UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LEONARDO DOS SANTOS CONCEIÇÃO

Web service para acesso a dados da aplicação Caronas

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Marcelo Soares Pimenta

Porto Alegre
2017

AGRADECIMENTOS

Agradeço aos meus pais, por terem me dado condições para estudar, chegar à UFRGS, e concluir este trabalho.

Também agradeço ao professor Marcelo Soares Pimenta pela paciência e orientação durante este trabalho.

RESUMO

Este trabalho descreve o desenvolvimento e a implementação do *web service* que serve como *back-end* da aplicação de caronas solidárias Caronas. O *web service* descrito faz uma ponte entre os bancos de dados do Centro de Processamento de Dados da UFRGS e as aplicações clientes. A implementação é baseada estilo arquitetural REST, e responde em formato JSON e XML.

Palavras-chave: REST, *web service*, banco de dados.

Data access web service for the Caronas application

ABSTRACT

This work describes the development and implementation of the Caronas carpooling app back-end web service. The described web service bridges the UFRGS's Centro de Processamento de Dados (Data processing center) data banks to the client applications. The web service's implementation is REST based, and its responses are in the JSON or XML formats.

Keywords: REST, web service, database.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CPD	Centro de Processamento de Dados
GWT	Google *Web Toolkit*
RPC	*Remote Procedure Call*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTTP	*Hypertext Transfer Protocol*
JSON	*JavaScript Object Notation*
REST	*Representational State Transfer*
ROA	*Resource Oriented Architecture*
SQL	*Structured Query Language*
UFRGS	Universidade Federal do Rio Grande do Sul
URI	*Uniform Resource Identifier*

SUMÁRIO

1 INTRODUÇÃO

O Caronas é uma aplicação desenvolvida para plataformas móveis com o propósito de ajudar alunos, professores e funcionários da Universidade Federal do Rio Grande do Sul tanto a anunciarem sua disposição de oferecer transporte entre campi da universidade como a encontrar tais anúncios e solicitar uma vaga em um deles.

Para exercer sua função, a aplicação necessita das informações de anúncios e solicitações criadas pelos usuários, assim como informações básicas como nomes e números de cartões dos mesmos. Todas estas informações são armazenadas em bancos de dados da universidade, e precisam ser resgatados e armazenados diversas vezes pela aplicação durante o seu funcionamento.

Para evitar os vários problemas relacionados ao acesso direto da aplicação aos bancos de dados (dificuldade de implementação e manutenção do código, segurança comprometida), é necessária a imposição de uma ponte entre os bancos e a aplicação cliente, que neste caso tomou a forma de um *web service*, ou serviço *web*[*], que é o foco deste trabalho. O autor encarregou-se de desenvolver esse *web service* enquanto atuava como bolsista no CPD, o Centro de Processamento de Dados da UFRGS, onde o Caronas é desenvolvido.

Esse *web service* deve oferecer segurança às transações entre as duas partes, ter performance relativamente rápida, devido ao uso de aplicações clientes em plataformas móveis, e deixar fácil a compreensão da implementação das chamadas necessárias para o acesso ao banco de dados a fim de facilitar sua manutenção e eventuais mudanças.

A implementação desse *web service* foi feita usando a linguagem PHP aliada ao *framework* Yii 2, que oferece suporte à criação de *web services* no estilo arquitetural REST, combinação já usada no CPD. Ela faz parte da API UFRGS, um conjunto de serviços que atende outras aplicações criadas no CPD.

1.2 Objetivo do trabalho

Este trabalho foca-se em projetar e desenvolver um *web service* que possa servir como *back-end* para a aplicação Caronas. Para atender este objetivo, a aplicação deve ser segura, ter respostas de tamanho moderado devido à natureza móvel da aplicação e de fácil manutenção e compreensão pelos membros do CPD da UFRGS. É planejada a liberação do *web service*

*Neste trabalho, a implementação é alternadamente referida como *web service*, *back-end* ou serviço, tratando-se sempre do mesmo projeto.

diretamente pelo público no futuro, portanto é necessário deixar as fundações na implementação para que isso seja possível.

1.3 Estrutura do trabalho

Este trabalho está dividido em 6 capítulos, descritos a seguir.

Capítulo 1, Introdução – Contém o contexto deste trabalho, seus objetivos e sua estrutura.

Capítulo 2, Fundamentos e Tecnologias – Faz uma revisão dos fundamentos técnicos e tecnologias utilizadas no desenvolvimento do serviço.

Capítulo 3, Trabalhos Relacionados – Apresenta trabalhos relevantes ao assunto deste trabalho, e *web services* comerciais semelhantes ao desenvolvido.

Capítulo 4, Arquitetura do *Back-end* do Aplicativo Caronas – Descreve a arquitetura geral do *web service* desenvolvido, incluindo sua posição no sistema já criado pelo CPD.

Capítulo 5, *Back-end* do Aplicativo Caronas – Lista os requisitos do serviço, mostra a metodologia usada no desenvolvimento do mesmo e como ele é utilizado.

Capítulo 6, Conclusão – Resume os resultados do trabalho, as limitações da implementação e possíveis desenvolvimentos futuros do *back-end*.

2 FUNDAMENTOS E TECNOLOGIAS

Neste capítulo é feita uma revisão dos fundamentos e tecnologias utilizadas no desenvolvimento do *web service*. A seção 2.1 trata de SOAP e REST, duas classes de *web service*. A seção 2.2 trata sobre REST, *REpresentational State Transfer*, o estilo arquitetural introduzido por Roy Fielding. A seção 2.3 trata sobre REST como arquitetura aplicada a *web services*, como escrito por Leonard Richardson e Sam Ruby. A seção 2.4 trata sobre as ferramentas utilizadas na implementação do *web service*.

2.1 Serviços SOAP e REST

Existem dois tipos de *web service* em maior uso atualmente, os que utilizam o protocolo SOAP (*Simple Object Access Protocol*, Protocolo Simples de Acesso a Objetos) e os que seguem o estilo arquitetural REST (*REpresentational State Transfer*, ou Transferência Representacional de Estado), também chamados de *REstful Web Services*.

SOAP é um protocolo completo, com mensagens pré-definidas e o necessário *overhead* e pré-implementação para o seu processamento. Uma referência para o SOAP é oferecida por Gudgin et al (2007). Em contraste, serviços do tipo REST apenas seguem o estilo arquitetural REST, que prioriza o uso de características já existentes na estrutura da Internet, evitando o uso de novos “envelopes” para suas mensagens. Não há padrões oficiais para serviços REST. Por suas mensagens utilizarem o protocolo HTTP, qualquer aplicação com capacidade de *browser* pode interpretar suas mensagens, o que lhes dá certa versatilidade.

A API UFRGS desenvolvida no CPD, da qual o serviço desenvolvido neste trabalho fará parte, segue o estilo REST de acordo com o framework Yii 2 utilizado em sua implementação. Considerando isso, adicionando a sua maior facilidade de entendimento, melhor performance para tarefas simples, e popularidade maior considerando-se *web services* em uso, foi decidido que serviço implementado neste trabalho seguirá o estilo arquitetural REST.

2.2 REST

REST, *Representational State Transfer,* é um estilo arquitetural introduzido por Fielding (2000) em sua tese de doutorado. Em seu trabalho, ele referencia e avalia diversos

estilos de arquitetura baseada em redes e então introduz REST, um estilo híbrido derivado de vários dos estilos antes referenciados, combinado com restrições adicionais que definem uma interface de conectores uniformes.

Um conector, Fielding (2000) define, é um mecanismo abstrato que media comunicação, coordenação ou cooperação entre componentes. Componentes, ele define, como uma unidade abstrata de instruções de software e estado interno que provém uma transformação de dados via sua interface.

Fielding (2000) descreve REST a partir de um sistema em que não há restrições entre os limites de seus componentes, e então aplicando as seguintes restrições:

Cliente-servidor – Baseado na separação de responsabilidades. Separa as responsabilidades da interface de usuário e do armazenamento de dados, assim aumentando a portabilidade da interface de usuário entre várias plataformas e melhorando a escalabilidade pela simplificação dos componentes. Porém, considerando a *web*, talvez o fato mais significante seja que a separação permite a esses componentes uma evolução independente, então suportando o requerimento de escala da Internet que gera a necessidade de múltiplos domínios organizacionais.

Comunicação *stateless* – Restringe a comunicação entre cliente e servidor para que seja sempre de uma natureza *stateless* (sem estado), ou seja, toda requisição do cliente ao servidor precisa ter todos os dados necessários para o entendimento da mesma, não sendo possível tomar vantagem da utilização de dados já no servidor. Assim, o estado da sessão é contido inteiramente no cliente.

Esta restrição induz as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é melhorada devido à necessidade de verificação de apenas uma troca de dados por requisição por um sistema de monitoramento para entender a natureza integral da mesma. A confiabilidade é melhorada porque a recuperação após falhas parciais é facilitada. A escalabilidade é melhorada por dar ao servidor a oportunidade de liberar recursos rapidamente, e ainda simplifica a implementação por razão de o servidor não precisar gerenciar recursos entre requisições.

Desvantagens trazidas por essa restrição são diminuição da performance da rede devido ao *overhead* causado pela repetição de dados entre as requisições, devido à impossibilidade de armazenamento de dados de no servidor, e a redução do controle do

servidor em relação à consistência de comportamento da aplicação, já que a aplicação se torna dependente da correta implementação de semânticas através de múltiplas versões de cliente.

***Cache* –** Adicionada como forma de aumentar a eficiência da rede. Essa restrição requer que os dados dentro de uma resposta a uma requisição sejam implicitamente ou explicitamente marcados como armazenáveis ou não armazenáveis em *cache*. Se uma resposta é armazenável em *cache*, então o *cache* do cliente tem o direito de reutilizar a mesma resposta para requisições equivalentes posteriores.

Esta restrição permite a eliminação completa ou parcial de várias interações entre cliente e servidor, melhorando eficiência, escalabilidade, e performance percebida pelo usuário através da redução da latência média de uma série de interações. A desvantagem é que um *cache* pode diminuir confiabilidade se os dados contidos no mesmo forem significativamente diferentes do que os dados que seriam recebidos se a requisição fosse feita diretamente ao servidor.

Interface uniforme – Aplicando-se o princípio de generalização da engenharia de software, a arquitetura geral do sistema é simplificada e a visibilidade das operações é aumentada. Implementações são desacopladas dos serviços que elas provêm, o que encoraja a capacidade de evoluírem independentemente. A desvantagem é que uma interface uniforme degrada a eficiência, já que informações são trocadas de forma padronizada no lugar de um formato específico às necessidades da aplicação.

Essa restrição é a característica principal que distingue o estilo arquitetural REST de outros estilos baseados em redes.

Sistema em camadas – Restrição inserida para melhorar ainda mais o comportamento para requerimentos de escala da Internet. O estilo de sistema em camadas permite a uma arquitetura a sua composição por camadas hierárquicas restringindo o comportamento de componentes tal que cada componente não pode “ver” além da camada imediata com a qual está interagindo. Restringindo-se o conhecimento do sistema a uma única camada, há uma limitação na complexidade geral do sistema e promove-se independência do substrato. Camadas podem encapsular serviços legados do sistema e proteger novos serviços de clientes legados, simplificando componentes, movendo funcionalidades raramente usadas para um intermediário compartilhado. Intermediários podem ser usados para aumentar a escalabilidade do sistema permitindo o balanceamento de carga de serviços através de múltiplas redes e processadores.

A principal desvantagem trazida por esta restrição são o *overhead* e latência adicionais causadas ao processamento de dados, reduzindo a performance percebida pelo usuário. Para um sistema baseado em redes que suporta as restrições de *cache*, esta desvantagem pode ser suprimida pelos benefícios trazidos por *caches* compartilhados nos intermediários. A inserção de *caches* compartilhados nos limites de um domínio organizacional pode resultar em benefícios significativos na performance. Tais camadas também permitem políticas de segurança a serem aplicadas em dados cruzando o limite organizacional, como requerido para o uso de *firewalls*.

Código sob demanda – REST permite que a funcionalidade de clientes seja estendida baixando-se e executando-se código na forma de *applets* ou *scripts*. Isso simplifica clientes reduzindo o número de características necessárias que devem ser previamente implementadas. Permitir que características sejam baixadas após a distribuição dos clientes melhora a extensibilidade do sistema. Em contrapartida, essa permissão também reduz a visibilidade, e portanto é apenas uma restrição opcional na REST.

Resumidamente, REST é estilo arquitetural para sistemas distribuídos de hipermídia. REST provê um conjunto de restrições arquiteturais que, quando aplicadas como um todo, dão ênfase à escalabilidade das interações entre componentes, à generalização de interfaces, à distribuição independente de componentes, e à inserção de componentes intermediários para reduzir a latência durante interações, aumentar a segurança, e encapsular sistemas legados (FIELDING, 2000).

2.3 Web services RESTful

Web services surgiram para suprir a necessidade de integração entre diferentes aplicações. Eles servem como interfaces, conectadas através da Internet, que possibilitam a troca de dados entre elas. Um *web service* RESTful, ou simplesmente REST, é um serviço *web* baseado no estilo arquitetural REST, funcionando através do protocolo HTTP. Não há um padrão bem definido para este tipo de serviço, mas há estilos arquiteturais associados a eles. Segundo Sousa (2014), serviços *web* do tipo RESTful são do tipo mais utilizado hoje em dia, devido à sua simplicidade, escalabilidade e versatilidade.

Um dos estilos arquiteturais usados em *web services* é chamado *Resource Oriented Architecture* (Arquitetura Orientada a Recursos), elaborado por Richardson e Ruby (2007).

Em serviços *web* RESTful há o conceito de recurso. Segundo Richardson e Ruby (2007), um recurso é qualquer coisa importante o suficiente para que possa ser referenciado como uma coisa em si. Na prática, isso geralmente equivale a algum tipo de dado ou representação que deve ser representado de forma independente. Por exemplo, neste trabalho o recurso "oferecimento" é uma representação de dados sobre um oferecimento de carona.

Cada recurso é associado a um URI (*Universal Resource Identifier*). Enquanto essa é a regra geral, um certo dado pode ser associado a mais de um URI, gerando mais de um recurso. Por exemplo, as seguintes URIs:

http://www.exemplo.com.br/programa/versao2.0.zip
http://www.exemplo.com.br/programa/ultimaversao.zip

Podem apontar para o mesmo dado, sendo dois recursos diferentes, tratando de conceitos diferentes (um sempre apontando para a última versão do programa, e o outro para a versão 2.0, que no momento é a última). O URI é ao mesmo tempo o endereço e o nome do recurso. Então no exemplo anterior, o recurso "oferecimento" seria melhor representado por algo como:

http://www.exemplo.com.br/webservice/oferecimento

Ou abreviando-se, /oferecimento.

Richardson e Ruby (2007) descrevem as características da ROA a seguir:

Addressability **(endereçabilidade) –** Um serviço é endereçável se ele expõe características relevantes de seu conjunto de dados como recursos.

Statelessness **(não há estado)** – Cada requisição HTTP ocorre isoladamente. Em prática, cada requisição de um cliente ao servidor deve conter os dados necessários para a execução da mesma. Característica derivada da restrição REST descrita na seção 2.2, comunicação *stateless*.

Connectedness **(conectividade) –** Característica conhecida como "hipermídia como motor do estado da aplicação" (HATEOAS, na sigla em inglês). Em prática, traz o significado

de que adicionar ligações a outros recursos quando possível e coerente melhora a usabilidade do cliente. Por exemplo, um serviço que retorna os *posts* criados por um usuário em um *blog* pode trazer na resposta os URIs para os *posts* em questão.

Interface Uniforme – Derivada da restrição de mesmo nome no estilo arquitetural REST. Toda interação entre clientes e servidores deve se dar através de métodos HTTP, aproveitando suas funções para melhor entendimento do funcionamento do serviço.

Alguns métodos HTML e seus usos recomendados na ROA são apresentados na tabela 2.3.

