AGÊNCIA EXPERIMENTAL DA ENGENHARIA DE SOFTWARE
PROJETO STOP DA CIDADANIA

DOCUMENTO DE ARQUITETURA E PROJETO DETALHADO

Professor responsável:
Daniel Callegari

Arquitetos responsáveis:
Marlon Furtado
marlon.furtado@edu.pucrs.br

Luis Gustavo Santana
luis.santana@edu.pucrs.br

Porto Alegre, 24 de junho de 2019

Introdução

O objetivo deste documento é fornecer uma visão geral do planejamento da arquitetura e do projeto detalhado no desenvolvimento do projeto **Stop da Cidadania**, realizado durante o semestre 2019/01 na Agência Experimental de Engenharia de Software (AGES) do curso de Engenharia de Software da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Este documento utiliza o modelo C4 de documentação e abrange o escopo, definições, modelos e a visão geral da Arquitetura de Software utilizados no projeto.

Projeto

O projeto é um aplicativo baseado no jogo "STOP" e nasceu da necessidade de o cidadão aprender sobre conceitos básicos relacionados à vida pública (ciência política, direito constitucional, economia, etc..), e conceitos sobre cidadania, com o foco principal nos jovens. O principal objetivo é ter uma plataforma que estimule o aprendizado dos jovens de uma forma lúdica e divertida.

Escopo

Está definido como parte do escopo do projeto:

1. Jogo
    a. Cadastro dos jogadores;
    b. Definição das configurações para iniciar jogo;
    c. Sala de jogo multiusuários;
    d. Randomizar letra conforme as categorias;
    e. Calcular e gerar o resultado do jogo;

2. Administrativo
    a. Cadastro e edição das categorias;
    b. Cadastro e edição das alternativas;
    c. Controle dos dados e informações das partidas jogadas;
    d. *Download* das informações dos jogos e usuários;

Tecnologias

A principal tecnologia utilizada no projeto foi predominantemente a linguagem Javascript, com Node.js para a API e o framework Vue.js para a aplicação frontend.
Em complementação ao Node.js, as seguintes tecnologias foram escolhidas para apoiar o desenvolvimento do backend:

- **Express.js:** um framework para construção de webservices RESTful que provê um conjunto de funcionalidades robustas para criação de APIs de forma fácil e rápida.
- **KeystoneJS:** é um sistema de gerenciamento de conteúdo. Feito para ser usado como um framework para desenvolvimento de aplicações web. É construído com base no *express.js* e *mongoose*.

Para a construção do *frontend* da aplicação foi utilizado o Vue.js. Um framework progressivo para a construção de interfaces de usuário. A aplicação foi construída com base nos princípios do PWA (Progressive Web App) para ter um maior alcance, funcionar de forma independente de um sistema operacional e ter a usabilidade de um aplicativo nativo.

Banco de Dados

O banco de dados escolhido foi o MongoDB, um banco não relacional e orientado a documentos.
Por ser orientado à documentos JSON (armazenados em modo binário), as aplicações podem modelar a informação de modo mais natural, pois os dados podem ser aninhados em hierarquias complexas e ainda serem indexáveis e fáceis de buscar, igual ao que já é feito em javascript.

Além disso, do ponto de vista do desenvolvedor, usar MongoDB permite criar uma stack completa apenas usando javascript. Uma vez que temos javascript no lado do cliente, do servidor (com Node) e do banco de dados (com Mongo).

O desenvolvimento da modelagem do banco de dados passou por diversas fases. A primeira modelagem foi o modelo ER, usado para ter uma ideia inicial da estrutura.

