

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Autor: Victor Jorge da Silva Gonçalves
Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF
2023



Victor Jorge da Silva Gonçalves

Monografia submetida ao curso de graduação
em Engenharia de Software da Universidade
de Brasília, como requisito parcial para ob-
tenção do Título de Bacharel em Engenharia
de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2023

Victor Jorge da Silva Gonçalves
/ Victor Jorge da Silva Gonçalves. – Brasília, DF, 2023-
55 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2023.

1. otimização. 2. aplicação web. I. Prof. Dr. Renato Coral Sampaio. II.
Universidade de Brasília. III. Faculdade UnB Gama. IV.

CDU 02:141:005.6

Errata

Elemento opcional da [ABNT \(2011, 4.2.1.2\)](#). **Caso não deseje uma errata, deixar todo este arquivo em branco.** Exemplo:

FERRIGNO, C. R. A. **Tratamento de neoplasias ósseas apendiculares com reimplantação de enxerto ósseo autólogo autoclavado associado ao plasma rico em plaquetas:** estudo crítico na cirurgia de preservação de membro em cães. 2011. 128 f. Tese (Livre-Docência) - Faculdade de Medicina Veterinária e Zootecnia, Universidade de São Paulo, São Paulo, 2011.

Folha	Linha	Onde se lê	Leia-se
1	10	auto-conclavo	autoconclavo

Victor Jorge da Silva Gonçalves

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 01 de junho de 2013 – Data da aprovação do trabalho:

Prof. Dr. Renato Coral Sampaio
Orientador

Titulação e Nome do Professor
Convidado 01
Convidado 1

Titulação e Nome do Professor
Convidado 02
Convidado 2

Brasília, DF
2023

*Dedico este trabalho à minha mãe, que sempre me apoiou e ajudou
durante minha trajetória, sendo a pessoa mais importante em todo o processo.*

Agradecimentos

Agradeço ao Prof. Dr. Renato Coral Sampaio por todo o apoio, gentileza e paciência durante o processo de orientação.

Agradeço à Prof. Dr. Carla Rocha pelo apoio e oportunidade de me aprofundar no tema através da minha participação no projeto do Brasil Participativo.

Resumo

O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (...) As palavras-chave devem figurar logo abaixo do resumo, antecidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto. O texto pode conter no mínimo 150 e no máximo 500 palavras, é aconselhável que sejam utilizadas 200 palavras. E não se separa o texto do resumo em parágrafos.

Palavras-chave: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Key-words: latex. abntex. text editoration.

Lista de ilustrações

Figura 1 – Comunicação entre as entidades do MVC	30
Figura 2 – Exemplo de <i>template</i> que renderiza objetos com <i>caches</i> aninhados . . .	35
Figura 3 – Inclusão da opção <i>touch</i> na declaração de relacionamento entre <i>models</i>	36
Figura 4 – Arquitetura do <i>Decidim</i>	41

Lista de tabelas

Tabela 1	–	Módulos dos espaços participativos padrões do <i>Decidim</i>	41
Tabela 2	–	Módulos dos espaços participativos padrões do <i>Decidim</i>	42

Lista de abreviaturas e siglas

ACID	<i>Atomicity, Consistency, Isolation, and Durability</i>
API	<i>Application Programming Interfaces</i>
DRY	<i>Don't Repeat Yourself</i>
FTP	<i>File Transfer Protocol</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
HTTPS	<i>Hypertext Transfer Protocol Secure</i>
JSON	<i>JavaScript Object Notation</i>
MGI	Ministério da Gestão e Inovação em Serviços Público
MVC	<i>Model View Controller</i>
ORM	<i>Object Relational Mapping</i>
PPA	Plano Plurianual Participativo
RAM	<i>Random Access Memory</i>
SMTP	<i>Simple Mail Transfer Protocol</i>
SGPR	Secretaria Geral da Presidência da República
SNPS	Secretaria Nacional de Participação Social
SQL	<i>Structured Query Language</i>
XML	<i>Extensible Markup Language</i>

