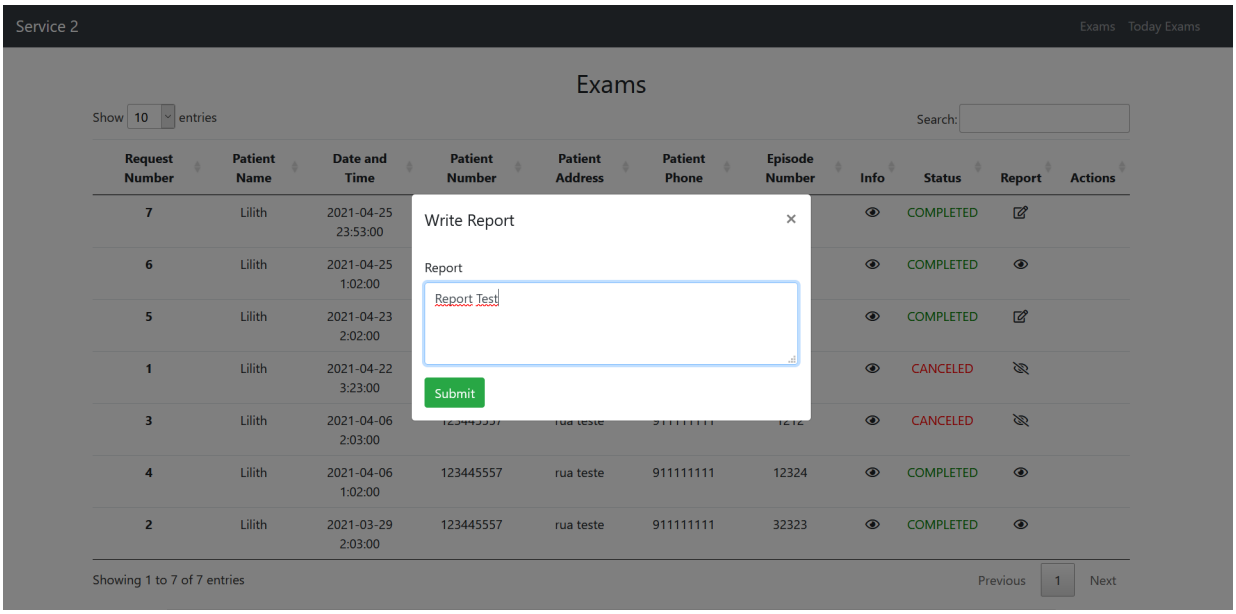

Figura 9: Exemplo de escrita de relatório

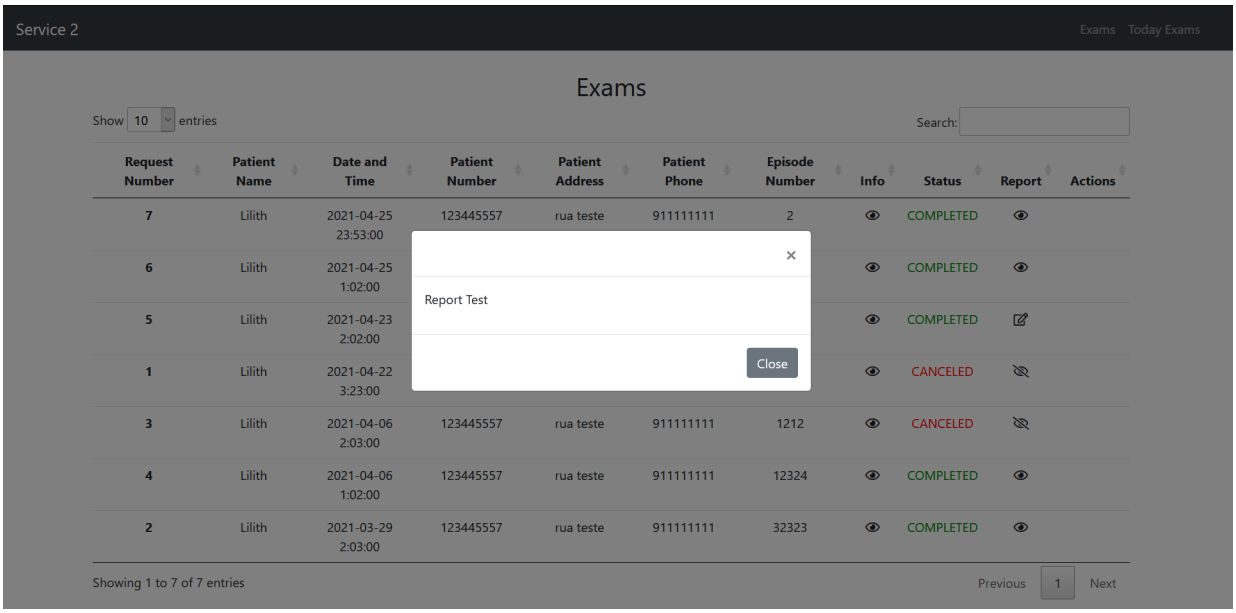

Figura 10: Ver relatório

Por fim, e de forma semelhante ao que acontece no serviço 1, este serviço permite filtrar os exames, mas em vez de ser de qualquer dia, apenas apresenta os exames do dia atual. Como se vê na Figura 11 os exames de hoje encontram-se listados.

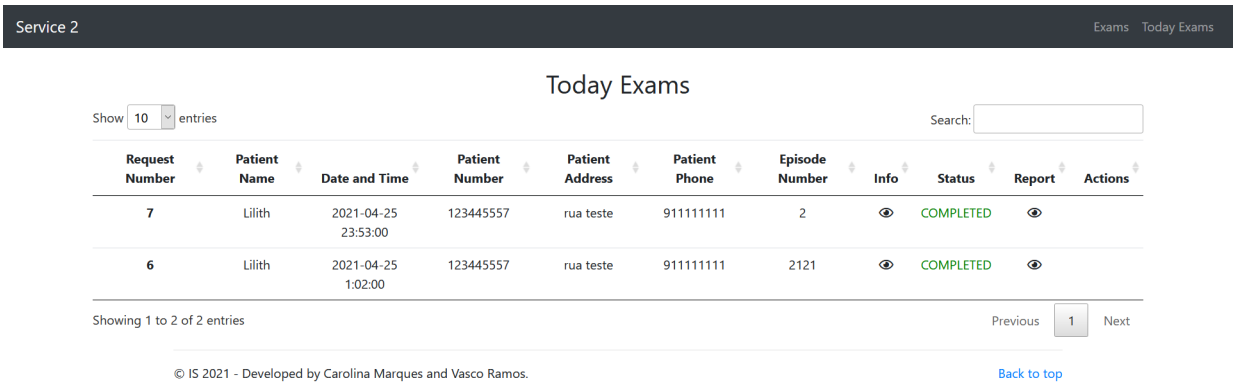

Figura 11: Listagem dos exames de hoje

4.2 Discussão

Tendo em conta o resultado atingido importa discutir e analisar o impacto do mesmo. Antes de mais, é relevante referir que, de facto, foi possível implementar com sucesso a arquitetura pretendida.

Tal como no trabalho anterior, a distribuição e separação de responsabilidades permite uma maior modularidade e eficácia do sistema no seu todo, o que faz com que não seja necessário executar todas as ações nos dois serviços, podendo algumas ações serem executadas no serviço 1 e as restantes no serviço 2. Esta abordagem permite, também, uma otimização da utilização dos recursos disponíveis para o sistema.

Esta arquitetura distribuída, tal como referido no trabalho anterior, tem, também, desvantagens, sendo que uma das mais proeminentes é a necessidade de garantir que ambos os serviços funcionam corretamente, pois, eventualmente, a falha de um pode comprometer o correto funcionamento do outro, pelo menos ao nível da atualização dos dados.

Relativamente à tecnologia que se usa para implementar este sistema, no trabalho anterior foi utilizado HL7 e neste trabalho utilizou-se serviços web (REST e SOAP). A utilização de serviços web ao invés de HL7 tem vantagens e desvantagens, tal como tudo em engenharia.