Tabela 2.1 – Métodos HTML e seus usos recomendados na ROA

Método	Uso
HTTP GET	Retornar uma representação do recurso quando usado com seu URI.
HTTP PUT	Criar recursos, apontando para um novo URI, ou modificar um já existente apontando para seu URI.
HTTP POST	Criar um recurso, apontando para um URI já existente.
HTTP DELETE	Apagar um recurso quando usado com o seu URI.
HTTP HEAD	Retorna uma representação baseada apenas em metadados apontado para o URI de um recurso.
HTTP OPTIONS	Retorna quais métodos são executáveis em um recurso quando usado com seu URI.

Fonte: Produzido pelo autor com base em Richardson e Ruby (2007).

Essas características servem como guias gerais, não como regras invioláveis. Por exemplo, digamos que um recurso só deve ser acessados por um cliente em específico 100 vezes por dia, por razões de custo de operação. A variável contendo o número de vezes que um cliente acessa esse recurso não deveria ficar no cliente, pela possibilidade de perda ou falsificação dessa informação. Neste caso, ela poderia ficar armazenada no servidor, violando a característica *statelessness* da ROA.

2.4 Ferramentas utilizadas

Foi utilizado o *framework* Yii 2, já utilizado na implementação do núcleo da API UFRGS, da qual o serviço implementado neste trabalho faz parte, como um módulo. O Yii 2 é um *framework* PHP *open source* e gratuito (YII SOFTWARE LLC, 2017) que auxilia, entre outras características, a criação de web services REST. Seu uso foi decidido por facilitar a integração com a API UFRGS e já ser conhecida por membros do CPD.

Para testes básicos, foi utilizado o Postman, uma aplicação que fornece uma interface gráfica para requisições HTML, além de ter outras características como a possibilidade de salvar requisições e compartilhá-las com uma equipe (POSTDOT TECHNOLOGIES INC, 2017). As figuras ilustrando as requisições do capítulo 5 foram obtidas utilizando o Postman.

3 TRABALHOS RELACIONADOS

Neste capítulo são resumidamente descritos alguns trabalhos acadêmicos e comerciais relevantes a este trabalho. A seção 3.1 apresenta trabalhos acadêmicos, a seção 3.2 apresenta soluções comerciais, e a seção 3.3 oferece uma breve introdução à API UFRGS.

3.1 Trabalhos Acadêmicos

Não foram encontrados trabalhos acadêmicos que seguem a exata mesma linha deste trabalho, ou seja, o desenvolvimento de um *back-end* para uma aplicação de carona solidária. São apresentados trabalhos semelhantes focando-se em seus *web services*.

3.1.1 Desenvolvimento de um protótipo de solução colaborativa para o controle de listas de compras

Neste trabalho Abegg (2014) descreve a sua implementação de uma aplicação móvel com código aberto para gerência de listas de compras.

O *front-end* móvel da aplicação conecta-se a um servidor que roda um *web service* em estilo REST implementado com a linguagem Ruby acompanhada do *micro-framework* Sinatra.

O serviço implementado tem a única finalidade de atender ao *front-end*, e não implementa restrições às entradas enviadas pelo usuário ou algum método de autenticação, sendo a última característica uma função planejada.

3.1.2 Saci – Sistema de apoio à coleta de informações

O trabalho de Souza (2014) descreve a aplicação que o autor desenvolveu para resolver o problema da variabilidade da coleta de dados nas auditoria dos pontos de venda de uma empresa de marketing de atuação nacional.

A aplicação consiste de um *front-end* móvel na plataforma Android e um módulo *Web* que serve como o *back-end*. Este *back-end* consiste de um servidor rodando um *web service* REST escrito em Java, aliado ao GWT. Devido a decisões de projeto, o serviço foi desenvolvido em 3 camadas: A de comunicação, que utiliza GWT RPC para serializar os objetos a serem transferidos, a de serviços, que faz a ligação entre a camada de comunicação e a de acesso aos dados, e a camada de acesso aos dados, que utiliza o *framework* Spring para

gerenciar os ciclos de vida dos objetos que se comunicam com o banco de dados.

É utilizado o *framework* Spring Security para implementar a autenticação de usuários, e há regras de acesso ao banco de dados e regras de negócio implementadas na aplicação.

3.2 Soluções Comerciais

Existem alguns produtos que executam funções semelhantes à da aplicação Caronas, mas devido a problemas de privacidade dos dados seria impossível ter a integração da base que contém os dados dos alunos e os produtos.

3.2.1 Bynd e Caronetas

Estes dois produtos são semelhantes no seu funcionamento. Um usuário pode criar uma conta em um deles, e assim pode oferecer e ter acesso a oferecimentos de carona criados pela sua comunidade. As comunidades englobam empresas ou entidades específicas, e no caso do Caronetas, grupos mais abstratos como comunidades de redes sociais. Ambos são acessados tanto por clientes na *web* quanto por aplicações clientes móveis.

O Caronetas (CARONETAS, 2017) tem uso gratuito, e o Bynd (BYND SERVIÇOS DE TECNOLOGIA LTDA, 2017) oferece em seu site meios de contato para pedir orçamento para empresas.

3.2.2 BlaBlaCar

O BlaBlaCar (BLABLACAR, 2017) tem uma proposta um pouco diferente dos produtos citados na seção 3.2.1. Ele parece combinar a função de uma aplicação de carona solidária com a de uma aplicação de transporte como o Uber ou Cabify. Nele, os oferecimentos têm preços definidos para as vagas pelos motoristas. Ao contrário de serviços no estilo táxi como Uber e Cabify, não é possível fazer uma solicitação de carona a ser atendida por um motorista, apenas os motoristas podem criar os oferecimentos.

O BlaBlaCar não tem um sistema de comunidades isoladas como o Bynd e o Caronetas, todos os motoristas e solicitantes dividem o mesmo espaço. Assim como as aplicações descritas anteriormente, ele oferece clientes *web* e clientes em versões para aplicações móveis.

3.3 A API UFRGS

A API (*Application Programming Interface*) UFRGS é uma aplicação desenvolvida no CPD para atender com serviços *web* outras aplicações criadas no CPD. Ela é implementada com a linguagem PHP com o auxílio do *framework* Yii 2, e consiste em um "núcleo" onde são implementadas as funções de segurança e modelos básicos (como o Pessoa, que contém dados de alunos da UFRGS) e vários módulos com propósitos variados, como atender ao aplicativo móvel das bibliotecas UFRGS. O serviço Caronas descrito nesse trabalho é um módulo da API UFRGS desenvolvido para atender os clientes móveis da aplicação Caronas.

4 ARQUITETURA DO *BACK-END* DA APLICAÇÃO CARONAS

Neste capítulo é descrita a arquitetura do *web service* que serve como *back-end* para a aplicação Caronas. A seção 4.1 discute a arquitetura geral do sistema da aplicação Caronas e a posição do serviço nele. A seção 4.2 discute a arquitetura do serviço em si.

4.1 Arquitetura geral

Nesta seção discute-se a posição geral do trabalho dado o sistema existente já utilizado no Centro de Processamento de Dados da UFRGS.

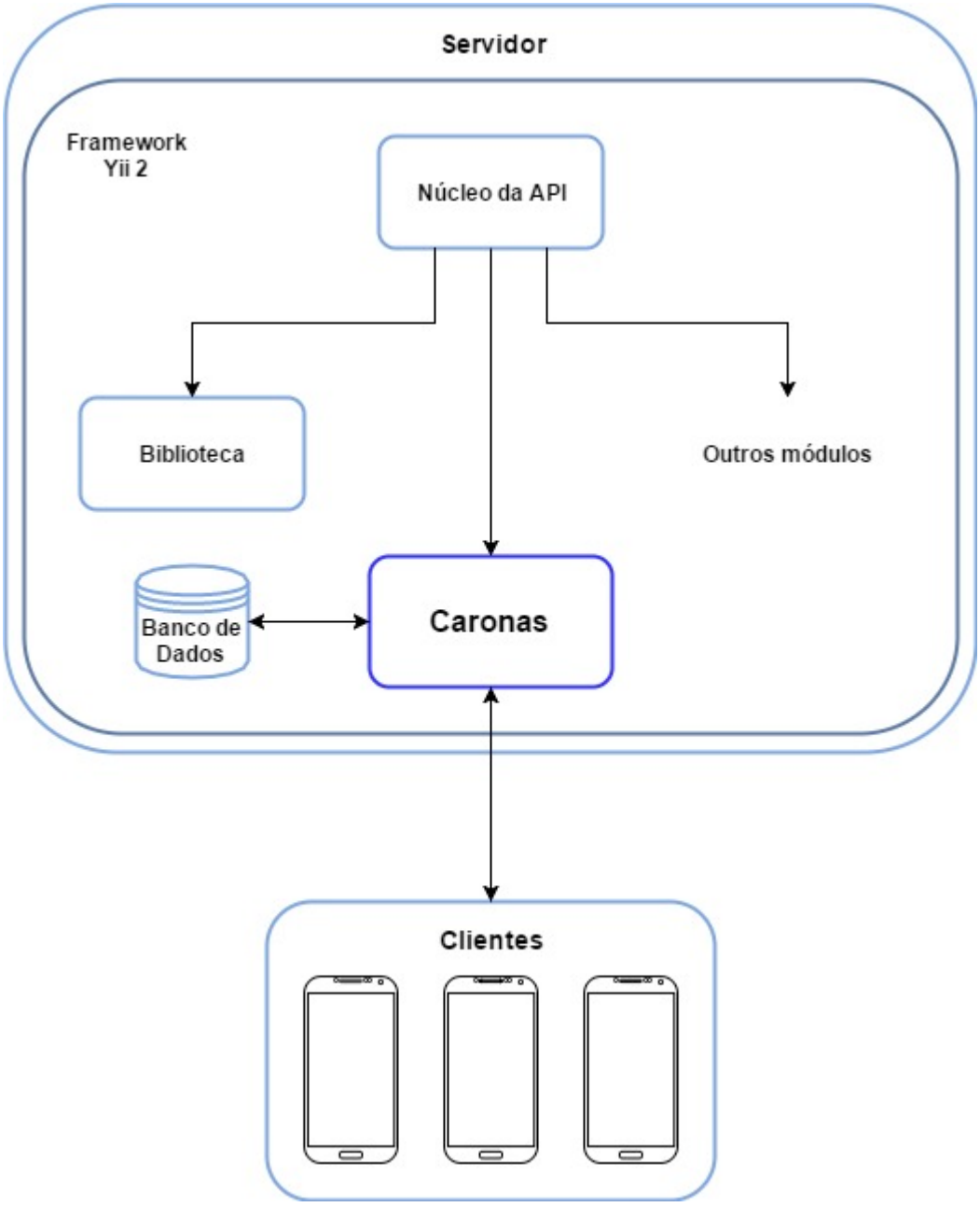
A figura 4.1 ilustra a posição do serviço desenvolvido como um módulo da API UFRGS, uma interface de programação de aplicações que serve a vários propósitos, como o serviço de *back-end* do aplicativo UFRGS Mobile e seu módulo de acesso a bancos de dados das bibliotecas.

Cada um desses módulos implementa um conjunto de modelos e controladores, de acordo com a arquitetura MVC proposta pelo framework Yii 2, que coordena o fluxo de dados entre as aplicações clientes e o banco de dados no servidor.

Este sistema já estava em atividade no início do desenvolvimento do módulo Caronas, e nele foram feitas apenas pequenas modificações para a integração do módulo desenvolvido neste trabalho.

Do módulo principal da API, aqui denominado “Núcleo da API”, o módulo caronas herda algumas características, como a implementação do protocolo de autenticação OAuth 2 e seu modelo de gerenciamento baseado em escopos, no qual o serviço tem seu próprio escopo, e modelos como o Pessoa, que contém dados pessoais de pessoas vinculadas à universidade.

Figura 4.1 – Posição do módulo Caronas considerando-se toda a API UFRGS



Fonte: Produção do autor.

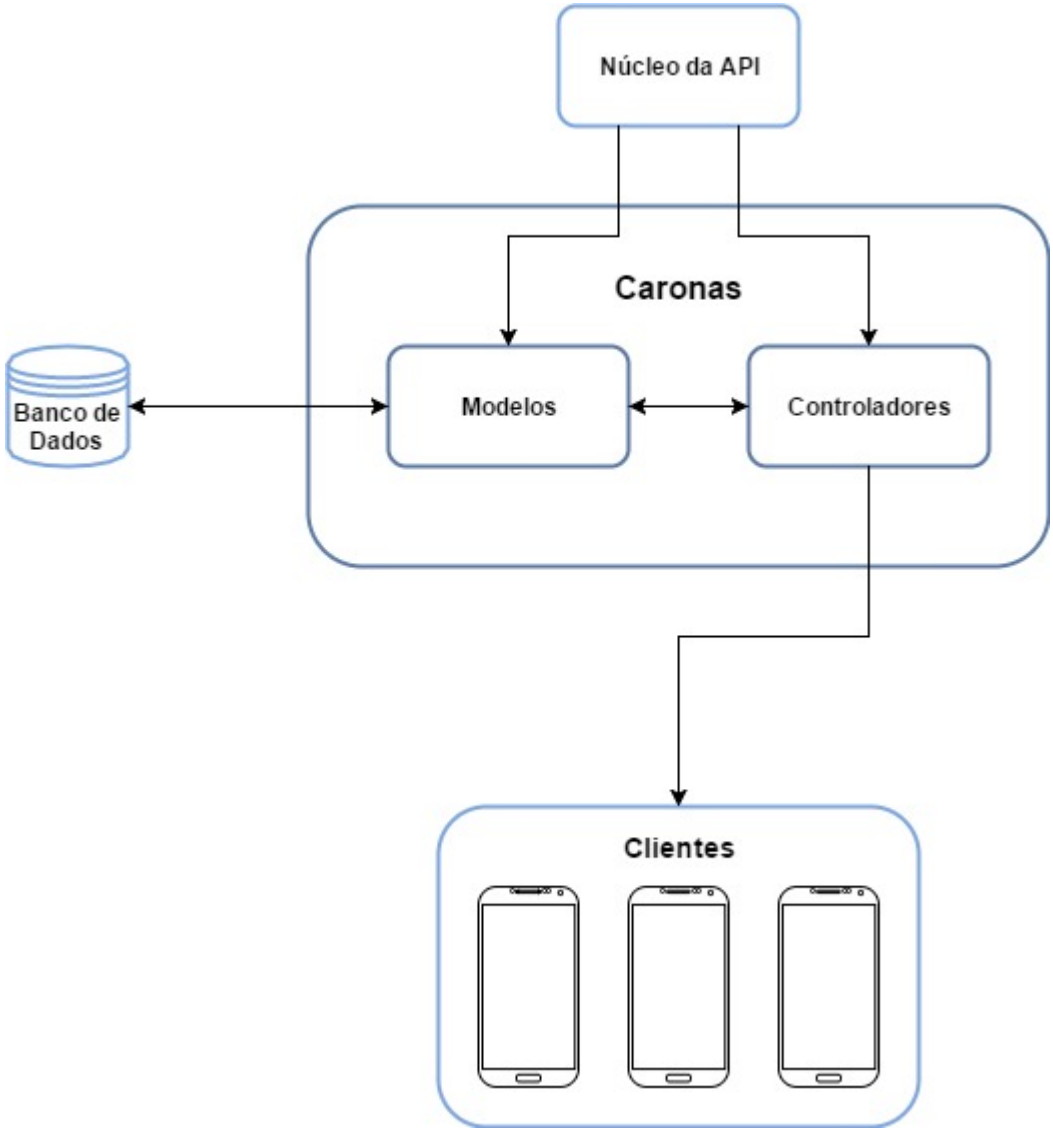
4.2 Arquitetura do módulo Caronas

Nesta seção é introduzida a arquitetura do módulo implementado neste trabalho, que também é nomeado Caronas, como a aplicação a que serve.

4.2.1 Arquitetura geral do módulo

A figura 4.2 ilustra a arquitetura do serviço.

Figura 4.2 – Arquitetura do módulo Caronas



Fonte: Produção do autor.

A arquitetura interna do módulo segue a arquitetura MVC proposta pelo framework Yii 2. Os modelos implementados incluem Campus, CarroPessoa, Cor, OferecimentoCarona, SolicitacaoCarona e UsuarioCaronas.

4.2.2 Modelos e descrições

Campus – Associa cada campus da Universidade a um código numérico.