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
\in	Pertence

Sumário

1	REFERENCIAL TEÓRICO	25
1.1	Conceitos Gerais	25
1.1.1	Aplicações <i>Desktop</i>	25
1.1.2	Aplicações <i>Web</i>	26
1.1.2.1	Modelo Cliente/Servidor	26
1.2	Componentes Gerais de uma Aplicação <i>Web</i>	27
1.2.1	<i>Front-end</i>	27
1.2.2	<i>Back-end</i>	27
1.2.3	Banco de Dados	28
1.2.3.1	Banco de Dados Relacionais	28
1.2.3.2	Transações ACID	28
1.2.3.3	<i>PostgreSQL</i>	29
1.2.3.4	<i>Redis</i>	29
1.3	Padrão de Projeto <i>Model-View-Controller</i>	29
1.3.1	<i>Model</i>	29
1.3.2	<i>View</i>	30
1.3.3	<i>Controller</i>	30
1.4	<i>Ruby on Rails</i>	31
1.4.1	<i>Active Record</i>	31
1.4.1.1	<i>Active Record como uma interface de ORM</i>	31
1.4.2	<i>Active Model</i>	31
1.4.3	<i>Action View</i>	32
1.4.3.1	<i>Templates</i>	32
1.4.3.2	<i>Partials</i>	32
1.4.4	<i>Action Controller</i>	32
1.5	Performance e Escalabilidade	32
1.6	Cache com <i>Ruby on Rails</i>	33
1.6.1	Tipos de Cache no <i>Ruby on Rails</i>	33
1.6.1.1	Page Caching	34
1.6.1.2	Action Caching	34
1.6.1.3	Fragment Caching	34
1.6.1.4	Russian Doll Caching	35
1.6.2	Bases de dados para Cache no <i>Ruby on Rails</i>	35
1.6.2.1	Memory Store	36
1.6.2.2	File Store	36

1.6.2.3	Mem Cache Store	37
1.6.2.4	Redis Cache Store	37
1.7	Considerações Finais do Capítulo	37
2	PROPOSTA	39
2.1	Contextualizando a Plataforma Brasil Participativo	39
2.2	Aspectos Técnicos da Plataforma Brasil Participativo	40
2.2.1	Como foi Desenvolvida a Plataforma Brasil Participativo	40
2.2.2	Arquitetura do Decidim	40
2.2.2.1	Participatory Spaces	41
2.2.2.2	Components	42
2.3	42
2.4	Principais Problemas de Performance da Plataforma Brasil Participativo	42
2.5	Proposta de Intervenção na Plataforma Brasil Participativo	42
	REFERÊNCIAS	43
	APÊNDICES	45
	APÊNDICE A – PRIMEIRO APÊNDICE	47
	APÊNDICE B – SEGUNDO APÊNDICE	49
	ANEXOS	51
	ANEXO A – PRIMEIRO ANEXO	53
	ANEXO B – SEGUNDO ANEXO	55

1 Referencial Teórico

Este capítulo possui como propósito expor e definir tópicos pertinentes sobre o assunto abordado nesse trabalho, a fim de trazer maior contexto e entendimento sobre o universo do assunto. Os conceitos que aqui serão abordados são: Conceitos Gerais (Aplicações *Standalone* e Aplicações *Web*), Componentes Gerais de uma aplicação *Web* (*Front-end*, *Back-end* e Banco de Dados), padrão de projeto *Model View Controller* (MVC), *Ruby on Rails*, Escalabilidade e performance, e Cache com *Ruby on Rails*.

Com a clarificação desses assuntos, será traga maior fundação teórica para que se siga com a pesquisa e com o caso de estudo, possibilitando assim maior embasamento durante as discussões referentes à problemática a ser estudada.

1.1 Conceitos Gerais

A seção Conceitos Gerais tem como objetivo introduzir conceitos fundamentais no escopo deste trabalho, a saber Aplicações *Standalone* e Aplicações *Web*. A apresentação destes dois tópicos permite com que sejam destacados aspectos pertinentes ao funcionamento de um *software*, bem como a aplicação Brasil Participativo, através da comparação direta desses dois principais modelos de *software* que operam no dia a dia, desde o computador pessoal de um estudante até os grandes *mainframes* do mercado.

1.1.1 Aplicações *Desktop*

Aplicações *Desktop*, também chamadas de aplicações *standalone* são softwares que atingem seu proposito de operacionalidade sem a necessidade de estarem conectados à internet, isto é, estas aplicações rodam em um computador ou dispositivo local que não necessariamente estão conectados à *web*. Uma aplicação *desktop* é autocontida, portanto, todo o código e os recursos utilizados por ela se encontram presentes no dispositivo que a executa.