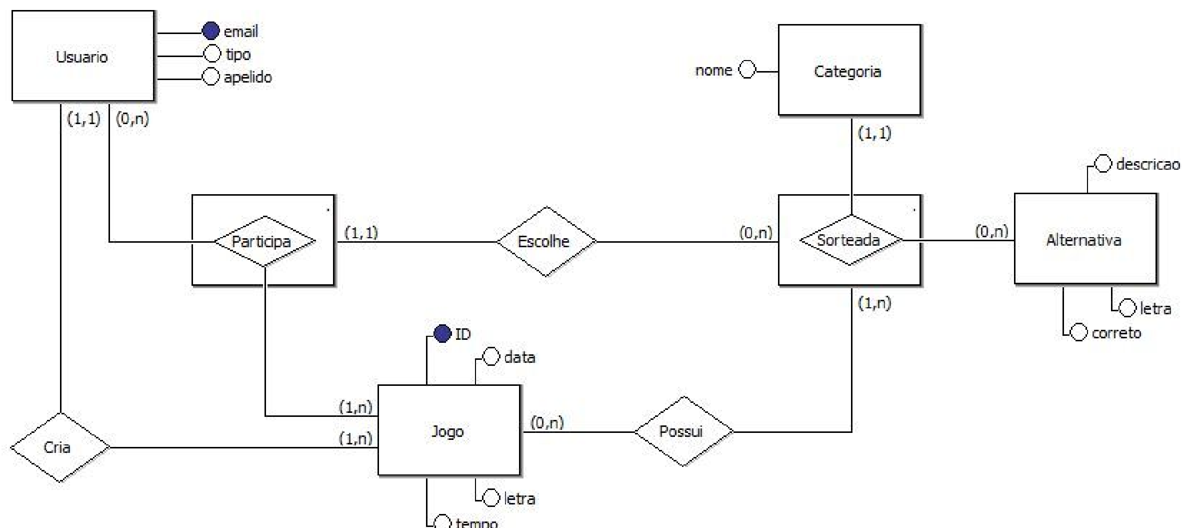

Figura 1: primeiro modelo do banco de dados (modelo ER)

Após o entendimento da aplicação. O segundo momento foi desenvolver a estrutura JSON que seria armazenada no banco de dados.
Após diversas melhorias, as estruturas finais são apresentadas abaixo.

```
Game {
  gameId,
  status,
  maxNumberCategories,
  maxNumberPlayers,
  initTime,
  endTime,
  date,
  user,
  letter,
  time,
  category
}
```

```
User {
  email,
```

```
  nickname,
  type,
  score,
  game,
  alternative
}
```

```
Category {
  name,
  letter,
  alternative,
  game,
  isDisabled
}
```

```
Alternative {
  description,
  isCorrect,
  category,
  user
}
```

Arquitetura

Foi escolhido o modelo C4 de documentação para arquitetura de software. Esse modelo consiste em um conjunto hierárquico de diagramas de arquitetura de software para contexto, containers, componentes e código.

A hierarquia dos diagramas C4 fornece diferentes níveis de abstração e cada um deles é relevante para um público alvo.

O motivo da escolha desse modelo é devido a sua facilidade de desenvolvimento e entendimento pelas partes envolvidas.

Nível 1 - Diagrama de Contexto do Sistema

O diagrama de contexto mostra o sistema que está sendo construído. No diagrama proposto os jogadores usam uma aplicação para jogar, enquanto o administrador possui uma outra aplicação para gerenciar os dados. Ambas aplicações estão conectadas e utilizam a mesma API.

