Arquitetura de Comunicação do Health Simulator

Fernando Alex Helwanger[1], *Maria Priscila Rolim*[2], *Paulo Ricardo Muniz Barros*[3]
Universidade Feevale

Resumo

O desenvolvimento de sistemas informatizados para o ensino e educação tem crescido a cada ano e, com sua evolução, diversas tecnologias têm sido empregadas e testadas. O Health Simulator é um jogo que simula um Paciente Virtual (PV), projetado para ser um recurso adicional na formação de estudantes da área de saúde. O sistema é dividido em duas grandes áreas, em função das diferentes arquiteturas utilizadas, o aplicativo do jogo, chamado de *front-end*, e o servidor, que é utilizado como um repositório de informações, chamado de *back-end*. O principal objetivo deste artigo é apresentar a estrutura de comunicação utilizada para a troca de informações entre o *front-end* e o *back-end*, informando quais problemas ela propõe-se a resolver. O artigo apresenta a arquitetura de comunicação do Health Simulator e analisa os serviços utilizados por meio de pesquisa bibliográfica e experimentação.

Palavras-chave: Health Simulator, Jogos, Simuladores, REST, Web Service, Escalabilidade.

Abstract

The development of computerized systems for teaching and education increases every year, and, with its evolution, several technologies are used and tested. The Health Simulator is a game that simulates a Virtual Patient (VP), designed to be an additional resource in the formation of students in the health field. The system is divided in two great fields, in reason of the different architectures used, the application of the game, called back-end, and the server, that it is used as a repository of information, called back-end. The main objective of this work is present the communication structure used for the exchange of information between the front-end and the back-end, informing the problems that it proposes to resolve. The article presents the communication architecture of the Health Simulator and analyzes the services used through of bibliographic search and experimentation.

Keywords: Health Simulator, Games, Simulators, REST, Web Service, Scalability.

[1] Acadêmico de Ciência da Computação na Universidade Feevale.
[2] Acadêmica de Ciência da Computação na Universidade Feevale.
[3] Mestre em Informática e Educação na Saúde pela UFCSPA. Professor dos cursos de Ciências da computação e Jogos Digitais da Universidade Feevale

INTRODUÇÃO

O desenvolvimento de sistemas informatizados para o processo de ensino e aprendizagem tem crescido a cada ano e, com sua evolução, diversas tecnologias têm sido empregadas e testadas, com o intuito de agregar funcionalidades e benefícios em seu uso. Seguem, desta forma, ambientes complexos voltados ao ensino e à formação de indivíduos. Serviço Web é uma das soluções mais utilizadas atualmente para integração de sistemas complexos e comunicação entre diferentes aplicações (SOMMERVILLE, 2011).

O Health Simulator é um simulador do tipo Paciente Virtual (PV), com o propósito de simular a vida real em cenários clínicos, que permite o aprendizado de atos do profissional da saúde, obtendo a história clínica, exames e realizando diagnóstico e decisões terapêuticas (Orton e Mulhausen, 2008). A área da saúde (no que tange ao diagnóstico investigativo e a conduta terapêutica) caracteriza-se como um ambiente de domínio incerto, nestes ambientes destaca-se como alternativa de ensino a utilização de simuladores (BARROS, 2012).

De acordo com Holzinger e outros (2009), a grande vantagem da aprendizagem com simulações interativas está na natureza construtivista proporcionada pelos processos exploratórios de aprendizagem. Os alunos em sistemas complexos podem, por exemplo, modificar as variáveis de entrada, e com isto perceber as alterações provocadas no ambiente, recebendo diretamente *feedback*. Deste modo, percebem as consequências de suas ações, podendo alterar novamente os parâmetros, revisando suas ações e, por consequência, construir o seu conhecimento de forma dinâmica e interativa.

O Health Simulator foi projetado para ser um recurso adicional na formação de estudantes na área da Saúde. Para isto foi necessário o desenvolvimento de uma estrutura robusta que proporcionasse confiabilidade e segurança. Deste modo, o sistema foi dividido em duas grandes áreas em função das diferentes arquiteturas utilizadas, o aplicativo do jogo, chamado de *front-end*, e o servidor, que é utilizado como um repositório de informações, chamado de *back-end*.