Geralmente, aplicações *desktop* são específicas à plataforma no qual foi projetada para operar, seja Windows, MacOS ou Linux. Dessa maneira, outro conceito pertinente que surge em meio às aplicações *desktop* é a portabilidade. Portabilidade se refere à possibilidade de uma aplicação ser executada em diferentes ambientes com a mínima quantidade de ajustes que for necessária. Um *software* dito portátil pode executar, por exemplo, em Windows e MacOS com pouca ou nenhuma adaptação em seu código fonte, enquanto que, um software dito pouco portátil, pode precisar ser reescrito do absoluto zero

ou ter boa parte do seu código fonte adaptado para rodar em uma plataforma diferente da qual foi primariamente projetado (TANENBAUM; KLINT; BOHM, 1978).

1.1.2 Aplicações Web

Aplicações *Web*, diferentemente de aplicações *desktop*, são executadas no dispositivo do usuário através de outra aplicação, o *Web Browser*. Aplicações *Web* funcionam através do uso do modelo cliente/servidor, onde um dispositivo remoto, que possui os recursos da aplicação, os fornece para o dispositivo cliente, que serão exibidos para o usuário pelo *browser*. Para que uma aplicação *web* execute, é necessário que o dispositivo cliente esteja conectado à internet, ou, no mínimo à uma rede que tenha acesso ao dispositivo servidor. Geralmente, a conexão estabelecida no modelo cliente/servidor é através do protocolo *Hypertext Transfer Protocol* (HTTP) ou sua variante *Hypertext Transfer Protocol Secure* (HTTPS) (CONALLEN, 1999).

1.1.2.1 Modelo Cliente/Servidor

No modelo cliente/servidor, diferentemente de aplicações *desktop* convencionais, o processamento de dados é feito no lado servidor. Portanto, na maioria das vezes, o lado cliente fica isento do processamento pesado de dados, sendo responsável apenas por exibir o resultado final ao usuário. Em geral, o fluxo seguido por uma aplicação cliente/servidor é iniciado pelo dispositivo cliente (usuário final), que solicita ao dispositivo servidor algum recurso; o servidor processa a requisição do cliente realizando o processamento necessário, e em seguida, devolve o recurso em um formato que o lado cliente consiga processar e exibir ao usuário final. Essa comunicação pode ocorrer através de diversos protocolos: *File Transfer Protocol* (FTP) para transferência de arquivos, *Simple Mail Transfer Protocol* (SMTP) para envio e recebimento de e-mails, e o já mencionado HTTP.

Cada protocolo dispõe de particularidades que derivam dos seus propósitos, fazendo com que cada um deles seja aplicável em um contexto específico. Entretanto, uma aplicação pode se beneficiar do uso de vários protocolos diferentes para cada uma de suas funcionalidades, isso a depender do propósito de cada um de seus módulos. Uma aplicação *web* moderna, geralmente dispõe de funcionalidades para gerenciar recursos e exibi-los aos usuários utilizando o protocolo HTTP, em contrapartida, a própria entidade usuário pode ser considerada um recurso, onde, cada usuário tem um e-mail vinculado, permitindo que a aplicação envie informações utilizando o protocolo SMTP em algum outro momento oportuno.

A adoção do modelo cliente/servidor pode proporcionar à aplicação *web* diversas características importantes, sendo a principal delas a centralização dos dados no lado servidor. Uma vez que os dados estejam centralizados no lado servidor, máquinas com mais recursos computacionais podem ser alocadas para que o processamento ocorra mais

rapidamente e de forma desacoplada do ambiente do lado cliente, permitindo maior portabilidade da aplicação (OLUWATOSIN, 2014).

1.2 Componentes Gerais de uma Aplicação Web

Uma aplicação *web* moderna em geral pode ser vista dividida em duas partes: *back-end* e *front-end*. Ao contrário de aplicações *standalone* convencionais que possuem a lógica de negócio altamente acoplada com a lógica de exibição, aplicações *web* tendem a ter uma divisão mais clara nesse aspecto, permitindo com que a lógica de negócio não necessariamente interfira na renderização. Vale notar ainda a grande semelhança dessa abordagem com o modelo cliente/servidor (GONG et al., 2020). Outro importante componente no contexto de aplicações *web* é o banco de dados, que cumpre o papel de persistir informações da aplicação. A seção Componentes Gerais de uma Aplicação Web possui como objetivo introduzir cada um desses conceitos e elucidar seus papéis, trazendo maior clareza para cada um de seus papéis e como estes influenciam no funcionamento da aplicação *web*, e consequentemente na sua performance.