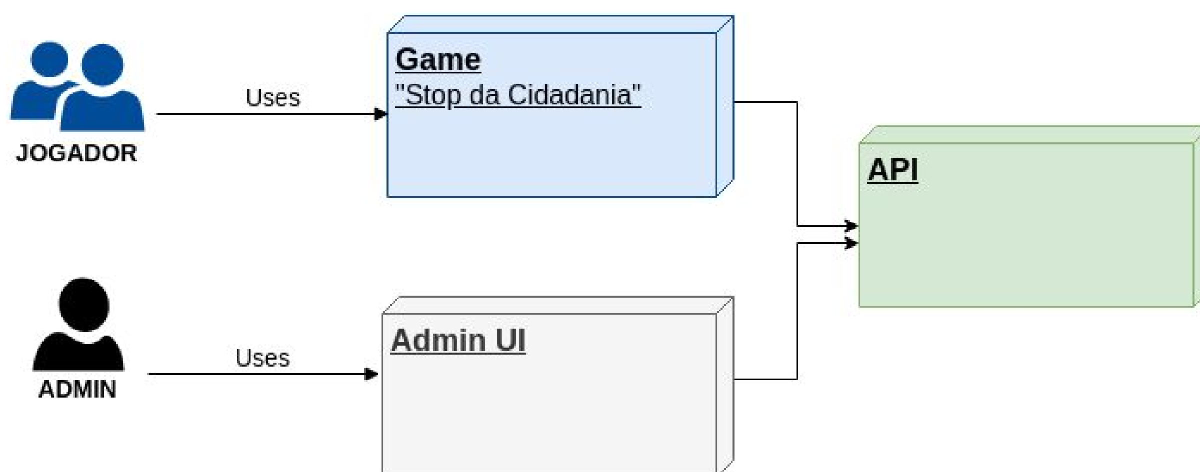

Figura 2: Diagrama de contexto

Nível 2 - Diagrama de Container

O diagrama de container amplia o sistema de software e mostra os containers (aplicativos, armazenamentos de dados, serviços, etc.) que compõem esse sistema de software.
Aqui já é possível ver algumas decisões de tecnologias, decisões de arquitetura e a comunicação entre os serviços.