CarroPessoa – Contém dados de um carro e identificação de seu proprietário.

Cor – Associa cores a códigos numéricos.

OferecimentoCarona – Contém dados sobre uma oferta de carona, incluindo a identificação do criador do oferecimento, o carro utilizado e o ponto de encontro dos usuários.

SolicitacaoCarona – Contém dados sobre uma solicitação de carona, incluindo o criador da solicitação, o oferecimento à qual esta solicitação se refere e uma possível mensagem ao criador do oferecimento.

UsuarioCaronas – Contém dados referentes ao uso de *front-ends* ligados ao módulo Caronas, como uma variável que indica o aceite ou não dos termos de uso. Dados do usuário em si, como nome e identificação numérica da UFRGS, são implementados no modelo Pessoa herdado do núcleo da API.

Cada um destes modelos possui uma ligação individual a uma parte de um banco de dados onde os dados em si são armazenados. Geralmente é seguida a correlação modelo/tabela de dados relacional, porém podem existir exceções.

Os modelos que recebem entradas diretamente do usuário, como CarroPessoa, OferecimentoCarona e SolicitacaoCarona, implementam regras de validação de dados com o intuito de evitar tentativas de exploração de vulnerabilidades do sistema por usuários maliciosos.

Cada modelo tem um controlador associado, que implementa ações, algumas acessíveis aos clientes através de chamadas HTTP.

4.2.3 Controladores e ações

Esta subseção descreve alguns controladores e suas ações disponíveis no módulo Caronas. Nota-se que as ações padronizadas fornecidas pelo Yii 2 foram alteradas ou removidas para melhor servir às necessidades da aplicação, e algumas ações personalizadas foram criadas. É necessário relembrar a herança do protocolo OAuth 2 do módulo núcleo da API UFRGS, pois todas as ações descritas necessitam de um *token* OAuth 2 fornecido pelo núcleo para serem executadas, tanto por motivos de segurança como para personalizar as saídas para cada usuário.

Nas subseções 4.2.3.1 a 4.2.3.5 a seguir, note que “usuário” se refere ao usuário ligado ao *token* fornecido durante a chamada.

Mais informações sobre os recursos são dadas no capítulo 5.

4.2.3.1 CarroController

Controlador associado ao modelo CarroPessoa a ao recurso /carros. Implementa as seguintes ações:

Índice – Retorna instâncias de CarroPessoa ligadas a modelos OferecimentoCarona disponíveis ao usuário.

Acessar Carros Usuário – Retorna instâncias de CarroPessoa criadas pelo usuário.

Exibir – Retorna uma instância específica de CarroPessoa dado um código identificador.

Criar – Cria uma nova instância de CarroPessoa , ligada à conta do usuário.

Atualizar – Altera os dados de uma instância de CarroPessoa.

Deletar – Invalida uma instância de CarroPesoa.

4.2.3.2 CorController

Controlador associado ao modelo Cor e ao recurso /cores. Implementa a seguinte ação:

Índice – Retorna todas as instâncias de Cor disponíveis.

4.2.3.3 OferecimentoController

Controlador associado ao modelo OferecimentoCarona e ao recurso /oferecimentos. Implementa as seguintes ações:

Índice – Retorna instâncias de OferecimentoCarona disponíveis ao usuário (válidas e não criadas pelo próprio).

Acessar Oferecimentos Usuário – Retorna instâncias válidas de OferecimentoCarona criados pelo usuário.

Acessar Histórico Usuário – Retorna instâncias de OferecimentoCarona das quais a data de partida é maior que a atual e que não foram invalidadas.

Acessar Cancelados Usuário – Retorna instâncias de OferecimentoCarona invalidadas pelo usuário.

Acessar Solicitações Oferecimento – Retorna instâncias de SolicitacaoCarona ligadas a uma instância de OferecimentoCarona identificado por um código.

Exibir – Retorna uma instância específica de OferecimentoCarona dado um código identificador.

Criar – Cria uma nova instância de OferecimentoCarona, identificando o usuário como motorista.

Atualizar – Altera os dados de uma instância de OferecimentoCarona.

Deletar – Invalida uma instância de OferecimentoCarona.

4.2.3.4 SolicitacaoController

Controlador associado ao modelo SolicitacaoCarona e ao recurso /solicitacoes. Implementa as seguintes ações:

Índice – Retorna instâncias de SolicitacaoCarona ligadas a instâncias de OferecimentoCarona criadas pelo usuário.

Acessar Solicitações Usuário – Retorna instâncias de SolicitacaoCarona criadas pelo usuário.

Exibir – Retorna uma instância específica de SolicitacaoCarona dado um código identificador.

Criar – Cria uma nova instância de SolicitacaoCarona, identificando o usuário como solicitante.

Atualizar – Altera os dados de uma instância de SolicitacaoCarona, incluindo o indicador de estado da mesma (Aceita, Cancelada, Pendente).

Deletar – Assinala uma instância de OferecimentoCarona como cancelada.

4.2.3.5 UsuarioController

Controlador associado ao modelo UsuarioCaronas e ao recurso /usuario. Implementa apenas uma ação:

Aceitar Termo – Salva o aceite dos termos de uso pelo usuário. Não é possível executar qualquer outra ação dos outros controladores do módulo usando o token do usuário até o mesmo aceitar os termos de uso.

5 *BACK-END* DO APLICATIVO CARONAS

Este capítulo mostra o processo de desenvolvimento do módulo Caronas. A seção 5.1 lista os requisitos do *back-end* e a seção 5.2 lista os passos do desenvolvimento do módulo, de acordo com a *Resource Oriented Architecture*.

5.1 Requisitos do back-end

Anteriormente, no capítulo 1, foram definidos como requisitos segurança, performance e facilidade de manutenção do código. Esses pontos, porém, não definem qual a funcionalidade prática esperada do *back-end*.

Para determinar tais funcionalidades, foi examinado o fluxo planejado do aplicativo caronas, a definição das tabelas do banco de dados que atenderá ao aplicativo, e o núcleo da API UFRGS já implementado. Todos foram definidos antes do projeto e implementação do *back-end* do Caronas, pelo pessoal do CPD.

A figura 5.1 ilustra o fluxo do *front-end* móvel da aplicação Caronas. Indicada na figura estão as seguintes telas:

A – Tela de *login*. Aqui a aplicação gera o *token* de uso do *back-end*.

B – Mapa. Depois de obter o *token* ou usar um já existente, a aplicação acessa os dados referentes a oferecimentos existentes e os exibe no mapa.

C – Lista de carros pertencentes ao usuário.

D – Tela de edição ou criação de nova entrada de carro.

E – Oferecimentos criados pelo usuário.

F – Tela de criação de oferecimento de carona.

G – Lista de oferecimentos disponíveis.

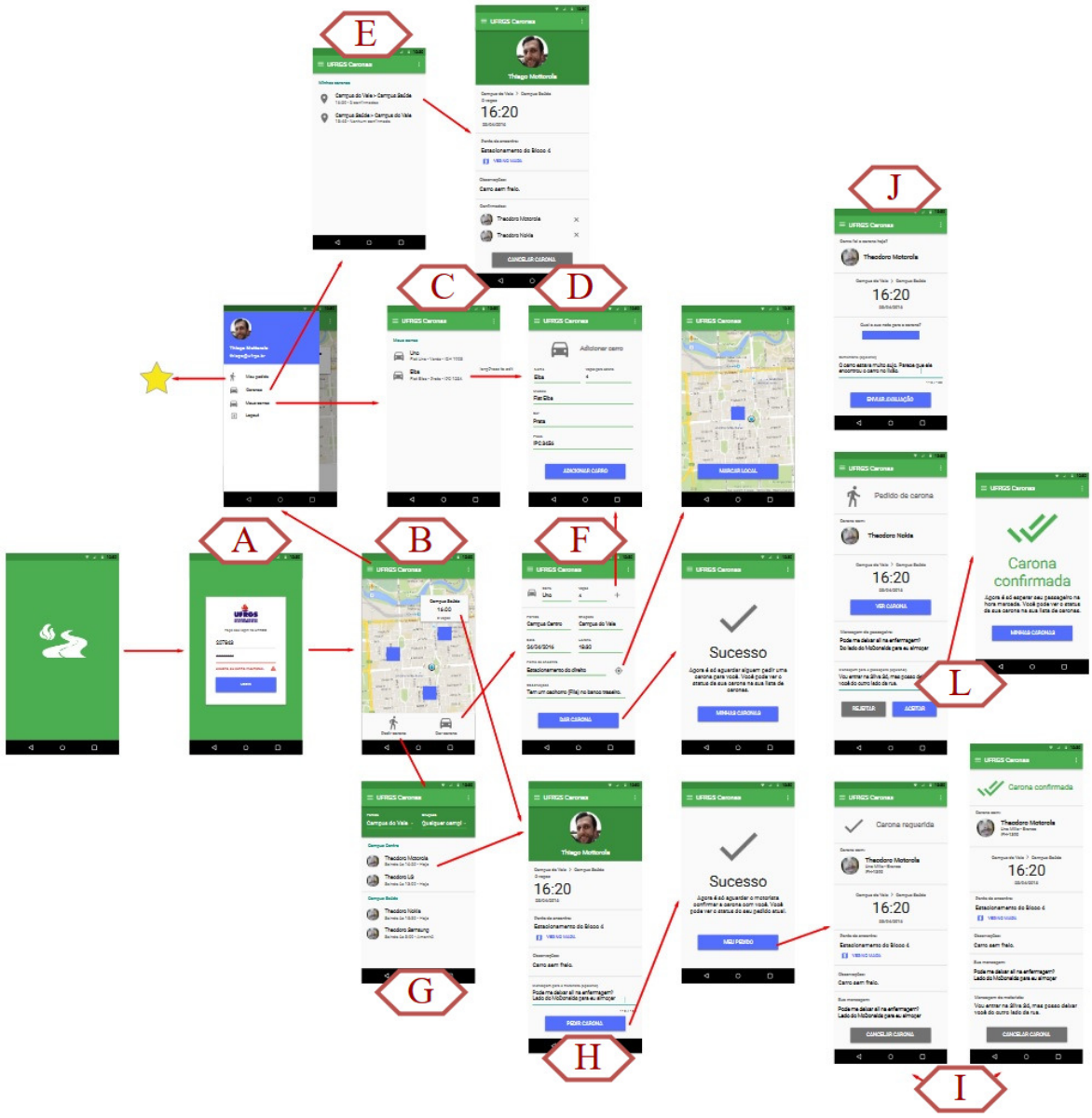
H – Tela de criação de solicitação.

I – Telas exibindo uma solicitação criada e a mesma solicitação após o aceite dela pelo motorista.

J – Tela de criação de avaliação da carona pelo solicitante.

L – Tela de aceita ou recusa de uma solicitação de carona.

Figura 5.1 – Fluxo do cliente mobile da aplicação Caronas



Fonte: Equipe do CPD, anotações pelo autor

A figura 5.2 ilustra a estrutura do banco de dados designado para armazenar os dados referentes ao Caronas.

Nela, estão descritas as seguintes tabelas:

USUARIO – Contém dados referentes ao aceite dos termos da aplicação pelos usuários.

OFERECIMENTOCARONA – Contém dados referentes a oferecimentos de carona.

SOLICITACAOCARONA – Contém dados referentes a solicitações de carona.

CARROPESSOA – Contém dados referentes a carros e a que usuário pertencem.

CAMPUS – Associa cada campus a um código numérico.

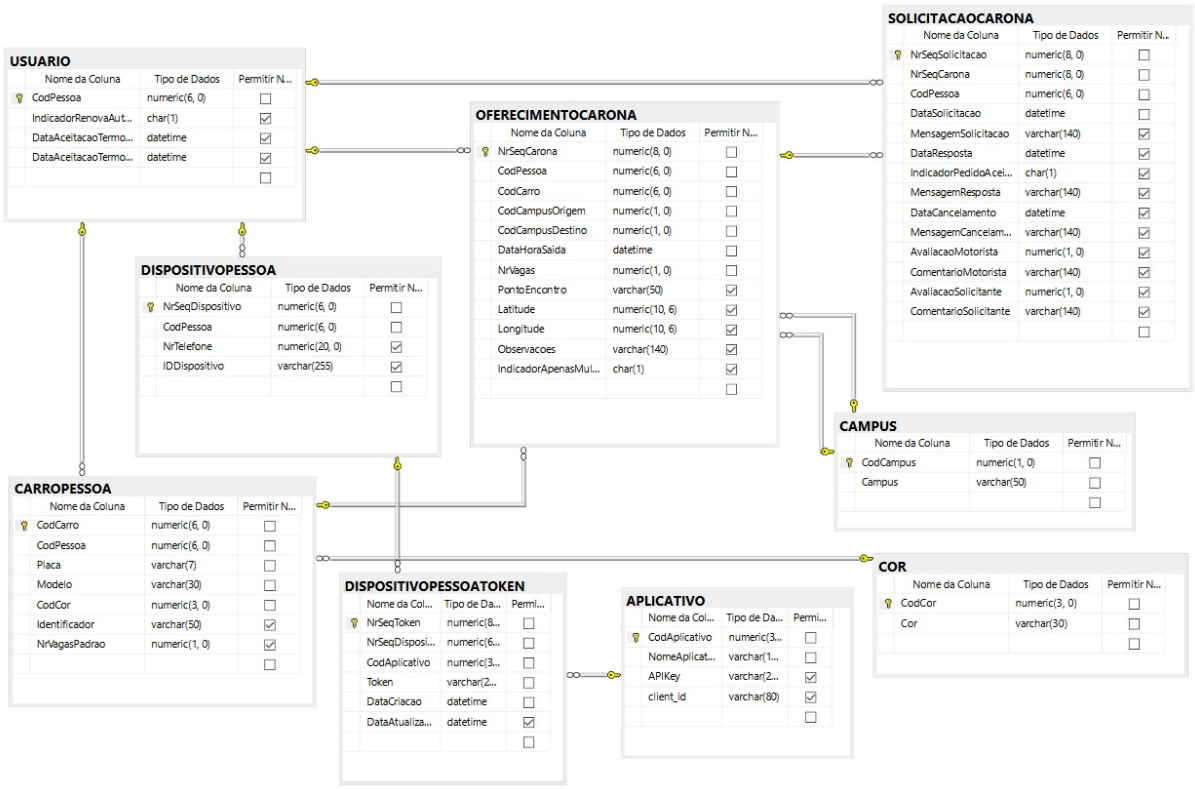
COR – Associa cores a códigos numéricos.

DISPOSITIVOPESSOA – Associa dispositivos a usuários.

DISPOSITIVOPESSOATOKEN – Associa relações dispositivo-usuário à *tokens* de autorização.

APLICATIVO – Contém dados de identificação de aplicações clientes.

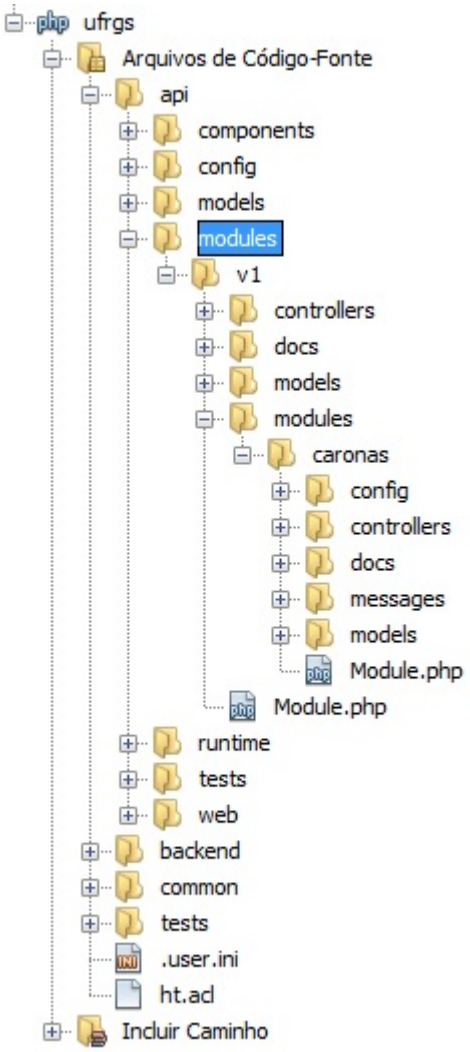
Figura 5.2 – Estrutura do banco de dados da aplicação Caronas.