O objetivo principal deste artigo é definir uma especificação para a estrutura de comunicação e troca de mensagens entre o *front-end* e o *back-end* do ambiente Health Simulator. O desenvolvimento deste meio de comunicação tem como principal objetivo uma alta escalabilidade. Um serviço escalável é um serviço que permite um aumento rápido e substancial em seu número de usuários, mantendo um bom desempenho (STEPHEN, 2001). Além disso, também é desejável um bom nível de portabilidade, ou seja, o mesmo serviço desenvolvido deve estar disponível em várias plataformas para utilização em necessidades futuras.

Em primeiro lugar, será apresentada uma visão geral do Health Simulator, explicando seu objetivo, utilização e estrutura. Após, será apresentada a definição de um *web service* RESTful, juntamente com os pontos fortes e fracos de sua implementação e a motivação para a aplicação no Health Simulator. Por fim, este trabalho mostrará alguns dos recursos implementados no Health Simulator, com o objetivo de apresentar o modelo e formato que servirá de base para o desenvolvimento de todo o serviço.

APRESENTAÇÃO DO HEALTH SIMULATOR

O sistema é dividido em duas grandes áreas, em função das diferentes arquiteturas utilizadas, o aplicativo do jogo, chamado de *front-end*, e o servidor, que é utilizado como um repositório de informações, chamado de *back-end*.

O *front-end* engloba a parte do simulador, a interface do aluno. Esta interface, no formato de um jogo sério (LIMA et al., 2015), será apresentada em 3 dimensões (3D), composta por espaços de atendimento à saúde e personagens que possam representar tanto profissionais da área quanto pacientes (Figura 1).

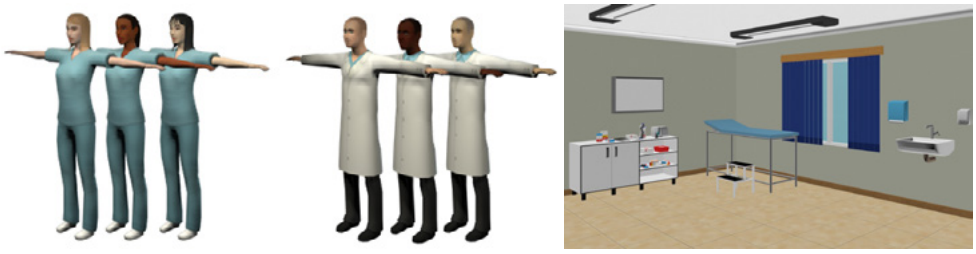

Figura 1 - Modelos de enfermeiras e médicos nas etnias branco, negro e asiático e exemplo de cenário para consultório Classe C (LIMA et al., 2015).

O *back-end*, por se tratar de uma estrutura extensa, foi dividido em três etapas distintas: Modelagem do conhecimento; Interface de administração e Serviço Web de comunicação.

A modelagem do conhecimento, no simulador proposto, será feita por especialistas, amparados pelas diretrizes clínicas, uma forma sistemática de orientação e delimitação do conteúdo, a ser desenvolvida a partir de evidências. Os componentes do raciocínio clínico serão representados por Redes Bayesianas. Redes Bayesianas são uma das técnicas de inteligência artificial para representação do conhecimento incerto (PEARL, 1993).

Posteriormente, com base no modelo bayesiano, o docente faz a formulação de casos clínicos, por meio de um ambiente web, denominado como Interface de Administração. O de-

senvolvimento desta interface foi realizado em um modelo de entidade e relacionamento, e a partir dele os casos de uso, juntamente com as regras do sistema e regras de interface. Todas as telas para o uso de professores na construção de casos clínicos e administração do sistema foram padronizadas, levando-se em consideração um design simples e de fácil utilização por parte dos professores (Figura 2).

Figura 2 - Tela de cadastro de Avatares.