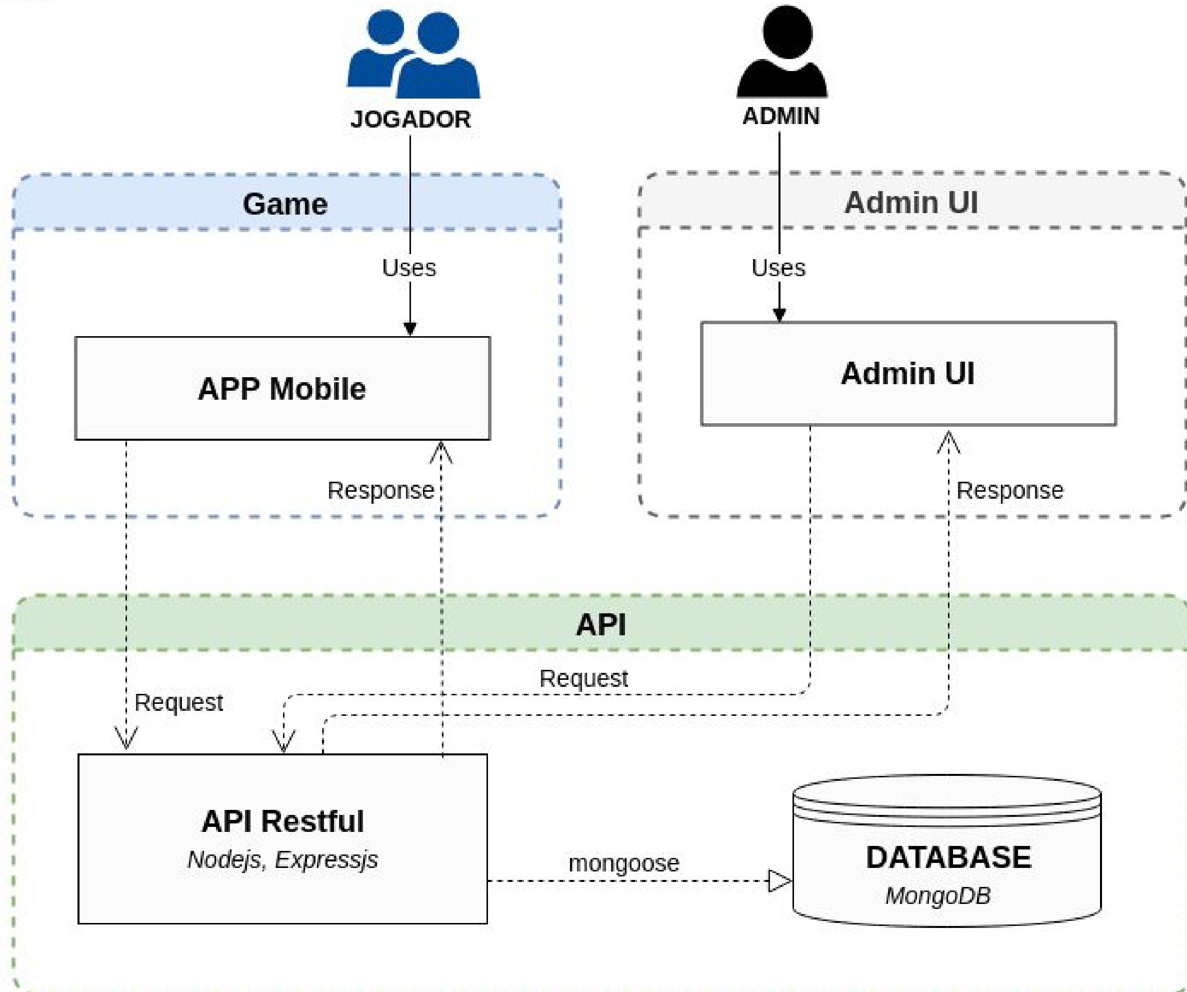

Figura 3: Diagrama de container

Nível 3 - Diagrama de Componentes

O diagrama de componentes amplia um container individual para mostrar os componentes dentro dele.

A seguir, na figura 4, é apresentado um exemplo da arquitetura interna do aplicativo e onde ocorre a comunicação com o servidor.
Na figura 5 é a arquitetura do servidor (API) e sua comunicação com o banco de dados e o retorno para o cliente.