Fonte: Equipe do CPD

Finalmente, a figura 5.1.3 ilustra a estrutura do código que implementa o núcleo da API UFRGS e o módulo Caronas.

Figura 5.3 – Estrutura do código do Núcleo da API UFRGS e do módulo Caronas



Fonte: Equipe do CPD, com módulo Caronas adicionado pelo autor

Tendo essas informações, utilizou-se o processo descrito por Richardson e Ruby (2007) para projetar o *web service*, de acordo com a *Resource Oriented Architecture* definida pelos mesmos. A aplicação da ROA é descrita na seção 5.2.

5.2 Aplicação da ROA

O processo de desenvolvimento da ROA é aplicado resolvendo-se as questões apresentadas nas subseções 5.2.1 a 5.2.9.

5.2.1 Descobrir o conjunto de dados

Examinando o fluxo do cliente Caronas e as tabelas do banco de dados, compreende-se que os dados a serem gerenciados pelo *web service* podem ser agrupados da seguinte forma: contas de usuário (nome e senha para o login no aplicativo), relação pessoa-carro, relação pessoa-dispositivo, relação *token*-dispositivo, dados de aplicativo, oferecimentos de carona, solicitações de carona, relação cor-código e relação campus-código.

5.2.2 Dividir o conjunto de dados em recursos

Usuários, contendo identificação do usuário e *flags* indicando a aceitação dos termos de uso da aplicação.

Carros, contendo dados de identificação de carros e seus proprietários.

Oferecimentos, contendo dados referentes a um oferecimento de carona.

Solicitações, contendo dados referentes a solicitações de carona.

Cores, contendo identificação de cores em códigos numéricos e texto.

Os seguintes não serão expostos como recursos, pois isso não seria necessário para o funcionamento da aplicação e/ou por motivos de segurança:

Dispositivos, contendo identificação de dispositivos, seus donos e número de telefone, quando aplicável.

Tokens, contendo informações referentes a um *token* de autenticação ligado a um dispositivo.

Aplicativos, contendo dados sobre aplicativos de *front-end* que acessam o *back-end* Caronas.

Campi, contendo identificação dos campi da UFRGS em códigos numéricos e texto.

5.2.3 Nomear os recursos

Sendo o nome de um recurso sua URI, como definido por Richardson e Ruby (2007), adota-se o URI relativo /caronas/ como raiz do *web service*.

Os objetos então são, inicialmente:

/caronas/usuarios

/caronas/carros

/caronas/oferecimentos

/caronas/solicitacoes

/caronas/cores

Durante o desenvolvimento do design do *back-end*, para atender o requisito de interface uniforme da REST e implementar funções necessárias à aplicação final, sub-recursos serão criados para evitar a sobrecarga do método HTML POST.

5.2.4 Expor um subconjunto da interface uniforme

Nesta subseção analisa-se cada recurso e seu uso na aplicação e define-se quais métodos serão expostos para aplicações clientes. Também são nomeados os sub-recursos criados para evitar a sobrecarga do método POST com o propósito de acessar outras funções necessárias para a aplicação final. Nota-se que devido ao uso do framework Yii 2, que tem o uso dos métodos HTML padronizados, há divergência do uso sugerido por Richardson e Ruby (2007) no caso do método HTML POST, que é usado como o autor sugere o uso do método HTML PUT. Métodos HTML HEAD e HTML OPTIONS não são expostos nesta versão do *back-end*, sendo sua implementação adiada até a possível liberação do uso dele ao público.

5.2.4.1 /usuarios

Único método exposto é o HTML POST, usado para um usuário aceitar os termos de uso da aplicação, portanto modificando a *flag* no banco de dados. O login do usuário e o armazenamento de sua identificação são tratados no núcleo da API UFRGS, do qual o *back-end* caronas é um módulo. Mais informações sobre o núcleo da API são dadas no capítulo 4.

5.2.4.2 /carros

É preciso introduzir dados sobre carros pelos usuários, então é necessário expor o método HTML POST para essa função. Como existirão várias entradas, para cada carro introduzido é definido um sub-recurso identificado pelo campo CodCarro do banco de dados, sendo então nomeados /caronas/carros/{CodCarro}. O método HTML GET é exposto para retornar entradas de carros disponíveis para solicitações de carona do usuário, uma função não planejada para o *front-end* em desenvolvimento, porém com possível aplicação futura.

5.2.4.3 /carros/{CodCarro}

O método HTML GET é exposto para retornar a entrada identificada por CodCarro. PUT e PATCH são expostos para modificar os dados da entrada, e DELETE é exposto para removê-la.

5.2.4.4 /carros/usuario

Expõe o método HTML GET, que retorna entradas de carros que sejam relacionadas (ou seja, pertencentes) ao usuário associado ao *token* de autorização.

5.2.4.5 /oferecimentos

É necessário que os usuários criem os oferecimentos para popular o banco, sendo então o método HTML POST exposto para esse propósito. Cada oferecimento criado gera um novo sub-recurso identificado pelo campo NrSeqCarona, sendo então esses sub-recursos nomeados /oferecimentos/{NrSeqCarona}. O método HTML GET é exposto para retornar oferecimentos disponíveis para criação de solicitações pelo usuário.

5.2.4.6 /oferecimentos/{NrSeqCarona}

É exposto o método HTML GET para retornar a entrada de oferecimento identificada por {NrSeqCarona}. Os métodos HTML PATCH e PUT são expostos para atualizar os dados desta entrada. O método HTML DELETE é exposto para apagar (cancelar) o oferecimento.

5.2.4.7 /oferecimentos/{NrSeqCarona}/solicitacoes

É exposto o método HTML GET para retornar entradas de solicitações relacionadas a uma entrada de oferecimento (ou seja, solicitações para uma vaga no oferecimento identificado por NrSeqCarona).

5.2.4.8 /oferecimentos/usuario

Expõe o método HTML GET, que retorna entradas de oferecimentos que foram criados pelo usuário que ainda estejam válidos.

5.2.4.9 /oferecimentos/usuario/historico

Expõe o método HTML GET, que retorna entradas de oferecimentos cuja data de partida já tenha passado e que foram criados pelo usuário.

5.2.4.10 /oferecimentos/usuario/cancelados

Expõe o método HTML GET, que retorna oferecimentos criados pelo usuário e cancelados.

5.2.4.11 /solicitacoes

É necessária a criação de novas solicitações pelos usuários, portanto o método HTML POST é exposto para este propósito. Cada solicitação criada é identificada pelo campo NrSeqSolicitacao, então são criados novos sub-recursos nomeados /solicitacoes/ {NrSeqCarona}. O método HTML GET é exposto para retornar solicitações relacionadas a oferecimentos criados pelo usuário.

5.2.4.12 /solicitacoes/{NrSeqSolicitacao}

O método HTML GET é exposto para retornar os dados da entrada de solicitação identificada por NrSeqSolicitacao. Os métodos PATCH e PUT são expostos para modificar dados de uma solicitação criada previamente (incluindo a *flag* de aceite da solicitação). O método HTML DELETE é exposto para apagar (cancelar) uma solicitação.

5.2.4.13 /solicitacoes/usuario

O método HTML GET é exposto para retornar entradas de solicitação criadas pelo usuário.

5.2.4.14 /cores

O método HTML GET é exposto para retornar as relações código-texto da tabela COR do banco de dados.

5.2.5 Projetar as representações que serão aceitas vindas do cliente

Todos os recursos com o método HTML POST ou HTML PUT/PATCH expostos precisam de uma representação que será usada pelos clientes para enviar informações ao banco de dados. Todos os recursos aceitam representações no formato *form* HTML (*media type* application/x-www-form-urlencoded), e seus campos são descritos a seguir. Nota-se que os nomes dos campos são ligeiramente diferentes dos campos do banco de dados. Isso é para que os formulários estejam na mesma formatação dos outros formulários na API UFRGS, usando somente letras minúsculas, e também porque a implementação deste mapeamento entre os nomes facilita possíveis modificações futuras feitas no banco de dados, já que desacopla os nomes dos campos do *back-end* dos nomes dos campos do banco.

Todas as chamadas ao *back-end* requerem um formulário HTML com o campo *authorization* preenchido com um *token* de utilização do Caronas, característica herdada do núcleo da API UFRGS.

O framework Yii 2 disponibiliza o uso de um verificador de entradas antes do armazenamento destas no banco de dados. Isso permite a implementação das restrições descritas nas subseções 5.2.5.1 a 5.2.5.8.

5.2.5.1 /usuarios

Recebe um formulário vazio do usuário. Como o resultado da modificação é sempre o mesmo, não sendo possível cancelar a aceitação dos termos de uso, a única informação adicional necessária é o número de identificação do usuário, que já é fornecido pelo seu *token* de autorização de uso do *back-end.*

5.2.5.2 /carros

Aqui são descritos todos os campos associados ao recurso /carros combinado com o método HTTP POST.

placa

Obrigatório.

String de no máximo 7 caracteres;

Não pode existir mais de um registro com a mesma combinação de codigodono e placa. O campo codigodono é descrito na subseção 5.2.6.1, e é preenchido automaticamente a partir do *token* de autenticação.

modelo

Obrigatório.

String de no máximo 30 caracteres.

codigocor

Obrigatório.

Integer de valor máximo 999;

Deve existir um registro com o código enviado no banco de dados.

identificador

Opcional.

String de no máximo 50 caracteres.

vagaspadrao

Obrigatório.

Integer de valor mínimo 1 e máximo 9.

5.2.5.3 /carros/{CodCarro}

Aqui são descritos todos os campos associados aos recursos /carros/{CodCarro} combinado com os métodos HTTP PUT ou PATCH.

modelo

Opcional.

String de no máximo 30 caracteres;

codigocor

Opcional.

Integer de valor máximo 999;

Deve existir um registro com o codigocor enviado no banco de dados.

identificador

Opcional.

String de no máximo 50 caracteres.

vagaspadrao

Opcional.

Integer de valor mínimo 1 e máximo 9.

5.2.5.4 /oferecimentos

Aqui são descritos todos os campos associados ao recurso /oferecimentos combinado com o método HTTP POST.

codigocarro

Obrigatório.

Integer de valor máximo 999999;

Deve existir um registro com o codigocarro enviado no banco de dados;

No banco de dados, a entrada de dados sobre o carro deve estar vinculada ao número

do cartão UFRGS do criador do oferecimento.

codigocampusorigem

Obrigatório.

Integer de valor máximo 9;

Deve existir um registro com o código enviado no banco de dados.

codigocampusdestino

Obrigatório.

Integer de valor máximo 9;

Deve existir um registro com o código enviado no banco de dados.

horasaida

Obrigatório.

Data no formato php:Y-m-d H:i:s;

Data deve ser maior do que a data atual;

Não pode existir mais de um registro com a mesma combinação de codigomotorista e horasaida. O campo codigomotorista é descrito na subseção 5.2.6.4.

vagas

Obrigatório.

Integer de valor mínimo 1 e máximo 9.

pontoencontro

Opcional.

String de no máximo 50 caracteres.

latitude

Opcional.

Número incluído na expressão regular /^[–+]?[0-9]{1,10}\.?[0-9]{0,6}$/.

longitude

Opcional.

Número incluído na expressão regular /^[–+]?[0-9]{1,10}\.?[0-9]{0,6}$/.

observacoes

Opcional.

String de no máximo 140 caracteres.

apenasmulheres

Opcional.

String de no máximo 1 caracter.

Caractere pode ser apenas S ou N.

Apenas usuários identificados como mulheres podem criar oferecimentos com o caractere com o valor S.

5.2.5.5 /oferecimentos/{NrSeqCarona}

Aqui são descritos todos os campos associados aos recursos /*oferecimentos/ {NrSeqCarona}* combinados com os métodos HTTP PUT ou PATCH.

observacoes

Opcional.

String de no máximo 140 caracteres.

5.2.5.6 /solicitacoes

Aqui são descritos todos os campos associados ao recurso /*solicitacoes* combinado com o método HTTP POST.

codigooferecimento

Obrigatório.

Integer de valor máximo 99999999;

Deve existir um registro com o codigooferecimento enviado no banco de dados

O oferecimento de carona identificado por codigooferecimento deve ter vagas disponíveis;

Não deve existir mais de uma solicitação válida vinculada ao codigooferecimento fornecido criada pelo usuário proprietário do *token*.

5.2.5.7 /solicitacoes/{NrSeqSolicitacao}

Aqui são descritos todos os campos associados aos recursos */solicitacoes/ {NrSeqCarona}* combinados com os métodos HTTP PUT ou PATCH.

Este é o recurso mais complexo em relação às suas restrições. Ele trabalha com quatro casos de validação:

Caso 1: Aceitar.

Se o campo indicadorpedido for preenchido no formulário, assume-se este caso.

Acessível apenas ao criador do oferecimento de carona vinculado à solicitação.

Campos aceitos:

indicadorpedido

Obrigatório.

String de no máximo 1 caractere;

Não pode ter um valor diferente de S.

mensagemresposta

Opcional.

String de no máximo 140 caracteres.

Caso 2: Avaliar motorista.

Se o campo avaliacaomotorista for preenchido no formulário, assume-se este caso.

Acessível apenas ao criador da solicitação de carona.

Campos aceitos:

avaliacaomotorista

Obrigatório.

Integer de valor mínimo 1 e máximo 5.

comentariomotorista

Opcional.

String de no máximo 140 caracteres.

Caso 3: Avaliar solicitante.

Se o campo avaliacaosolicitante for preenchido no formulário, assume-se este caso.

Acessível apenas ao criador do oferecimento de carona vinculado à solicitação.

Campos aceitos:

avaliacaosolicitante

Obrigatório.

Integer de valor mínimo 1 e máximo 5.

comentariosolicitante

Opcional.

String de no máximo 140 caracteres.

Caso 4: Atualizar solicitação.

Caso nenhuma das condições para assumir os casos anteriores ocorra, assume-se este caso.

Acessível apenas ao criador da solicitação de carona.

Campos aceitos:

mensagemsolicitacao

Opcional.

String de no máximo 140 caracteres.

5.2.6 Projetar as representações servidas ao usuário

O núcleo da API UFRGS já define um modelo básico de como o serviço deve responder, uma mensagem no formato JSON com os pares pré-definidos “success”, que pode ser emparelhado com *true* ou *false*, “code”, que é emparelhado com o código HTML da resposta, “message”, que é emparelhado com uma mensagem legível relacionada ao código HTML e “data”, que é emparelhado com um valor adicional relacionado à requisição. Este modelo é ilustrado na figura 5.4.

Figura 5.4 – Formato de resposta padrão da API UFRGS

```
"{
    "success": true,
    "code": 200,
    "message": "A requisição foi atendida.",
    "data": {
    }
}"
```

Fonte: Equipe do CPD

Todas as respostas do serviço seguem os padrões ilustrados nas figuras 5.2.6.2, 5.2.6.3 e 5.2.6.4.

A figura 5.5 ilustra uma resposta com sucesso que retorna uma lista de objetos, como uma lista de carros. O campo “data” está ilustrando como os objetos são retornados em JSON. O campo “_links” contém *links* (URIs) para as páginas vizinhas da página recebida, a primeira e a última página, e o link da própria página. O campo “_meta” contém dados sobre a resposta em si, incluindo o número de objetos, o número de páginas, a página presente e o número de objetos por página.

A figura 5.6 ilustra o modelo de uma resposta com o código de erro 422, significando uma entrada inválida. Isso significa que a requisição falhou porque os campos enviados não atenderam as restrições descritas na subseção 5.2.5. O campo “data” retorna preenchido com um envelope denominado “erros” que contém pares campo-descrição do erro na entrada. Este modelo de resposta não existia antes da implementação do módulo Caronas, sendo implementado para atender às suas necessidades.

No caso de um erro a que não seja atribuído o código 422, é retornado uma mensagem no modelo ilustrado na figura 5.7. O campo “data” é pareado com o valor *null*. O campo “message” contém uma descrição do erro.