Um caso clínico gerado no Health Simulator é armazenado no banco de dados, ficando disponível para utilização pelos alunos no jogo digital. Esta disponibilidade de informações é feita pela etapa 3, nominada como Serviço Web de comunicação.

O serviço de comunicação esta sendo desenvolvido com base em uma das soluções mais utilizadas para integração de sistemas e comunicação entre aplicações diferentes. Esta estrutura tecnológica permite a maior compatibilidade de softwares desenvolvidos em diferentes plataformas. O objetivo dessa arquitetura é atingir o máximo de escalabilidade e estar disponível ao maior número de usuários possível, sem que esta comunicação seja interceptada ou bloqueado por *firewalls* ou *proxies* de rede.

Desta forma, os Serviços Web fornecem uma solução que define um padrão para troca de mensagens através de uma arquitetura REST (*Representational State Transfer*) entre as aplicações cliente (jogo/cliente) e um fornecedor do serviço (*Back-end*) (SOMMERVILLE, 2011).

APRESENTAÇÃO DO ESTILO DE ARQUITETURA REST

Representational State Transfer (REST) é um estilo arquitetural de sistemas hipermídia distribuídos (FIELDING, 2000). Sistemas desenvolvidos nesse tipo de arquitetura são chamados de sistemas RESTful (RICHARDSON; RUBY, 2007).

Um *web service* RESTful, se comparado aos protocolos “WS-” (SOAP e WSDL são exemplos), tem a vantagem de aproveitar mais os recursos contidos no próprio protocolo HTTP, como códigos de retornos, cabeçalhos e verbos (WEBBER; PARASTATIDIS; ROBINSON,

2010). Com isto, há um nível a menos de abstração, o que simplifica a solução e pode diminuir a quantidade de dados trafegados.

Uma grande diferença entre os protocolos “WS-“ e um *web service* RESTful é que os protocolos “WS-” são baseados em operações, enquanto um *web service* RESTful é baseado em recursos (RICHARDSON; RUBY, 2007). Por exemplo, para manipular dados de uma pessoa, um *web service* SOAP disponibilizaria métodos do tipo: *getPeople*, *getPerson*, *savePerson*, etc. Já um *web service* RESTful para a mesma finalidade disponibilizaria um recurso do tipo “http://web.service/people” para manipular as pessoas, em que a operação a ser realizada seria baseada no verbo HTTP utilizado (GET, POST, PUT, DELETE, etc.).

Para que um sistema possa ser considerado RESTful, ele deve atender a um conjunto de restrições. Essas restrições têm como intuito atingir alta escalabilidade no serviço (FIELDING, 2000).

Cliente-Servidor

Uma das restrições de uma arquitetura REST é que o modelo de comunicação utilizado deve ser cliente-servidor (FIELDING, 2000). Neste modelo, a interação é iniciada por um cliente, que faz a requisição a um serviço disponibilizado pelo servidor. O servidor processa a requisição e envia uma resposta para o cliente (TANENBAUM; STEEN, 2006).

Com este modelo, a responsabilidade de cada grupo fica bem definida. O servidor é responsável pelo armazenamento dos dados e por disponibilizá-los aos clientes. Os clientes devem, de alguma forma, conectar ao servidor para a obtenção e manipulação da informação. Já que não existe dependência entre os clientes, a sua portabilidade para diversas plataformas é facilitada. A simplicidade do servidor melhora a sua escalabilidade. Além disso, esse modelo permite que os clientes e os servidores evoluam de maneira independente (FIELDING, 2000).

Stateless

Em uma arquitetura REST, a interação entre o cliente e o servidor deve ser *stateless*, ou seja, nenhum estado da sessão pode ser armazenado no servidor. Toda requisição deve conter a informação necessária para que seja entendida (FIELDING, 2000). Com isso, se ganha visibilidade, confiança e escalabilidade. A visibilidade é obtida porque basta observar a requisição para entendê-la, não é necessário procurar por nenhuma outra informação. A confiança é melhorada já que a recuperação de falhas parciais é facilitada. A escalabilidade é melhorada

porque o servidor não precisa armazenar algum estado entre as requisições e pode rapidamente liberar recursos. Além disso, a sua implementação é simplificada já que não é necessário gerenciar o contexto entre requisições.