1.2.1 Front-end

O *front-end* é a parte da aplicação *web* responsável pela exibição dos dados e da interface de usuário. Idealmente, o *front-end* lida apenas com a renderização de páginas, a lógica de *design* da aplicação, sistema de rotas de páginas e recursos estáticos (GONG et al., 2020).

1.2.2 Back-end

O *back-end* é a parte da aplicação *web* que lida com toda a lógica de negócio contida no escopo do *software*. Todos os dados, bem como o processamento desses dados é realizado nessa camada. Do ponto de vista do funcionamento da aplicação como um todo, o *back-end* possui como trabalho responder as requisições do usuário que vêm da camada do *front-end*. Uma vez que o usuário pode enviar e requisitar dados, o *back-end* deve ser capaz, através de um conjunto de métodos, processar a requisição e devolver uma resposta no menor tempo possível (ADAM; BESARI; BACHTIAR, 2019).

A comunicação entre *back-end* e *front-end* pode ser realizada de diversas maneiras e com vários protocolos. Uma das formas mais adotadas como mecanismo de comunicação em aplicações *web* modernas é a exposição de funções do *back-end* por meio de *Application Programming Interfaces* (APIs). As APIs podem ser vistas como interfaces de comunicação que visam abstrair a lógica de funcionamento por detrás de uma aplicação, expondo suas funcionalidades através de *endpoints* que podem ser acessados por

outras aplicações sem que a aplicação cliente precise saber como estas operam (GOUGH; BRYANT; AUBURN, 2021).

1.2.3 Banco de Dados

Banco de Dados, nesse contexto, uma forma reduzida do termo Sistema Gerenciador de Banco de Dados, é uma classe de *software* que lida com a manipulação e persistência de dados. Um banco de dados tem como objetivo fornecer aos seus usuários uma fonte de dados centralizada, com qualidade, integridade e segurança (FRY; SIBLEY, 1976). Existem diversos tipos de bancos de dados, sendo revelantes para este trabalho: bancos de dados relacionais e bancos de dados de chave-valor.

1.2.3.1 Banco de Dados Relacionais

mas o foco deste trabalho estará nos bancos de dados relacionais. Bancos de dados relacionais são bancos cujo os dados são guardados no formato de tabelas, também denominadas relações. As tabelas são compostas por colunas e linhas, semelhante a planilhas, onde cada linha representa um registro, denominado tupla, na tabela.

Bancos de dados relacionais permitem a criação de relacionamentos entre tabelas, sendo essa a motivação do termo "relacional". A maioria dos bancos de dados permitem manipular e consultar seus dados através do uso da linguagem de consulta *Structured Query Language* (SQL) (JATANA et al., 2012).

1.2.3.2 Transações ACID

No contexto de banco de dados, um importante conceito é o de transações. Uma transação é um agrupamento de operações realizadas pelo banco de dados, sendo a transação um conceito atômico, isto é, a menor instrução que o banco de dados realizará a mando do usuário. Existem características que um banco de dados pode ou não respeitar na sua implementação, estas são: atomicidade, consistência, isolamento e durabilidade (ACID). As propriedades ACID podem ser definidas da seguinte maneira:

- Atomicidade: as operações envolvidas em uma única transação são executadas como uma só, implicando em que, caso uma falhe, a transação como um todo será dada como falha.
- Consistência: os dados presentes no banco de dados devem permanecer consistentes após a execução da transação.
- Isolamento: estados intermediários da transação não são visíveis por outras transações concorrentes, implicando que, uma transação não interferirá em outra.

- Durabilidade: quando uma transação é executada e chega ao fim, seus efeitos são persistentes. Caso ocorra interrupções ou falhas no banco de dados, uma transação completa não será afetada.

(YU, 2009).