Figura 5.5 – Modelo de resposta contendo uma lista de objetos.

```
"{
    "success": true,
    "code": 200,
    "message": "A requisição foi atendida.",
    "data": [
        {},
        {},
        ...,
        {}
    ],
    "_links": {
        "self": {
            "href": "https://api.ufrgs.br/v1/{modulo}/{endpoint}?page=5"
        },
        "first": {
            "href": "https://api.ufrgs.br/v1/{modulo}/{endpoint}?page=1"
        },
        "prev": {
            "href": "https://api.ufrgs.br/v1/{modulo}/{endpoint}?page=4"
        },
        "next": {
            "href": "https://api.ufrgs.br/v1/{modulo}/{endpoint}?page=6"
        },
        "last": {
            "href": "https://api.ufrgs.br/v1/{modulo}/{endpoint}?page=20"
        }
    },
    "_meta": {
        "totalCount": "400",
        "pageCount": "20",
        "currentPage": "5",
        "perPage": "20"
    }
}"
```

Fonte: Equipe do CPD

Figura 5.6 – Modelo de resposta com código de erro 422

```
"{
    "success": false,
    "code": 422,
    "message": "Entrada inválida.",
    "data": {
        "erros": {
            "{nome do campo}": [
                "{mensagem de erro}",
                "{mensagem de erro}",
                ...,
                "{mensagem de erro}"
            ],
            "{nome do campo}": [
                "{mensagem de erro}"
            ]
            ...,
            "{nome do campo}": [
                "{mensagem de erro}",
                "{mensagem de erro}"
            ]
        }
    }
}"
```

Fonte: Equipe do CPD, novo modelo implementado para atender o Caronas

Figura 5.7 – Modelo de resposta com código de erro diferente de 422

```
"{
    "success": false,
    "code": 403,
    "message": "The request requires higher privileges than provided by the access token",
    "data": null
}"
```

Fonte: Equipe do CPD

As subseções 5.2.6.1 a 5.2.6.11 descrevem as respostas dadas pelo *web service* quando o método HTML GET é invocado em seus recursos.

5.2.6.1 /carros

Lista os dados de todos os carros vinculados a oferecimentos ativos que ainda possuem vagas e não foram criados pelo usuário proprietário do *token* de autenticação. O campo "data" da resposta retorna um vetor de estruturas *carros* com cada estrutura contendo os seguintes campos:

codigo

Número de identificação da entrada.

codigodono

Número do cartão UFRGS do criador da entrada.

placa

Número da placa do carro descrito na entrada.

modelo

Modelo do carro descrito na entrada.

codigocor

Código da cor do carro descrito na entrada.

identificador

Nome opcional para identificação do carro.

vagaspadrao

Número padrão de vagas quando um oferecimento é criado ligado a este carro.

5.2.6.2 /carros/usuario

Lista os dados de todos os carros criados pelo usuário proprietário do *token* usado na autenticação. O campo "data" da resposta contém um vetor *carros* contendo os mesmos campos que a resposta do HTML GET executado no recurso /carros descrita na subseção 5.2.6.1.

5.2.6.3 /carros/{codigo}

Retorna os dados de um carro identificado por *codigo*. O campo "data" da resposta retorna uma estrutura contendo os mesmos campos que as estruturas no vetor *carros* enviadas pela execução do HTML GET no recurso /carros descrita na subseção 5.2.6.1.

5.2.6.4 /oferecimentos

Lista os dados de todos os oferecimentos de carona ativos que ainda possuem vagas e não foram criados pelo usuário proprietário do *token* de autenticação. O campo "data" da resposta retorna um vetor de estruturas *oferecimentos* com cada estrutura contendo os seguintes campos:

codigo

Número de identificação da entrada.

codigomotorista

Número do cartão UFRGS do criador do oferecimento.

codigocarro

Número de identificação da entrada do carro usado no oferecimento.

codigocampusorigem

Número de identificação do campus onde se iniciará a viagem.

codigocampusdestino

Número de identificação do campus destino da viagem.

horasaida

Data e hora do início planejado da viagem.

vagas

Vagas disponíveis neste oferecimento.

pontoencontro

Descrição textual opcional do ponto onde o motorista e os solicitantes se encontrarão.

latitude

Latitude opcionalmente fornecida do ponto de encontro.

longitude

Longitude opcionalmente fornecida do ponto de encontro.

observacoes

Informações adicionais opcionais sobre o oferecimento.

apenasmulheres

Indicador que define se este oferecimento está disponível apenas para mulheres.

Este endpoint envia os seguintes vetores extras incluídos nas estruturas do vetor *oferecimentos* quando requisitados pelo parâmetro expand:

motorista

Dados sobre o criador do oferecimento de carona. Inclui os seguintes campos:

nome

Nome do criador do oferecimento.

sexo

Sexo do criador do oferecimento.

foto

String que representa a foto do motorista.

carro

Dados sobre o carro usado no oferecimento de carona. Inclui os seguintes campos:

placa

Número da placa do carro usado no oferecimento.

modelo

Modelo do carro usado no oferecimento.

5.2.6.5 /oferecimentos/usuario

Lista os dados dos oferecimentos de carona ativos e válidos criados pelo usuário proprietário do *token* de autorização. O campo "data" da resposta contém um vetor *oferecimentos* com a mesma estrutura que o enviado na execução do método HTML GET no recurso /oferecimentos descrito na subseção 5.2.6.4.

5.2.6.6 /oferecimentos/usuario/historico

Lista os dados de todos os oferecimentos de carona válidos não ativos criados pelo usuário proprietário do *token* de autorização. O campo "data" da resposta contém um vetor *oferecimentos* com a mesma estrutura que o enviado na execução do método HTML GET no recurso /oferecimentos descrito na subseção 5.2.6.4.

5.2.6.7 /oferecimentos/usuario/cancelados

Lista os dados de todos os oferecimentos de carona invalidados criados pelo usuário proprietário do *token* de autorização. O campo "data" da resposta contém um vetor *oferecimentos* com a mesma estrutura que o enviado na execução do método HTML GET no recurso /oferecimentos descrito na subseção 5.2.6.4.

5.2.6.8 /oferecimentos/{codigo}

Retorna os dados de um oferecimento de carona identificado por *codigo.* O campo "data" da resposta retorna uma estrutura contendo os mesmos campos que as estruturas no

vetor *oferecimentos* enviadas pela execução do HTML GET no recurso /oferecimentos descrita na subseção 5.2.6.4.

5.2.6.8 /oferecimentos/{codigo}/solicitacoes

Lista os dados de todas as solicitações de carona vinculadas ao oferecimento identificado por *codigo*. O campo "data" da resposta retorna um vetor de estruturas *solicitacoes* com cada estrutura contendo os seguintes campos:

codigo

Número de identificação da entrada.

codigooferecimento

Número de identificação do oferecimento referente à solicitação.

codigocriador

Número do cartão UFRGS do criador da solicitação.

datacriacao

Data e hora da criação da solicitação.

mensagemsolicitacao

Texto opcional enviado pelo criador da solicitação ao criador do oferecimento relacionado.

dataresposta

Data e hora do recebimento da resposta à solicitação.

indicadorpedido

Indicador que define o status da solicitação (pendente, aceita ou cancelada).

mensagemresposta

Texto opcional enviado ao solicitante durante o aceite da solicitação pelo criador do oferecimento.

datacancelamento

Data e hora do cancelamento da solicitação.

mensagemcancelamento

Texto opcional enviado ao solicitante ou ao criador do oferecimento durante o cancelamento.

avaliacaomotorista

Número que avalia o criador do oferecimento atribuído pelo solicitante.

comentariomotorista

Texto opcional enviado ao criador do oferecimento pelo solicitante ao final da viagem.

avaliacaosolicitante

Número que avalia o solicitante atribuído pelo criador do oferecimento.

comentariosolicitante

Texto opcional enviado ao solicitante pelo criador do oferecimento ao final da viagem.

5.2.6.9 /solicitacoes

Lista os dados de todas as solicitações de carona ativas vinculadas a oferecimentos criados pelo usuário proprietário do *token* de autorização. O campo "data" da resposta contém um vetor *solicitacoes* com a mesma estrutura que o enviado na execução do método HTML GET no recurso */oferecimentos/{codigo}/solicitacoes* descrito na subseção 5.2.6.8.

5.2.6.10 /solicitacoes/usuario

Lista os dados de todos as solicitações de carona criadas pelo usuário proprietário do *token* de autorização. O campo "data" da resposta contém um vetor *solicitacoes* com a mesma estrutura que o enviado na execução do método HTML GET no recurso */oferecimentos/{codigo}/solicitacoes* descrito na subseção 5.2.6.8.

5.2.6.11 /solicitacoes/{codigo}

Retorna os dados de uma solicitação de carona identificada por *codigo.* O campo "data" da resposta retorna uma estrutura contendo os mesmos campos que as estruturas no

vetor *solicitacoes* enviadas pela execução do HTML GET no recurso */oferecimentos/{codigo}/solicitacoes* descrito na subseção 5.2.6.8.

5.2.7 Ligar os recursos entre si

Richardson e Ruby (2007) recomendam a adição de URIs de recursos relacionados às respostas dadas pelo *web service*, conectando-os entre si e satisfazendo a característica HATEOAS dos *web services* REST. No ambiente de produção isso não se tornou necessário, devido a uma característica do Yii 2 que permite o retorno de campos adicionais adicionando-os como valor na variável *expand* da URI (como exemplificado na subseção 5.2.6.4), e devido à necessidade de reduzir o fluxo de dados entre servidor e cliente, a adição de URIs específicos não foi introduzida, ficando como função adicionável no lançamento ao público do serviço.

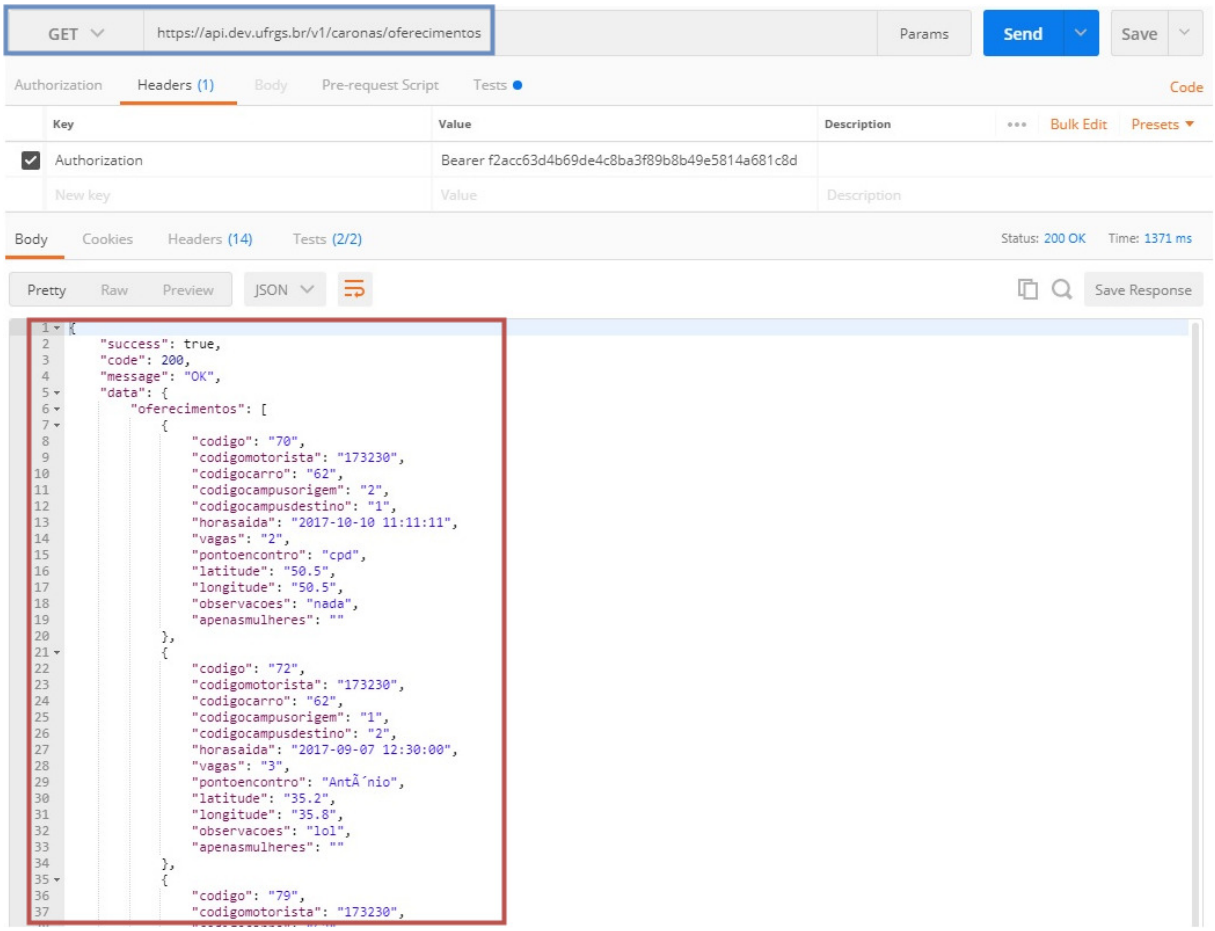
Uma característica semelhante foi implementada devido à paginação do conteúdo das respostas, contendo URIs das páginas adjacentes, inicial e final em cada resposta.

5.2.8 O que deve acontecer?

Assumindo que o usuário já tenha obtido um *token* de autenticação e o cliente o tenha anexado às requisições, os casos seguintes devem ocorrer. Nas figuras exemplo, os campos azuis destacam o recurso e método HTTP, os campos vermelhos as respostas e os verdes as informações passadas pelo formulário HTTP.

Caso o cliente execute um comando HTTP GET, ele deve receber uma resposta com código HTTP 200 e os dados requeridos no campo “data”, no formato indicado na subseção 4.2.6. Este caso é exemplificado na figura 5.8.

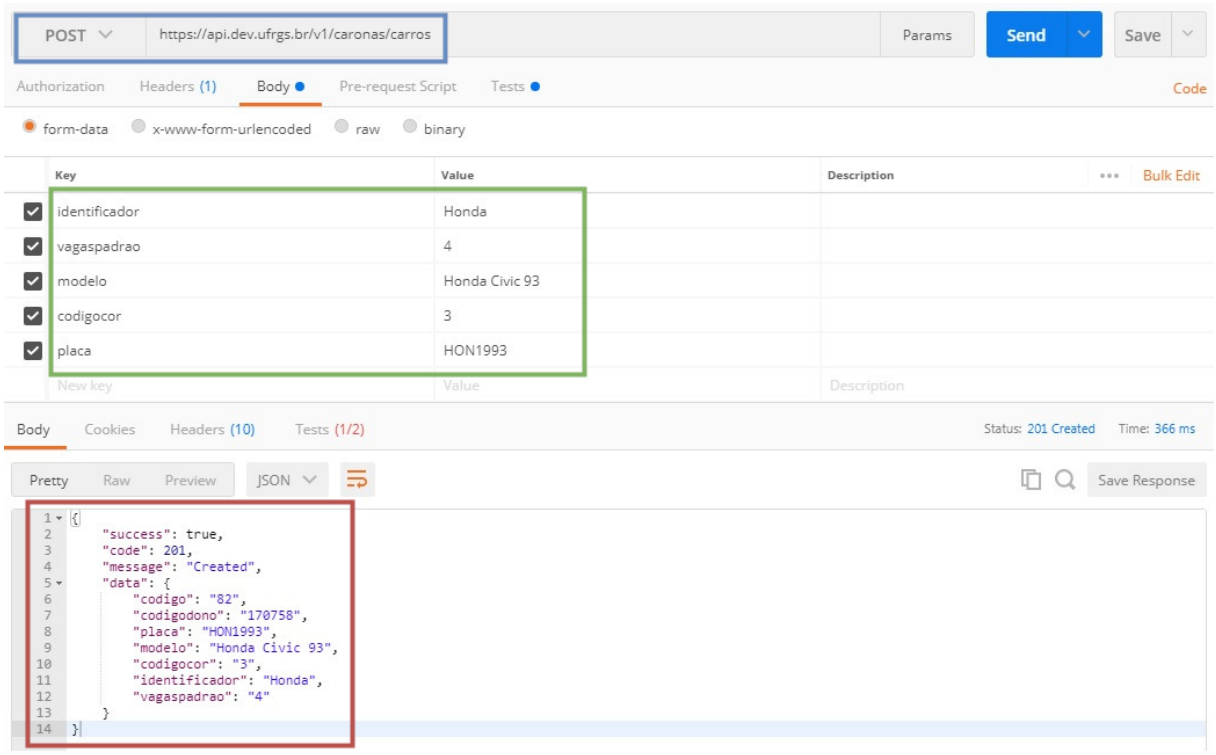
Figura 5.8 – Exemplo de execução do método HTML GET no recurso /oferecimentos



Fonte: Produzido pelo autor

Caso o cliente execute um comando HTTP POST, ele deve receber uma resposta com o código HTTP 201 e os dados da estrutura criada no campo “data”. Este caso é exemplificado na figura 5.9.