Assim como a maioria das decisões arquiteturais, a restrição *stateless* possui alguns pontos negativos. Ela aumenta o tráfego de rede informando dados repetidos e diminui o controle do servidor, delegando o trabalho de controlar o estado da aplicação para o cliente (FIELDING, 2000).

Cache

Para aumentar a eficiência da rede, toda resposta do servidor deve ser sinalizada, implícita ou explicitamente, como *cacheable* ou *non-cacheable*. Se a resposta for *cacheable*, o cliente pode reutilizar os dados já recebidos em próximas requisições equivalentes (FIELDING, 2000). Com isto, algumas interações são eliminadas, melhorando a eficiência, escalabilidade e o desempenho percebido pelo usuário. No entanto, requisições *cacheable* podem diminuir a fidelidade dos dados caso sejam diferentes dos que seriam obtidos por meio de uma nova requisição.

Interface Uniforme

Em um sistema RESTful, todos os componentes possuem uma interface uniforme. Desta forma, o sistema torna-se mais simples e a visibilidade das interações é melhorada (FIELDING, 2000). Além disso, a implementação fica desacoplada em relação aos serviços disponibilizados, o que permite que as duas partes evoluam de forma independente. O ponto negativo dessa abordagem é que, muitas vezes, uma interface generalizada pode não ser tão eficiente quanto uma específica para o problema. Para que um sistema possua uma interface uniforme, ele deve atender a quatro restrições: identificação de recursos, manipulação de recursos por meio de representações, mensagens auto-descritivas e hipermídia como o motor do estado da aplicação (FIELDING, 2000).

Embora teoricamente o protocolo HTTP seja apenas um dos muitos protocolos que podem ser utilizados para disponibilizar uma interface uniforme, a maioria das aplicações RESTful são implementadas utilizando-o (WEBBER; PARASTATIDIS; ROBINSON, 2010). No protocolo HTTP, a identificação de recursos é feita por meio de URIs (*Uniform Resource Identifiers*) (STEPHEN, 2001). Um exemplo de URI que identifica um recurso de produtos seria: http://nome.do.servidor/produtos.

Um recurso identificado por uma URI pode possuir diversos tipos de representação. O formato de representação de um recurso não precisa ser o mesmo no qual ele é armazenado no servidor. Este encapsulamento garante um baixo acoplamento (WEBBER; PARASTATIDIS; ROBINSON, 2010), o que facilita, por exemplo, que o formato de armazenamento do recurso seja alterado por outro mais eficiente sem impactar na maneira que os clientes utilizam o serviço. Alguns dos formatos de representação mais comuns na web são: HTML para documentos, PNG e JPEG para imagens, MPEG para vídeos, XML e JSON para dados, entre outros. Embora alguns serviços utilizem extensões na URI para identificar o formato da representação, como, por exemplo, http://nome.do.servidor/produtos.xml, existem maneiras mais robustas de resolver este problema. A técnica mais comum para isto é a negociação de conteúdo, na qual o cliente especifica no cabeçalho *Accept* do protocolo HTTP quais formatos ele suporta, e o servidor escolhe um formato dessa lista como o mais apropriado (WEBBER; PARASTATIDIS; ROBINSON, 2010). A URI deve apenas identificar um recurso no servidor, e não a maneira como ele é armazenado.

Mensagens são auto-descritivas, possuem toda a informação necessária que descreve a maneira como devem ser interpretadas. No protocolo HTTP, o cabeçalho da resposta *Content-Type* possui o tipo da mídia do corpo da resposta (RICHARDSON; RUBY, 2007). Desta forma, o cliente pode identificar uma maneira de interpretar a mensagem.

Hipermídia como o motor do estado da aplicação significa que o estado de uma “sessão” HTTP não é armazenado no servidor, mas sim definido pelo caminho que o cliente toma navegando pela Web (RICHARDSON; RUBY, 2007). O cliente tem conhecimento do ponto de entrada para a aplicação, a partir dele, o servidor envia, em cada resposta, *links* que apontam para os próximos recursos que podem ser acessados, guiando o caminho que o cliente pode tomar na aplicação.