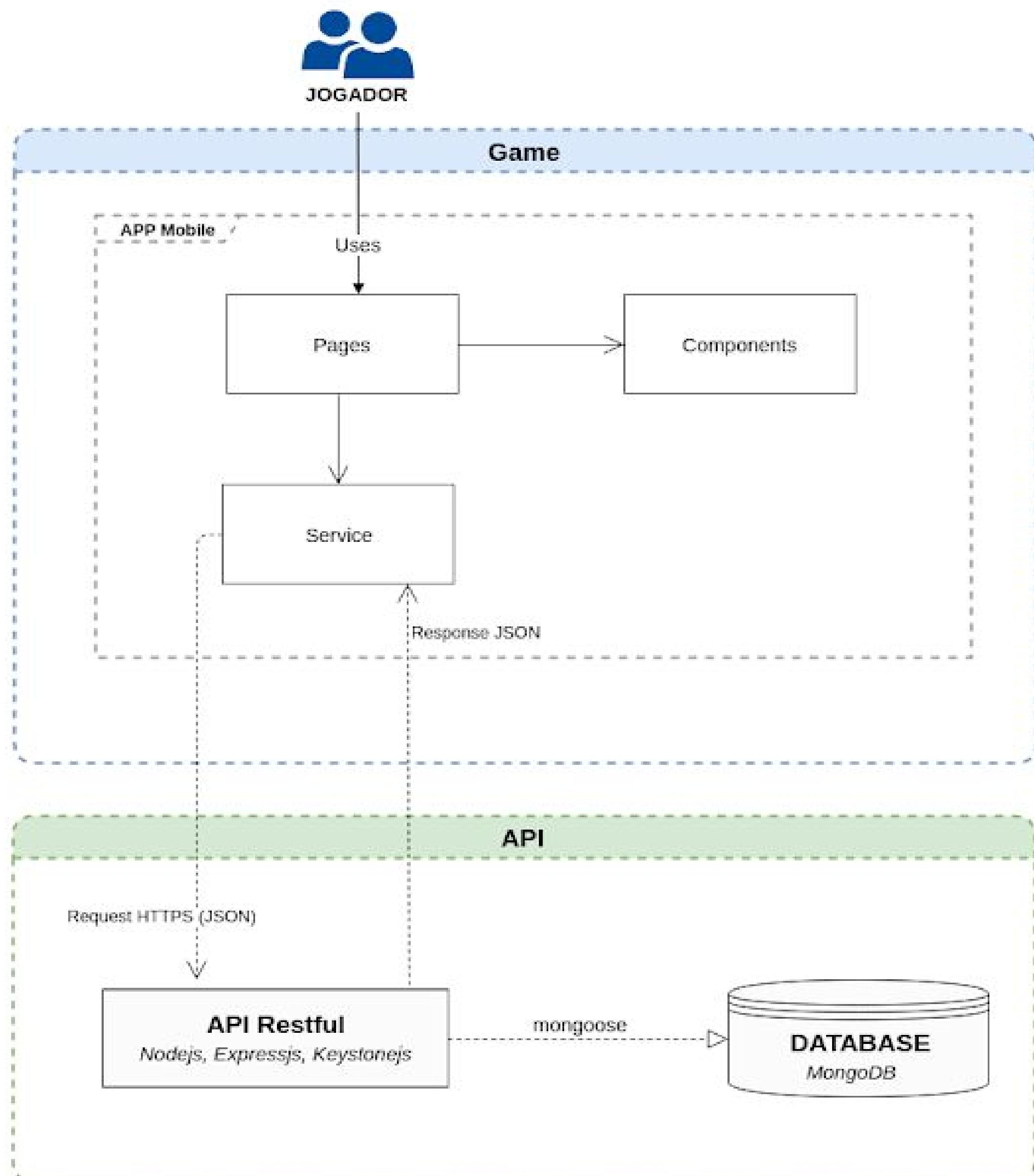

Figura 4: Diagrama de componente: comunicação API, banco de dados e aplicativo

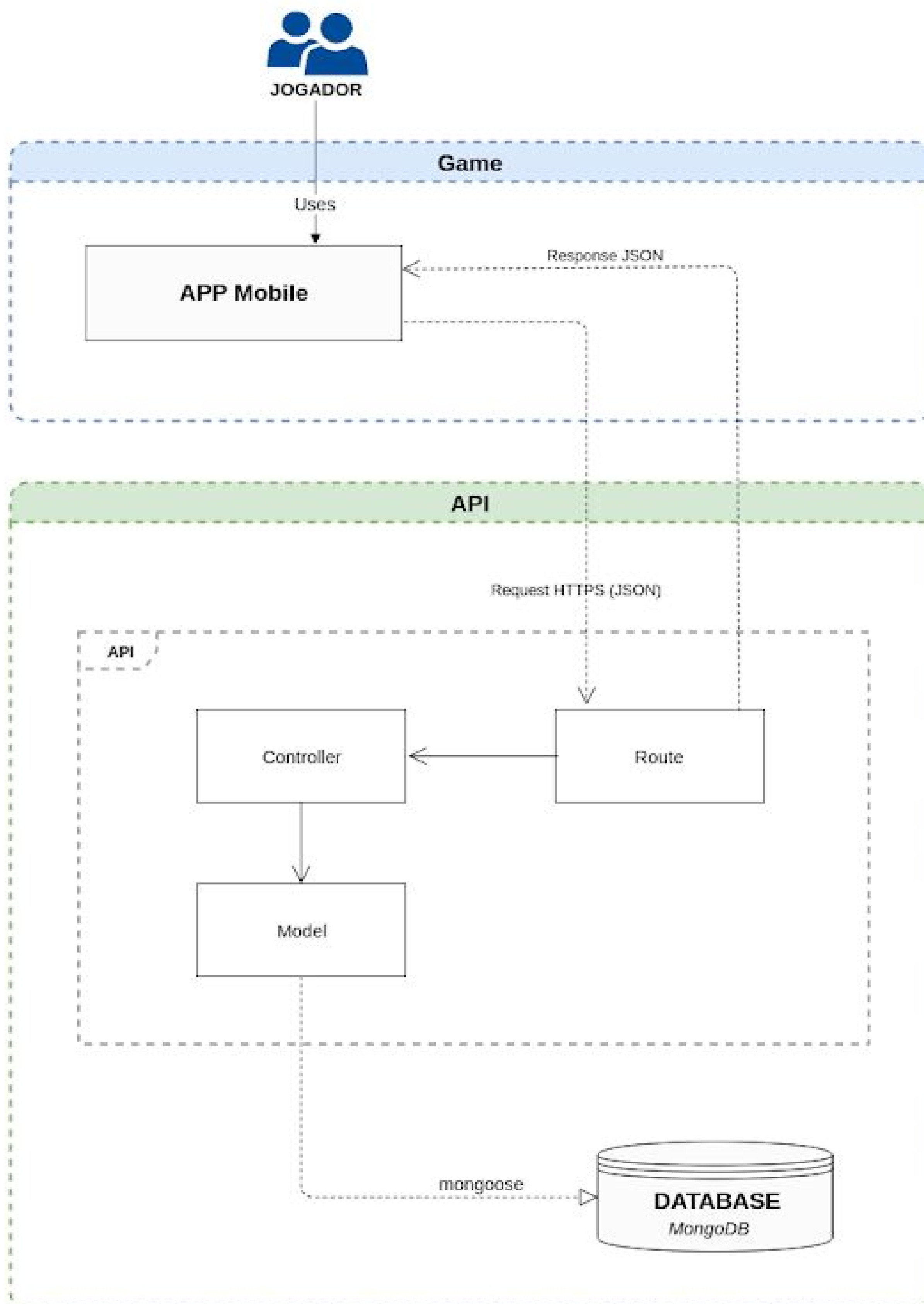

Figura 5: Diagrama de componente: comunicação API, banco de dados e aplicativo

Nível 4 - Código e Padrões de Arquitetura

O último nível é o 4. Aqui a arquitetura é detalhada a nível de código e implementação.
O padrão de arquitetura utilizada foi o de cliente-servidor em conjunto com a arquitetura REST para a construção da api e a integração com o frontend.

O modelo de cliente-servidor é uma estrutura de aplicação distribuída que distribui as tarefas e cargas de trabalho entre os fornecedores de um recurso ou serviço, designados como servidores, e os requerentes dos serviços, designados como clientes.
O objetivo desta divisão é separar a arquitetura e responsabilidades em dois ambientes. Assim, o cliente (consumidor do serviço) não se preocupa com tarefas do tipo: comunicação com banco de dados, gerenciamento de cache, log, etc. E o contrário também é válido, o servidor (provedor do serviço) não se preocupa com tarefas como: interface e experiência do usuário. Permitindo, assim, a evolução independente das duas arquiteturas.

A API, implementada em node.js, é a responsável por tratar os dados e fazer a conexão das informações entre a aplicação web (frontend) e o banco de dados. A organização foi pensada em três camadas: *Routes, Controllers e Models*.