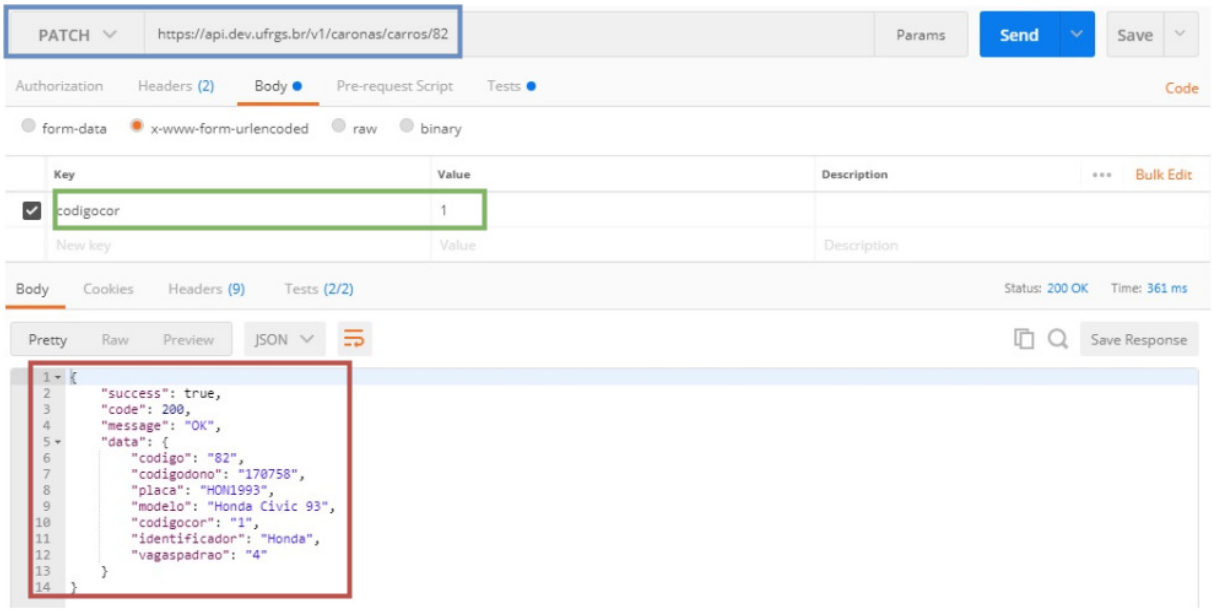
Figura 5.9 – Exemplo de execução do método HTTP POST no recurso /carros



Fonte: Produzido pelo autor

Caso o cliente execute um comando HTTP PUT, PATCH ou DELETE, ele deve receber uma resposta com o código HTTP 200 e os campos da estrutura modificada no campo “data”. Este caso é exemplificado na figura 5.10.

Figura 5.10 – Exemplo de execução do método HTML PATCH no recurso /carros/82



Fonte: Produzido pelo autor

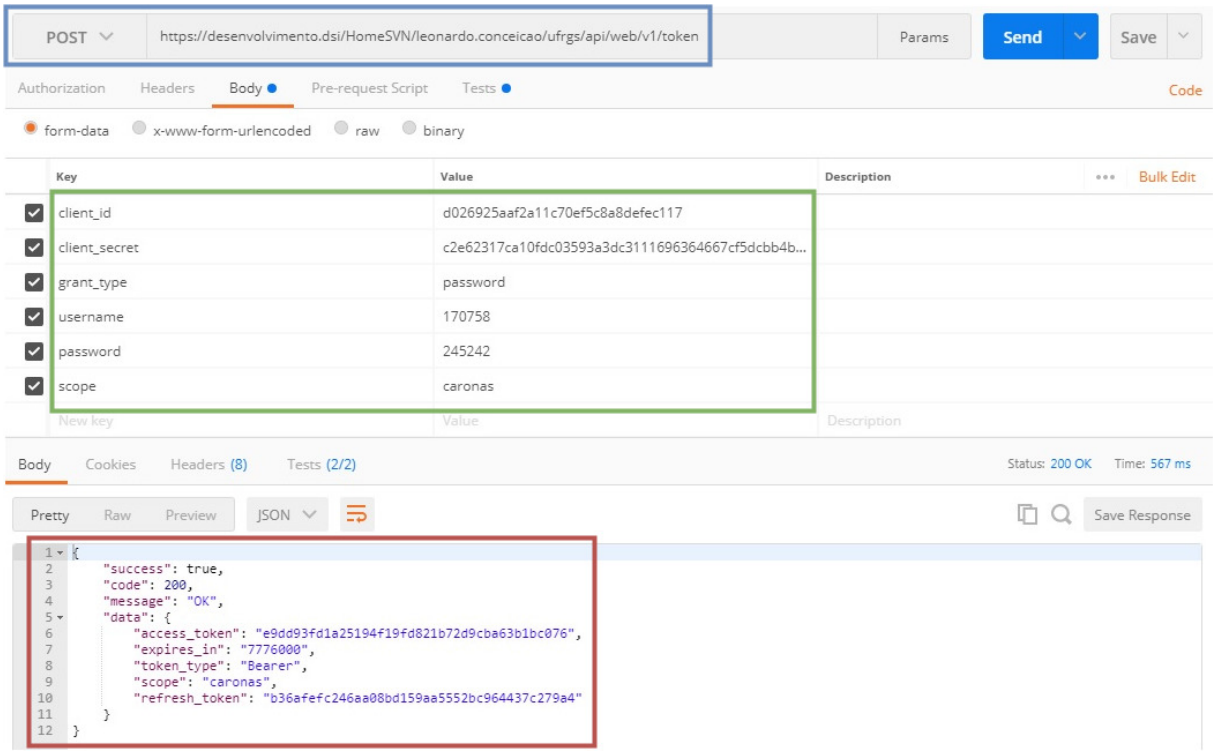
Nas subseções de 5.2.8.1 até 5.2.8.5 são mostrados casos de uso comuns ao uso da aplicação Caronas.

5.2.8.1 Iniciação do aplicativo móvel caronas, de acordo com o fluxo

Primeiramente, o aplicativo deve adquirir um *token* de autorização de uso do serviço, se isso ainda não foi feito, executando-se HTTP POST na URI apropriada, que é implementada no núcleo da API UFRGS, como ilustrado na figura 5.11. Depois disso, se o usuário ainda não aceitou os termos de uso, os mesmos devem ser aceitos através do método HTTP POST aplicado ao recurso /usuarios, como ilustrado na figura 5.12.

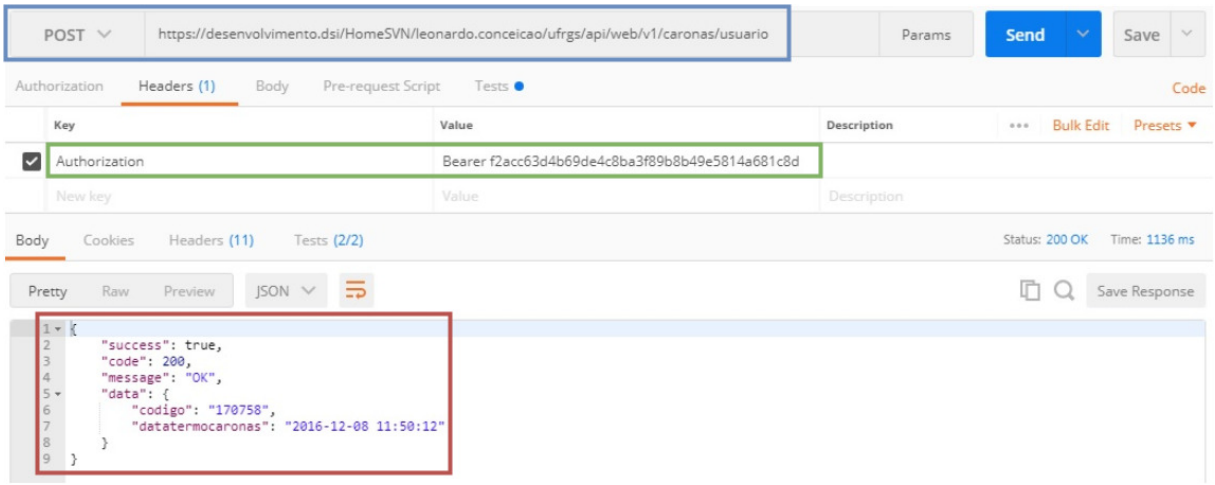
Depois da autenticação, a aplicação utiliza o *token* para executar as próximas requisições. Ela executa então uma requisição HTTP GET no recurso oferecimentos, com a variável extend=carro,motorista, assim recebendo tanto os dados do oferecimento como os dados necessários para exibir a foto e o tipo de carro no mapa, como ilustrado nas figuras 5.13 e 5.14. Caso necessário, são feitas outras requisições HTML GET nas páginas restantes para acessar todos os oferecimentos.

Figura 5.11 – Exemplo de obtenção de *token* de autenticação



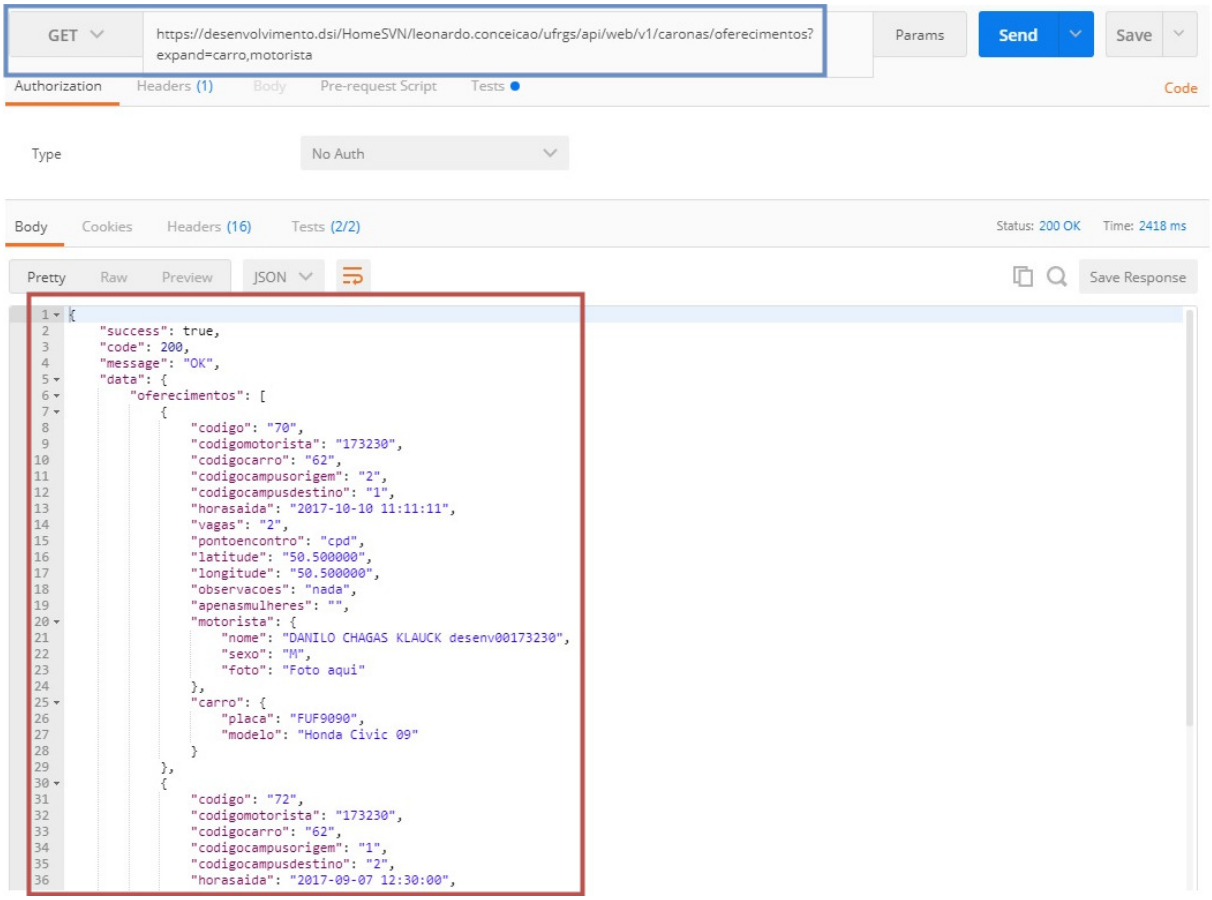
Fonte: Produzido pelo autor

Figura 5.12 – Exemplo de aceite de termos de uso



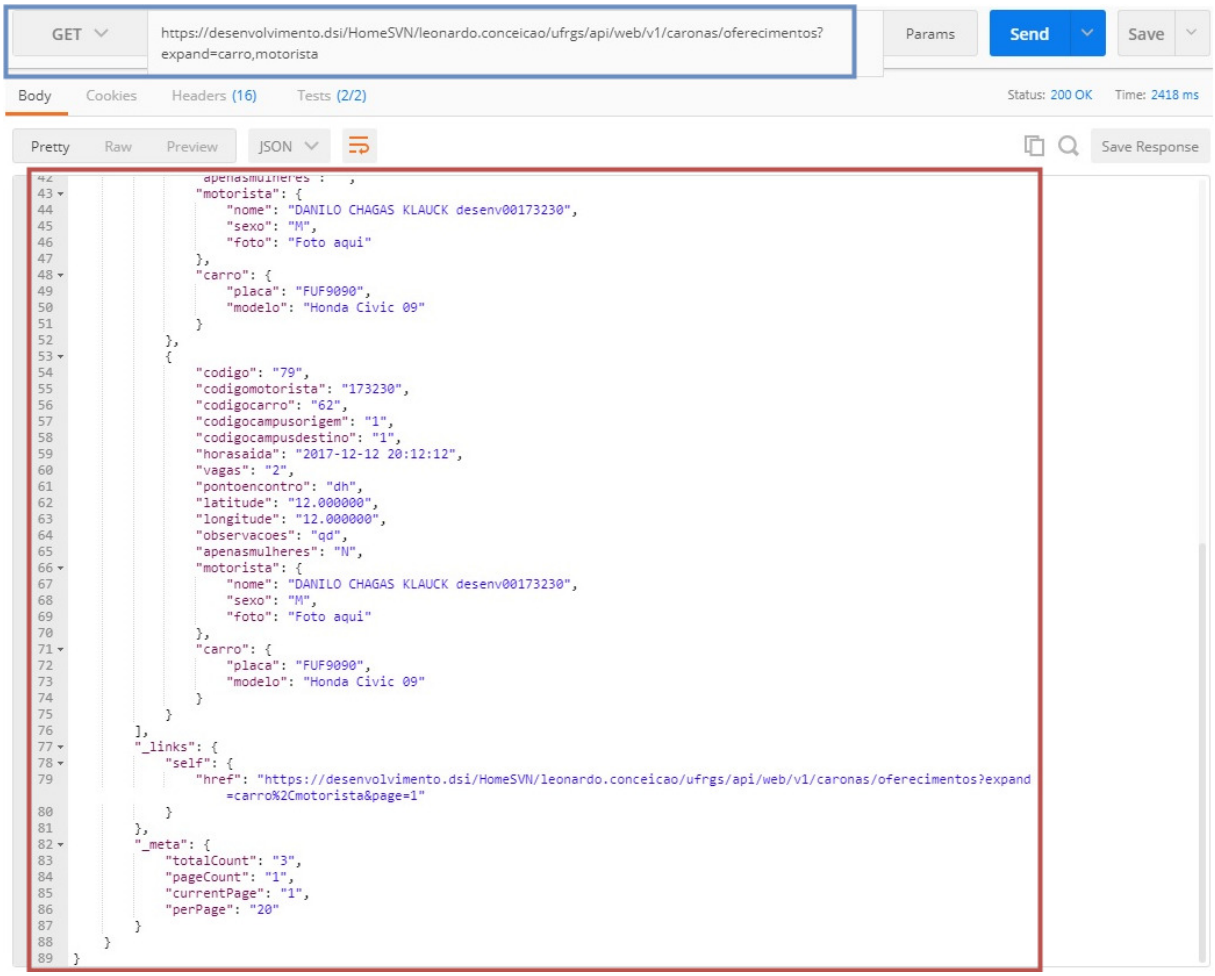
Fonte: Produzido pelo autor

Figura 5.13 – Exemplo de execução do método HTML GET com a variável *expand* preenchida