Sistema de Camadas

Um sistema de camadas pode ser visto como uma “torre” de camadas, em que cada uma é empilhada sobre outra. A camada que está em cima usa os serviços da camada abaixo dela, sendo que a camada que está embaixo não tem conhecimento da camada que está acima. Além disso, cada camada só é visível para as camadas que utilizam serviços disponibilizados por ela, ou seja, em um ambiente onde a camada 4 utiliza serviços da camada 3, e a 3 utiliza serviços da camada 2, a camada 4 não tem conhecimento da existência da 2 (FOWLER et al, 2012). Portanto, uma restrição que deve ser atendida por uma arquitetura REST é que ela deve

ser uma arquitetura de camadas, onde cada componente não consegue “enxergar” além da camada com a qual interage imediatamente (FIELDING, 2000). Desta forma, a complexidade de cada componente fica limitada. Novas camadas podem ser inseridas para encapsular serviços legados e para proteger novos serviços de clientes legados, simplificando os componentes. Camadas intermediares podem ser inseridas para melhorar a escalabilidade, permitindo o balanceamento de carga do serviço entre múltiplas redes e processadores.

Código Sob-Demanda

A última restrição para uma arquitetura REST é a possibilidade de o cliente baixar um código (na forma de *applets* ou *scripts*) do servidor (FIELDING, 2000). Isso simplifica o cliente, reduzindo o número de funcionalidades a serem implementadas, além de melhorar a extensibilidade. Porém, isso diminui a visibilidade do sistema, e portanto é uma restrição opcional para uma aplicação RESTful.

Dado que um dos principais objetivos do *web service* que serve como base para a maior parte do processamento de informação do Health Simulator é obter uma alta escalabilidade e estar disponível ao maior número de clientes possível, com as características citadas nesta seção uma arquitetura RESTful mostra-se adequada para o problema.

SERVIÇO DE COMUNICAÇÃO DO HEALTH SIMULATOR

Até o momento, foram definidos cinco recursos RESTful no *web service*: usuário, *avatar*, instituição, caso clínico e simulação. Eles são fundamentais para as interações entre o jogo (*front-end*) e Servidor (*back-end*). Todas as ações são feitas por meio de requisições HTTP com os dados serializados no formato JSON (JSON, 2014).

Foram definidos “turnos”, sendo estes intervalos de tempo previamente definidos em que o jogo irá interagir com o *back-end*. Na estrutura apresentada, o jogo é responsável por dar início aos atos comunicativos, uma vez que os demais atos são em decorrência dos turnos. A cada troca de turno ocorre um novo ato comunicativo, fazendo uso dos métodos desenvolvidos. A seguir será apresentado um dos recursos já desenvolvidos. Seguindo a restrição de que um serviço RESTful deve possuir uma interface uniforme, todos os outros métodos seguem o mesmo modelo e padrão.

O recurso /api/v1/usuario é utilizado para gerenciar os dados do usuário no *web service*. A Tabela 1 mostra as possíveis ações que podem ser feitas neste recurso.

URL	Verbo	Retornos
/api/v1/usuario	GET	200 – OK 404 – NOT FOUND
/api/v1/usuario/{id}	GET	200 – OK 404 – NOT FOUND
/api/v1/usuário	POST	201 – CREATED 400 – BAD REQUEST
/api/v1/usuario/{id}	PUT	204 – NO CONTENT 400 – BAD REQUEST 404 – NOT FOUND
/api/v1/usuario/{id}	DELETE	204 – NO CONTENT 404 – NOT FOUND 405 – METHOD NOT ALLOWED

Tabela 1. Definição de possíveis ações no recurso de usuários.

A primeira ação permite a listagem de todos os usuários cadastrados. Caso não exista nenhum, o código 404 é retornado, caso exista, uma listagem serializada em JSON é retornada ao cliente.