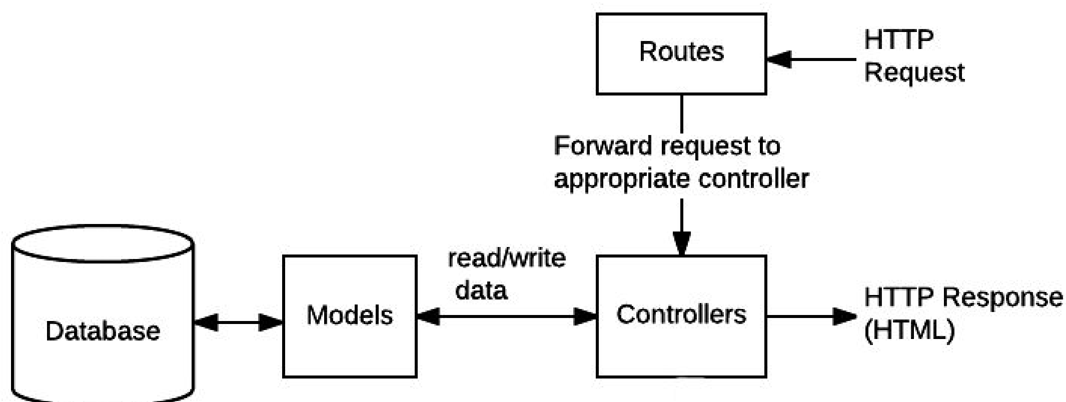

Figura 5: Organização da API em Routes, Controllers e Models

A camada de *Routes* é composta por todas as rotas da API que são expostas aos clientes para fazer a conexão via HTTP. A camada de *Models* define todas as estruturas de dados manipuladas pela *Controller*. E, por fim, a camada de *Controller* é a responsável pela lógica de negócio da aplicação. Toda rota definida em *Router* possui uma instância de *Controller* associada, responsável por sua lógica de negócio.

Para a aplicação web, implementada com o framework Vue.js, foi seguido o padrão de Component Based Architecture (Arquitetura Baseada em Componentes).

A organização foi pensada a partir do conceito de web components, onde cada "parte" da tela é um componente independente, customizável e que pode ser reutilizável em qualquer outra parte da aplicação. No exemplo a seguir, temos um exemplo da utilização de componentes no projeto representado como uma "árvore", tal qual a estrutura de dados.

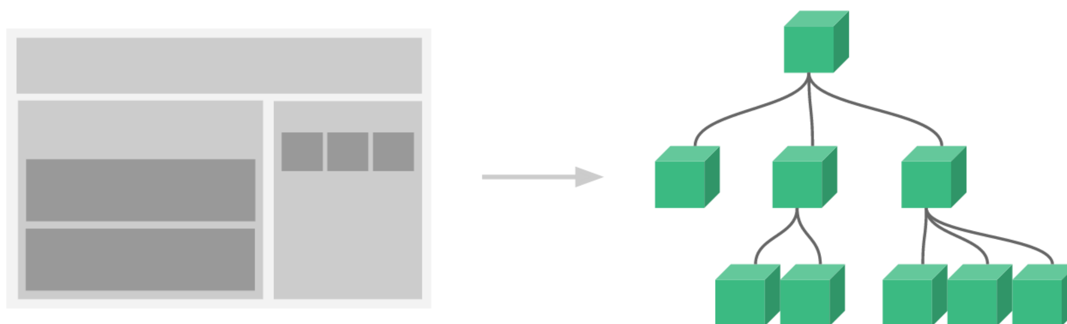

Figura 6: Representação de web components

Este tipo de arquitetura encapsula partes individuais de uma interface (componentes) em sistemas independentes e auto-sustentáveis.
Para o uso dessa arquitetura, os componentes precisam:

- Ser independentes
- Interagir com outros componentes no mesmo espaço, sem afetar ou ser afetado pelo outro
- Possuir sua própria estrutura, métodos e atributos
- Ser reutilizável em qualquer outro lugar da aplicação

Referências

INFOQ. **C4 Architecture Model.** Disponível em: <https://www.infoq.com/br/articles/C4-architecture-model>. Acesso em: 24 de junho de 2019

SIMON BROWN. **The C4 model for visualising software architecture.** Disponível em: <https://www.infoq.com/br/articles/C4-architecture-model>. Acesso em: 24 de junho de 2019

IMASTERS. **Definição, restrições e benefícios do modelo de arquitetura REST.** Disponível em: <https://imasters.com.br/desenvolvimento/definicao-restricoes-e-beneficios-modelo-de-arquitetura-rest>. Acesso em: 24 de junho de 2019.

TUTORIALS POINT. **Component-Based Architecture.** Disponível em: <https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm>. Acesso em: 24 de junho de 2019.

LINUX FOUNDATION. **NodeJS**. Disponível em <https://nodejs.org>. Acesso em: 24 de junho de 2019.

EXPRESSJS. **Express**. Disponível em: <http://expressjs.com/>. Acesso em: 24 de junho de 2019.

VUE.JS. **Vue.js**. Disponível em: <https://vuejs.org/>. Acesso em 24 de junho de 2019.

KEYSTONEJS. **KeystoneJS**. Disponível em: <https://keystonejs.com//>. Acesso em: 24 de junho de 2019.