Fonte: Produzido pelo autor

Figura 5.14 – Fim da resposta da execução do método HTML GET com a variável *expand* preenchida



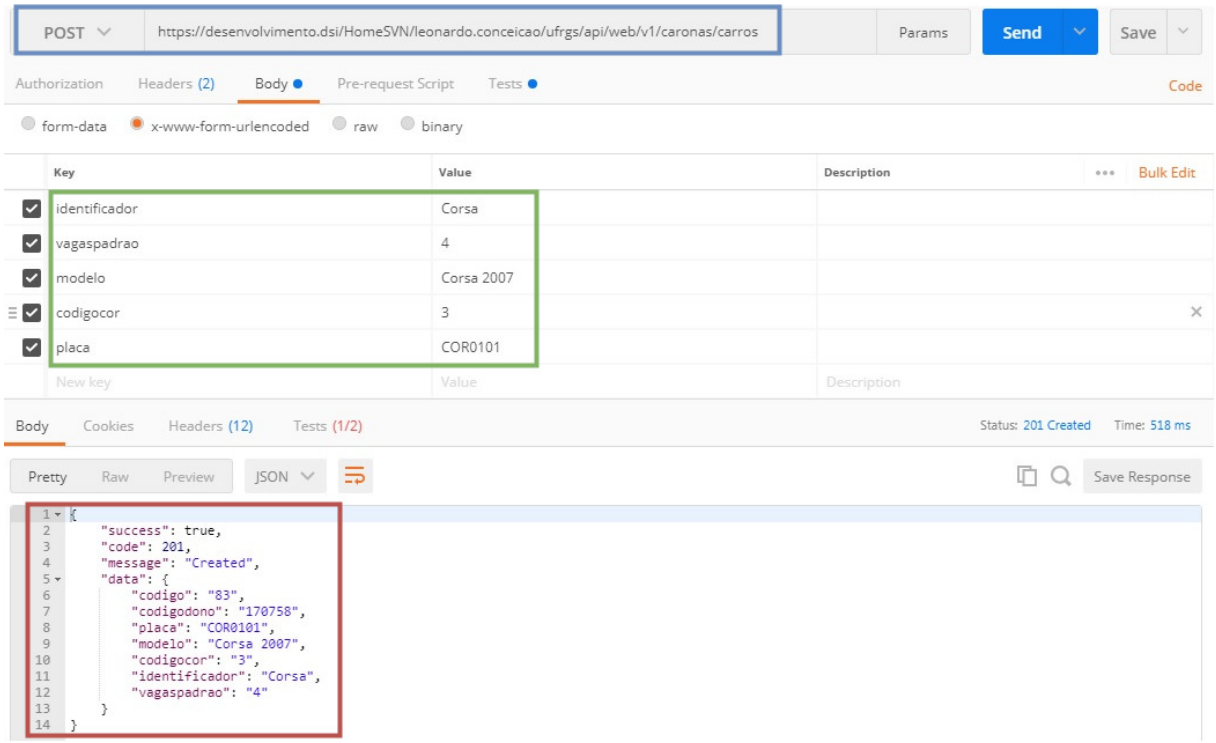
Fonte: Produzido pelo autor

5.2.8.2 Criar oferecimento

Para criar um oferecimento de carona, o usuário precisa ter uma entrada de dados sobre um carro ligado ao seu nome. Se uma entrada ainda não existe, executa-se HTTP POST no recurso /carros, enviando os dados necessários através do formato de formulário do HTML, como ilustrado na figura 5.15.

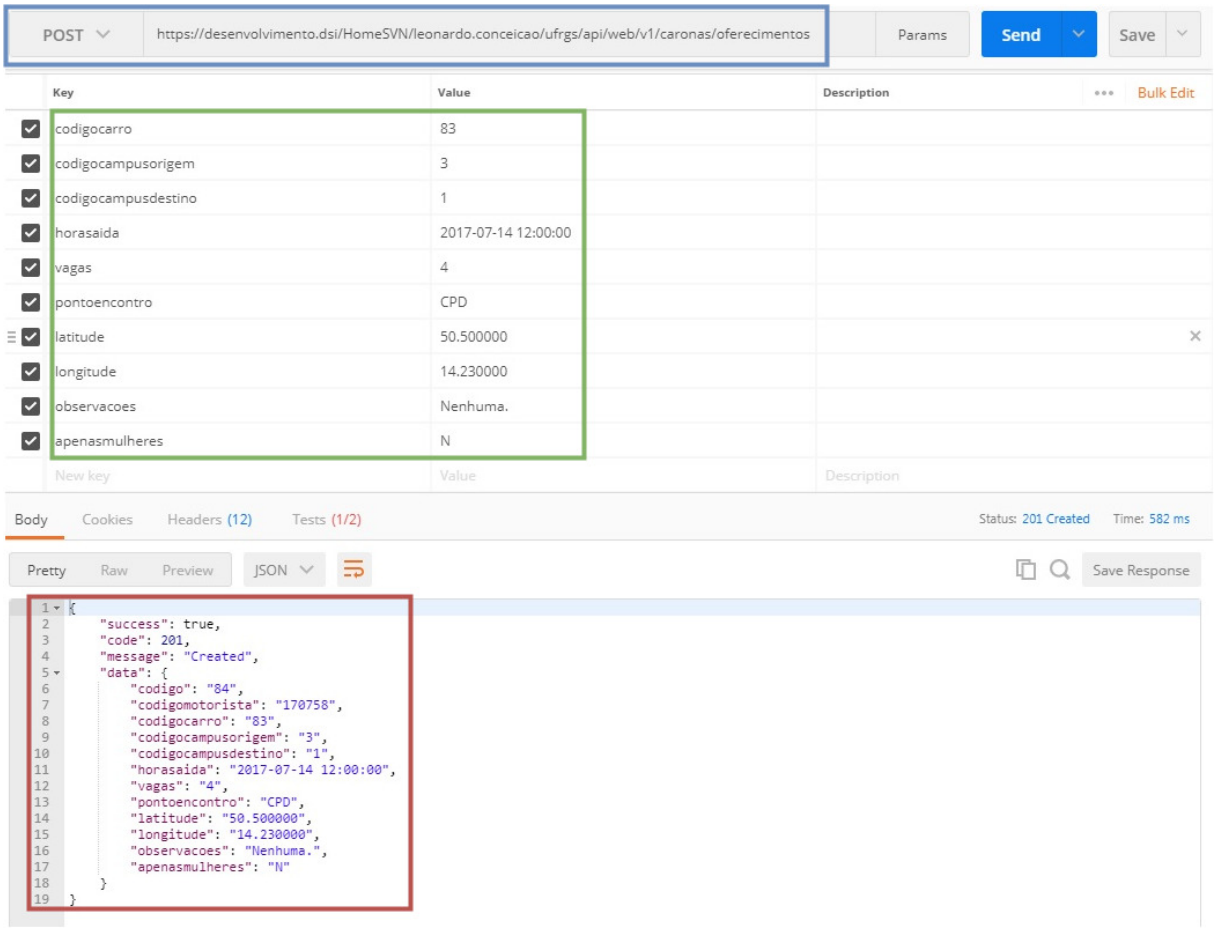
Depois disso, é possível utilizar o método HTTP POST no recurso /oferecimentos usando o código do carro vinculado, como mostrado na figura 5.16.

Figura 5.15 – Exemplo de criação de entrada de dados sobre um carro



Fonte: Produzido pelo autor

Figura 5.16 – Exemplo criação de oferecimento de carona

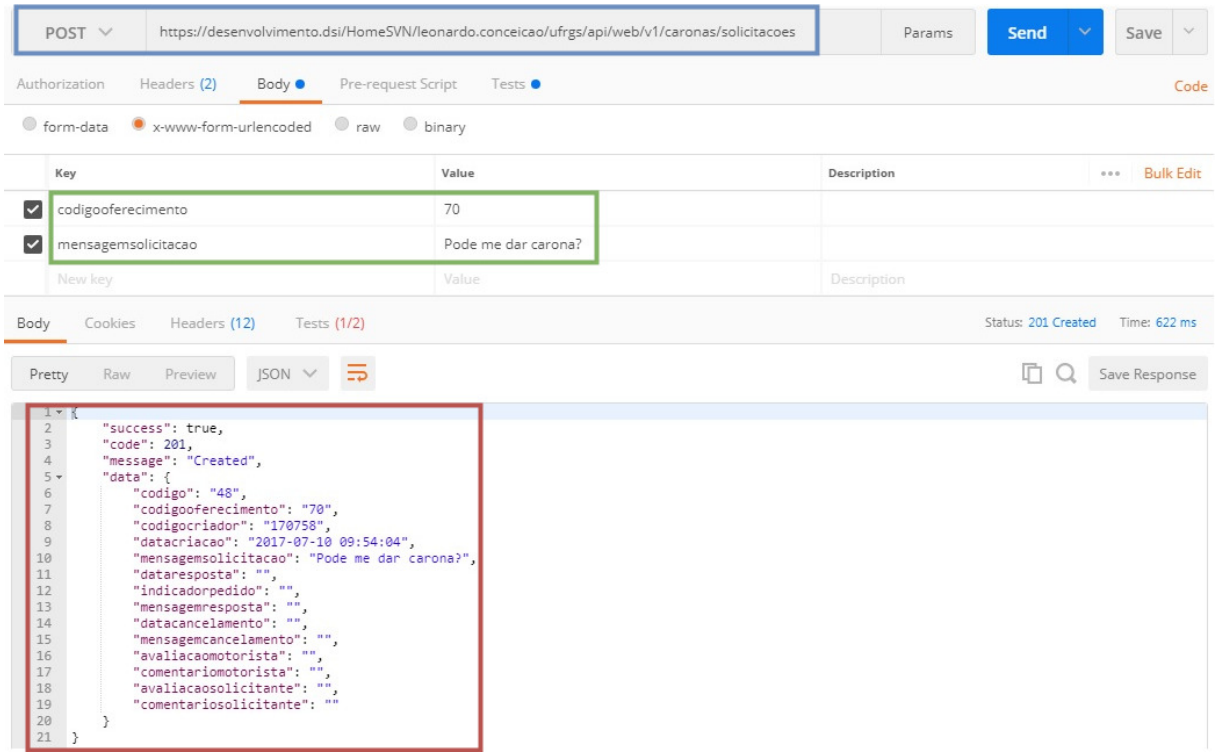


Fonte: Produzido pelo autor

5.2.8.3 Criar solicitação

Tendo o código de um oferecimento acessível pelo usuário, como os recebidos na subseção 4.2.8.1, simplesmente executa-se o método HTML POST no recurso /solicitacoes, enviando o código e a mensagem de solicitação. Este passo é ilustrado na figura 5.17. Releva-se que as vagas do oferecimento só são subtraídas à medida que o criador do oferecimento aceita solicitações.