A segunda ação permite consultar dados de um usuário específico, passando na URL o número identificador referente ao usuário desejado. Os códigos de retorno são semelhantes ao primeiro método, a única diferença é que o retorno da requisição traz os dados de um único usuário, e não uma lista.

A terceira ação possibilita o cadastramento de um novo usuário, passando no corpo da requisição dos dados serializados em JSON. Caso os dados informados forem inválidos, será retornado o erro 400, caso sejam válidos será retornado o código 201, informando que o registro foi criado.

A quarta ação é semelhante a terceira, mas serve para alterar dados de um usuário cadastrado, identificado por meio do ID informado na URL. A diferença dos códigos de retorno, em relação à terceira ação, é que quando o registro é alterado o código 204 é retornado. Caso não for possível encontrar o usuário, o código 404 é retornado.

A última ação serve para permitir a exclusão de um usuário, identificado pelo ID presente na URL. Retorna o código 204 ao excluir com sucesso, 404 ao não encontrar e 405 caso o cliente não esteja autorizado a fazer a exclusão.

Além dos códigos de retorno definidos na Tabela 1, todos podem retornar o código 500 – INTERNAL SERVER ERROR caso ocorra um erro inesperado no processamento da solicitação.

CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma abordagem utilizada na comunicação multiplataforma do projeto Health Simulator. A abordagem baseou-se nos estudos sobre essas tecnologias e seu uso em desenvolvimento de aplicações híbridas em ambientes heterogêneos, que é o caso do projeto em questão.

O projeto encontra-se em fase de desenvolvimento, no qual o protótipo possui cinco recursos definidos e funcionais. Ao final desta fase, já pode ser evidenciado que a tecnologia atende os requisitos necessários podendo consolidar-se no ambiente. Além disso, demonstra a integração de recursos da área de jogos, que dá a aplicação características positivas como atratividade realismo, com formalismos diferenciados da área da computação aplicada como redes bayesianas, possibilitando a comunicação entre esses dois agentes interativos.

Como trabalhos futuros verifica-se a finalização do protótipo, desenvolvendo os demais métodos para validação junto ao jogo Health Simulator.

REFERÊNCIAS

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Doctoral dissertation, University of California, Irvine, 2000.

FOWLER, Martin et al. **Patterns of Enterprise Application Architecture**. Addison-Wesley Professional, 2012.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services**. O'Reilly Media, 2007.

STEPHEN, Thomas. **HTTP Essentials**: Protocols for Secure, Scalable Web Sites. John Wiley & Sons, 2001.

TANENBAUM, Andrew Stuart; STEEN, Maarten Van. **Distributed Systems:** Principles and Paradigms. 2 ed. Pearson Prentice Hall, 2006.

WEBBER, Jim; PARASTATIDIS, Savas; ROBINSON; Ian. **REST in Practice: Hypermedia and Systems Architecture**. O'Reilly Media, 2010.

HOLZINGER, A., KICKMEIER-Rust, M.D., WASSERTHEURER, S., HESSINGER, M.: **Learning performance with interactive simulations in medical education:** Lessons learned from results of learning complex physiological models with the HAEMOdynamics SIMulator. Computers & Education 52(2), 292–301, 2009.

JSON, O. JSON - **JavaScript Object Notation.** Disponível em: <http://www.json.org/>. Acesso em: 27/10/2014

SOMMERVILLE, I. **Arquitetura orientada a serviços. Engenharia de Software.** 9th ed., p.355–368. São Paulo: Person Prentice Hall, 2011.

Pearl, Judea. **Belief Networks Revisited. Artificial Intelligence.** Amsterdam: Elsevier, v.59, p.49-56, 1993.

Barros, P. R. M., Cazella, S. C., Bez, M., Flores, C. D., Dahmer, A., Mossmann, J. B., Maroni, V. (2012). **Um simulador de casos clínicos complexos no processo de aprendizagem em saúde.** RENOTE, 10(1), 2012.

Lima, A., Stahnke, F., Barros, P., Benetti, D., Mello, B., & Cervi, G.. **Projeto para desenvolvimento do Simulador Health Simulator.** Anais do Computer on the Beach, 279-288. 2015.