Os serviços web permitem, de um modo geral, um maior desacoplamento entre a lógica e semântica do negócio (a implementação das ações) e os mecanismos de comunicação (ao contrário do que se verifica com a utilização de HL7). Permitem, também, um consumo heterogéneo de dados pelas camadas de apresentação, o que também contribui para o desacoplamento dos vários componentes do sistema. Por fim, a utilização de serviços de web, ao invés de Hl7, dota a equipa responsável pelo sistema da capacidade/possibilidade de eventual evolução (possivelmente, gradual e iterativa) para uma arquitetura de micro-serviços.

Por outro lado, a utilização de serviços de web acaba por ter a desvantagem (quando comparado com HL7) de a estrutura e semântica das mensagens não ser standard, estável e universalmente bem conhecida, principalmente quando se fala de serviços REST com a utilização de JSON.

5 Conclusão

Com este trabalho foi implementado um mecanismo de interoperabilidade semântica entre dois serviços distintos, através de serviços web, mais propriamente REST (com as mensagem em JSON) e SOAP (com as mensagens em XML). Tendo em conta os objetivos definidos no enunciado, foi possível cumprir com todos os requisitos e ainda implementar a funcionalidade adicional de permitir a admissão de pacientes.

Em suma, a realização deste trabalho permitiu ao grupo alargar os seus conhecimentos relativamente a conceitos expostos nas aulas, pelo que, e tendo em conta o resultado final, considera-se que o resultado do trabalho cumpriu e ultrapassou as expetativas e requisitos definidos.

A Especificação do domínio

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://service1"
    targetNamespace="http://service1" elementFormDefault="qualified">

    <xs:element name="getRequests" type="void" />

    <xs:element name="getRequestsResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="request" type="tns:request"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="filterRequestsByDate">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="date" type="xs:date"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="filterRequestsByDateResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="request" type="tns:request"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="cancelRequest" type="tns:requestId" />

    <xs:element name="cancelRequestResponse" type"xs:string" />

    <xs:element name="completeRequest" type="tns:requestId" />

    <xs:element name="completeRequestResponse" type"xs:string" />

    <xs:element name="writeReport">
        <xsd:complexType>
            <xsd:sequence>
                <xs:element name="reqId" type="xs:int"/>
                <xs:element name="report" type="xs:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xs:element>
```

```
    <xs:element name="writeReportResponse" type"xs:string" />

    <xsd:complexType name="void">
        <xsd:sequence>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="requestId">
        <xsd:sequence>
            <xs:element name="reqId" type="xs:int"/>
        </xsd:sequence>
    </xsd:complexType>

    <xs:complexType name="request">
        <xs:sequence>
            <xs:element name="number" type="xs:int"/>
            <xs:element name="date" type="xs:date"/>
            <xs:element name="hour" type="xs:time"/>
            <xs:element name="episodeNumber" type="xs:int"/>
            <xs:element name="status" type="xs:string"/>
            <xs:element name="info" type="xs:string"/>
            <xs:element name="report" type="xs:string"/>
            <xs:element name="patient" type="tns:patient"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="patient">
        <xs:sequence>
            <xs:element name="id" type="xs:int"/>
            <xs:element name="number" type="xs:int"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="phoneNumber" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Código 4: Especificação do domínio

B Implementação dos endpoints

```java
@Endpoint
public class RequestEndpoint {

    private static final String NAMESPACE_URI = "http://service1";

    private RequestRepository requestRepository;

    @Autowired
    public RequestEndpoint(RequestRepository requestRepository) {
        this.requestRepository = requestRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getRequests")
    @ResponsePayload
    public GetRequestsResponse getRequests(@RequestPayload GetRequests request) {
        // implementation
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "filterRequestsByDate")
    @ResponsePayload
    public FilterRequestsByDateResponse filterByDate(@RequestPayload FilterRequestsByDate request) {
        // implementation
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "cancelRequest")
    @ResponsePayload
    public CancelRequestResponse cancelRequest(@RequestPayload CancelRequest request) {
        // implementation
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "completeRequest")
    @ResponsePayload
    public CompleteRequestResponse completeRequest(@RequestPayload CancelRequest request) {
        // implementation
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "writeReport")
    @ResponsePayload
    public WriteReportResponse writeReport(@RequestPayload WriteReport request) {
        // implementation
    }
}
```

Código 5: Implementação dos endpoints