Figura 5.17 – Exemplo de criação de solicitação de carona

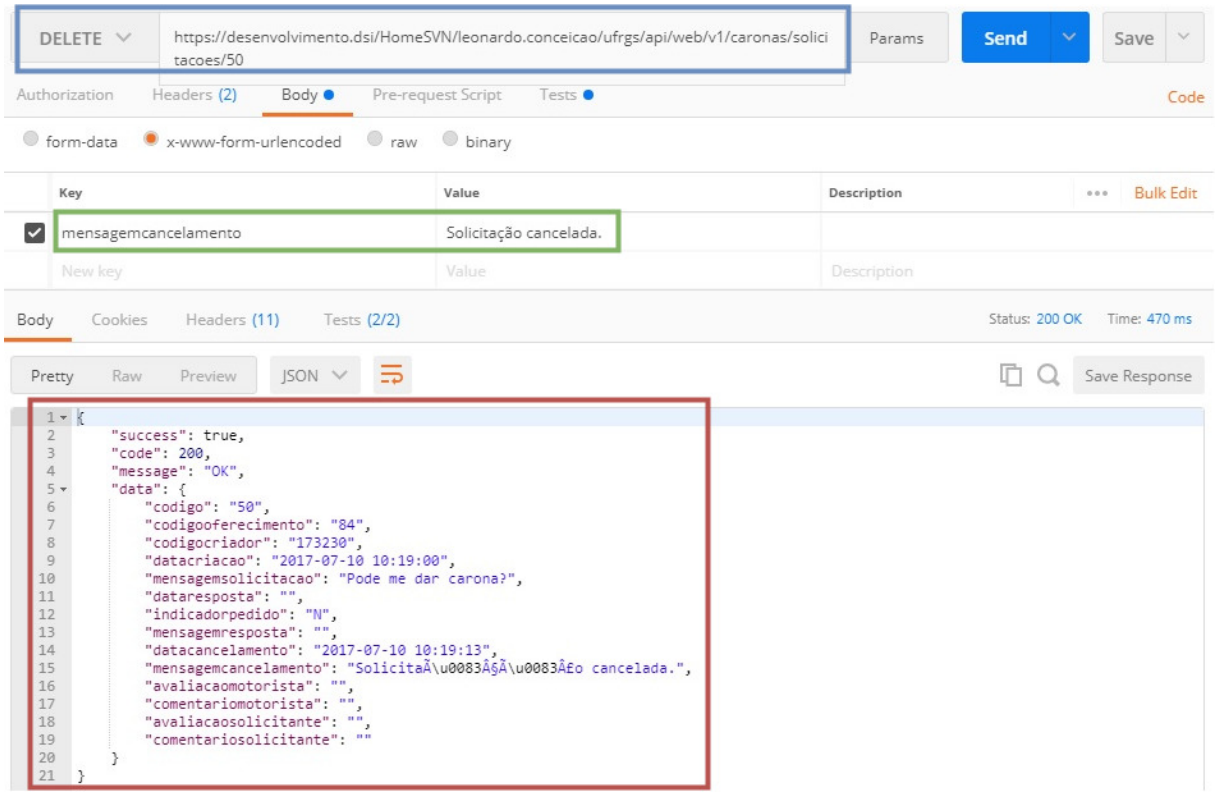


Fonte: Produzido pelo autor

5.2.8.4 Negar ou aceitar solicitação

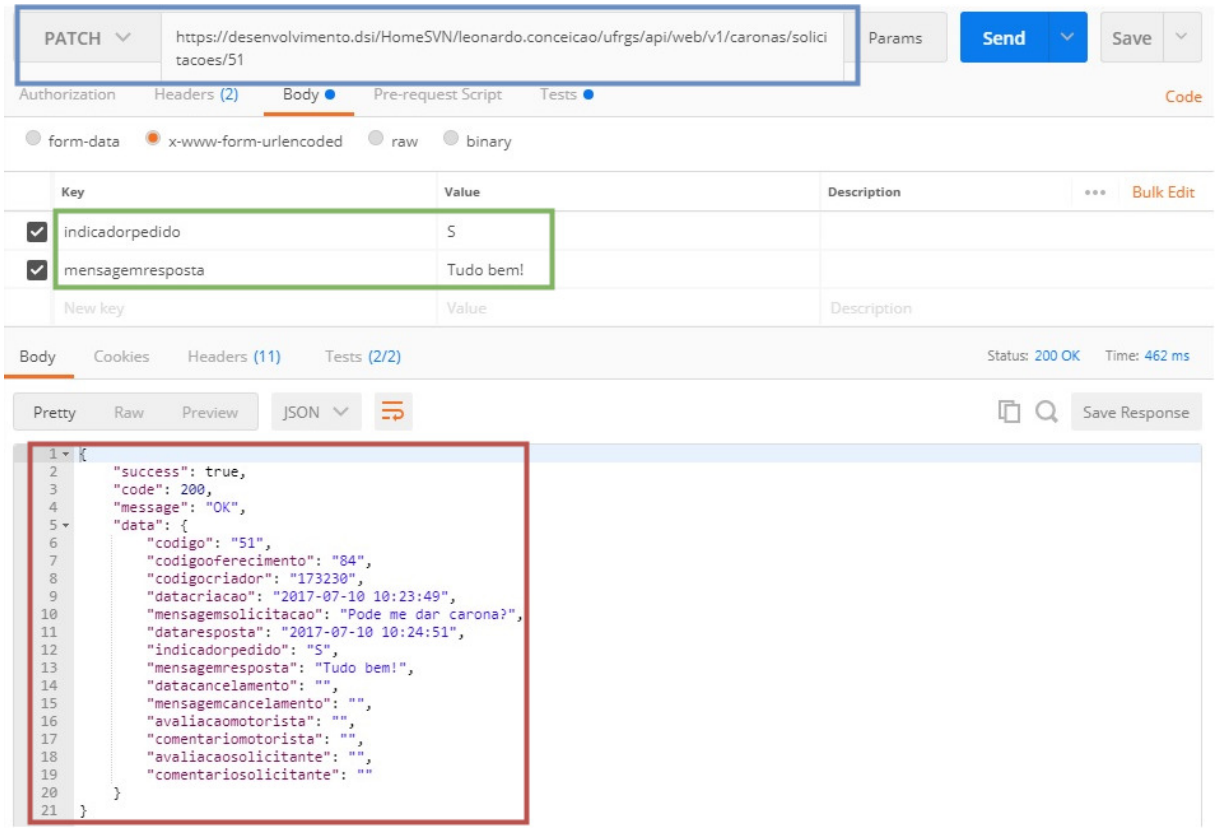
Tendo o código da solicitação, é possível cancelá-la usando o método HTTP DELETE ou aceitá-la usando o método HTTP PUT ou HTTP PATCH no recurso /solicitacoes/{código da solicitação}. Estes casos são ilustrados nas figuras 5.18 e 5.19.

Figura 5.18 – Exemplo de cancelamento de solicitação



Fonte: Produzido pelo autor

Figura 5.19 – Exemplo de aceite de solicitação

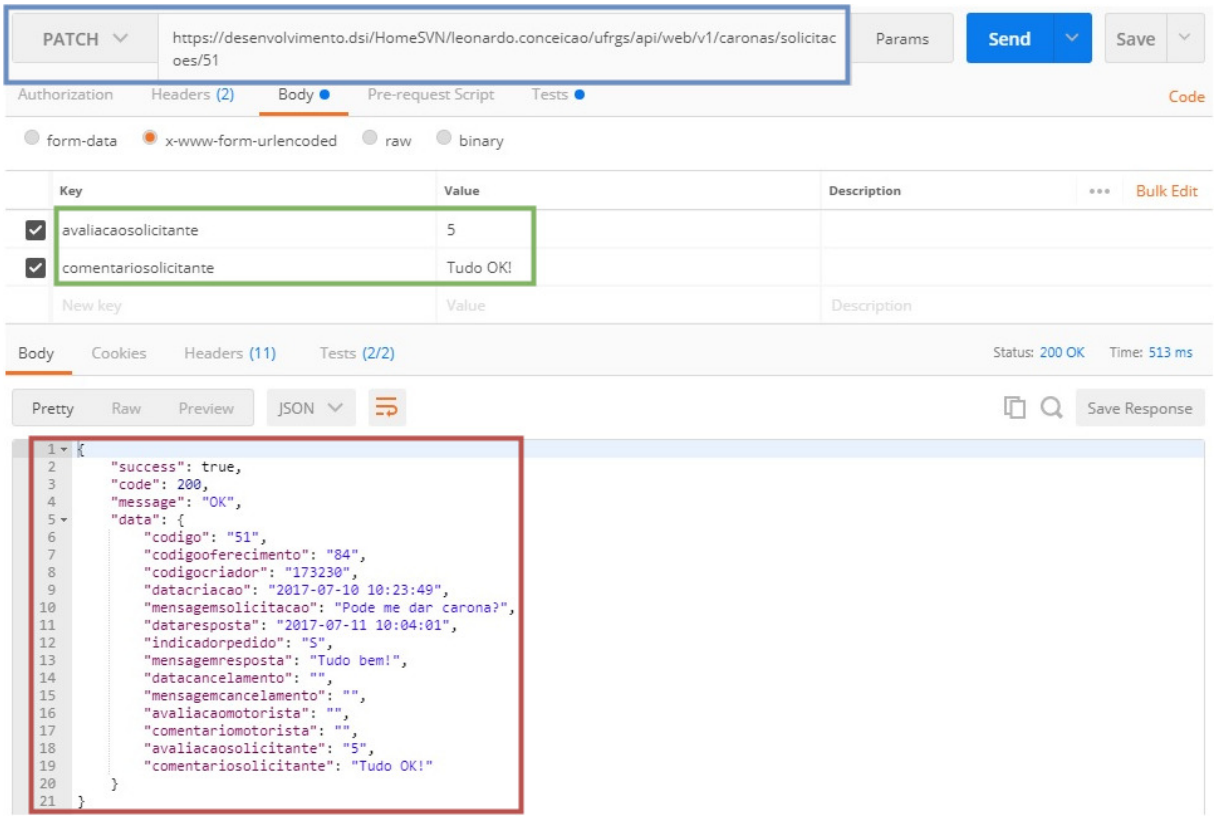


Fonte: Produzido pelo autor

5.2.8.5 Avaliar motorista ou solicitante

Para inserir a avaliação do motorista ou solicitante, executa-se o método HTTP PUT ou PATCH no recurso solicitacoes/{código da solicitação} enquanto um *token* de uso do serviço pertencente ao usuário adequado é usado. Um exemplo de avaliação de solicitante é ilustrado na figura 5.20.

Figura 5.20 – Exemplo de avaliação de solicitante

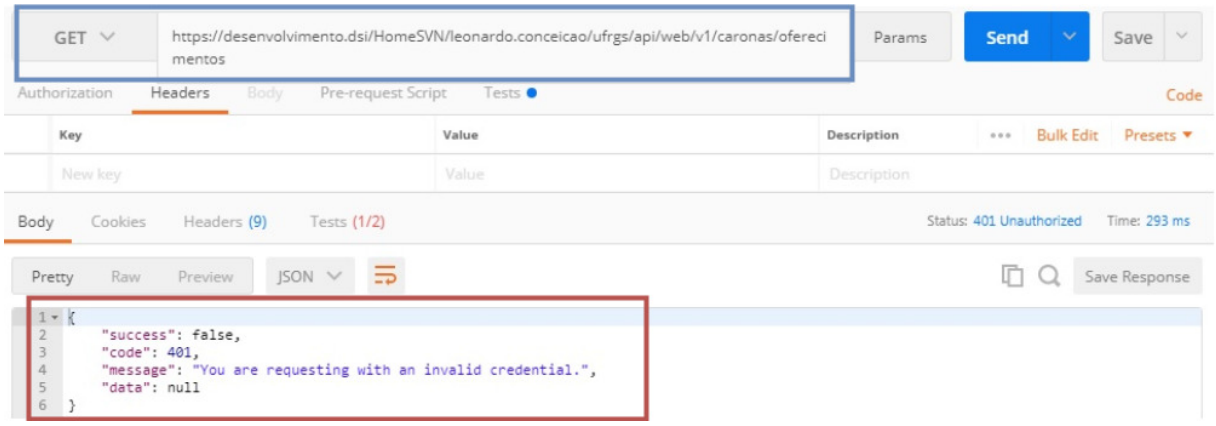


Fonte: Produzido pelo autor

5.2.9 O que pode dar errado

A primeira verificação em uma chamada ao serviço é a da autorização do usuário. Caso o usuário tente fazer uma requisição sem um *token* ou com um *token* inválido é emitida uma mensagem de erro com o código HTML 401, como ilustrado na figura 5.21

Figura 5.21 – Resposta a uma requisição sem um *token* válido

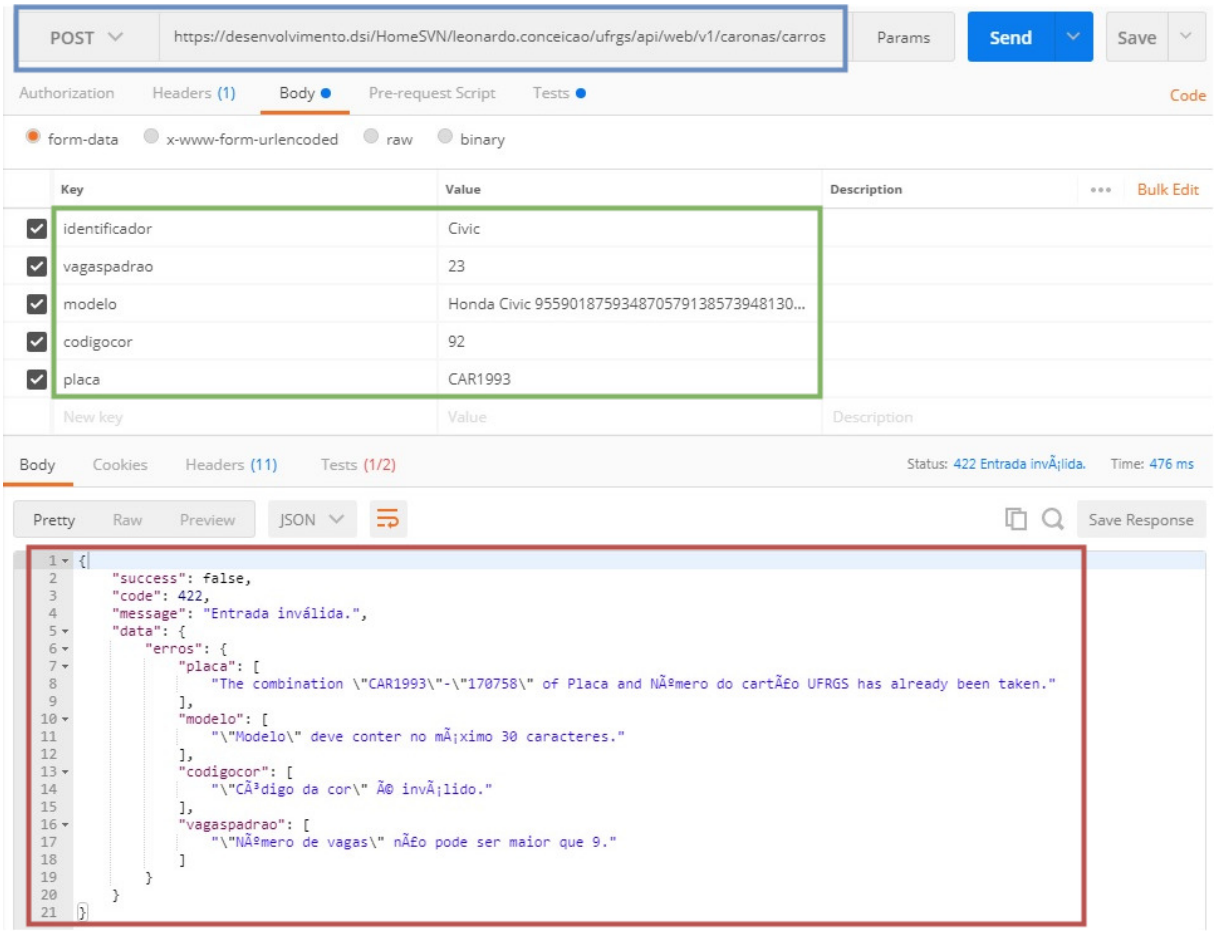


Fonte: Produzido pelo autor.

Caso seja feita uma requisição devidamente autorizada, outros erros comuns ainda podem acontecer, como tentativa de acesso a um recurso não existente, *token* sem o escopo Caronas, etc. Estes erros também retornam uma mensagem como na figura 5.21, com o código e mensagem adequados.

No caso de a requisição usar os métodos HTML POST, PUT ou PATCH, é possível que o usuário tenha inserido dados que não satisfazem as restrições descritas na subseção 4.2.5. Neste caso, o usuário recebe uma mensagem de erro com o código HTML 422, trazendo no campo data um envelope “erros” contendo estruturas que representam os campos problemáticos e uma descrição do porque a restrição não foi satisfeita, como ilustrado na figura 5.22.

Figura 5.22 – Resposta a uma requisição que não satisfaz as restrições dos seus campos



Fonte: Produzido pelo autor.

6 CONCLUSÃO

Neste capítulo são apresentadas as conclusões do trabalho. A seção 6.1 expõe os resultados do trabalho. A seção 6.2 trata das limitações da implementação e a seção 6.3 disserta sobre possíveis trabalhos futuros.

6.1 Resultados

Ao fim do trabalho, obteve-se um *back-end* com funcionalidade muito próxima da requisitada pelo CPD. A integração e testes básicos com um protótipo do cliente *mobile* não revelou erros na implementação. O *web service* implementa restrições que garantem a consistência do banco de dados. O processo de autenticação implementado pelo núcleo da API UFRGS impede o acesso por usuários não identificados. A implementação usa as mesmas ferramentas usadas pelo CPD e tem seu código comentado. Foi implementada a funcionalidade de resposta em inglês e português, antecipando o uso por alunos estrangeiros.

6.2 Limitações

Algumas funcionalidades requisitadas não foram atendidas. Umas das funcionalidades pedidas foi a implementação do envio de *push notifications* aos dispositivos móveis clientes. A implementação inicial foi concluída, porém não foi testada.

O projeto também não foi bem testado. Foram feitos apenas alguns testes básicos, para verificar o funcionamento e *bugs* mais óbvios, como nas restrições de valores. Não foram feitos registros sobre estes resultados, pois foi planejada uma suíte de testes automatizada, ainda não implementada. O *back-end* não chegou a ser testado fora de um ambiente de produção.

Existem alguns problemas de formatação nas mensagens de algumas respostas

6.3 Trabalhos Futuros

É necessário concluir o *front-end* para que a aplicação rode como planejado. Seria possível também a implementação de clientes *web* para a aplicação.

O framework Yii 2 permite a criação de testes automatizados, dariam maior robustez à implementação.

É possível adicionar novas línguas ao português e inglês já implementados.

REFERÊNCIAS

ABEGG, Matheus Pacheco. **Desenvolvimento de um protótipo de solução colaborativa para o controle de listas de compras.** 2014. 49 f. TCC (Graduação) – Curso de Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2014.

BYND SERVIÇOS DE TECNOLOGIA LTDA. **Bynd caronas corporativas.** Disponível em: <https://bynd.com.br/>. Acesso em: 04 ago. 2017.

CARONETAS. **Caronetas – Caronas Inteligentes.** Disponível em: <https://www.caronetas.com.br/>. Acesso em: 04 ago. 2017.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. 180 f. Tese (Doutorado) – Curso de Information And Computer Science, University Of California, Irvine, 2000.

GUDGIN, Martin et al (Ed.). **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition).** 2007. Disponível em: <https://www.w3.org/TR/soap12/>. Acesso em: 05 nov. 2016.

POSTDOT TECHNOLOGIES INC. **Postman | Supercharge your API workflow.** Disponível em: <https://www.getpostman.com/>. Acesso em: 04 ago. 2017.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Web Services.** Sebastopol: O'reilly Media, 2007. 448 p.

SOUSA, Maurício Vieira de. **SACI : sistema de apoio à coleta de informações.** 2014. 58 f. TCC (Graduação) – Curso de Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2014.

YII SOFTWARE LLC. **Yii PHP Framework: Best for Web 2.0 Development.** Disponível em: <http://www.yiiframework.com/>. Acesso em: 02 ago. 2017