1.2.3.3 *PostgreSQL*

Segundo a documentação, o PostgreSQL é um gerenciador de banco de dados relacional *open-source* que implementa e incrementa o padrão da linguagem SQL. O PostgreSQL implementa todos os princípios ACID desde 2001, além de oferecer *plugins* que permitem o uso de outras funcionalidades não convencionais no contexto de banco de dados relacionais (GROUP, 2023).

1.2.3.4 *Redis*

Bancos de dados chave-valor são sistemas gerenciadores de banco de dados que armazenam seus dados de forma não relacional utilizando de chaves e valores. Cada registro consistirá em uma chave e em um valor, podendo este ser armazenado tanto na memória RAM, quanto no disco. Uma das principais características de bancos de dados chave-valor é a sua velocidade quando comparado com bancos relacionais. O *Redis* é um banco de dados chave valor *open-source* extremamente rápido que guarda seus dados em forma persistente no disco ou na memória RAM (CARLSON, 2013).

1.3 Padrão de Projeto *Model-View-Controller*

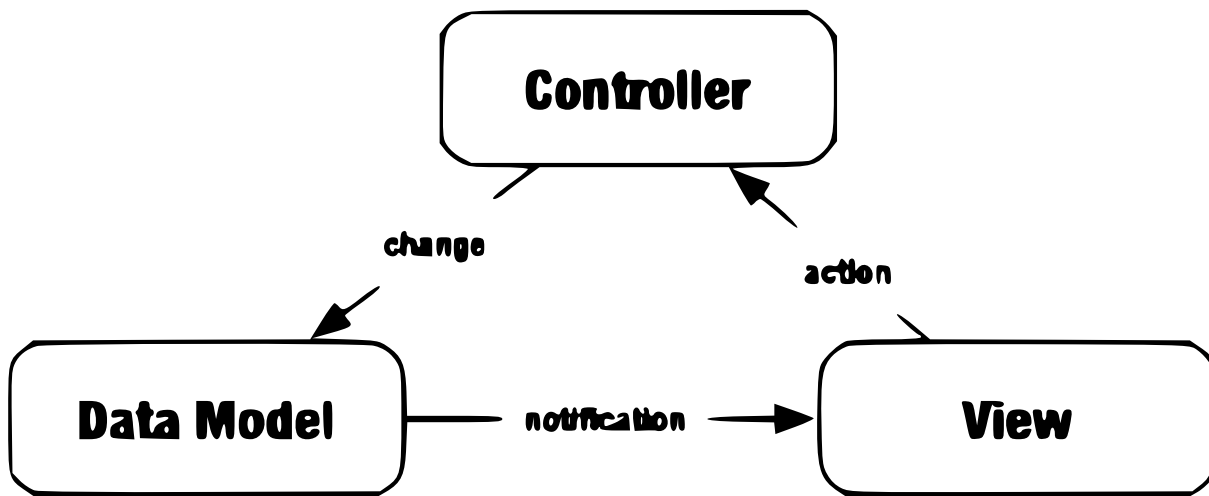
O MVC é um padrão de projeto que é considerado como um dos mais importantes na área da ciência da computação, pois este pode ser interpretado como algo mais próximo de uma filosofia de desenvolvimento de arquitetura de código, do que de um padrão de projeto que resolve instâncias de problemas de um domínio específico. O MVC possui uma grande abertura de interpretação e aplicação, podendo ser adotado desde subsistemas até aplicações inteiras. Essencialmente, o MVC é definido em três entidades: a *model*, a *view*, e a *controller* (MODEL-VIEW-CONTROLLER..., 2009).

As próximas subseções tem como objetivo descrever cada uma dessas entidades e qual papel desempenham dentro do contexto do padrão de projeto MVC.

1.3.1 *Model*

A *model* é a entidade do MVC que possui como preocupação encapsular informações do domínio do problema a ser resolvido pela aplicação. Um objeto de dados *model* deve saber guardar, encapsular e abstrair os dados. Vale observar também que

Figura 1 – Comunicação entre as entidades do MVC



Fonte: (MODEL-VIEW-CONTROLLER..., 2009)

cada entidade idealmente não deve extrapolar suas funções, portanto, uma *model* deve saber serializar seus dados, mas não deve saber como implementar uma funcionalidade de "salvar como" (MODEL-VIEW-CONTROLLER..., 2009).

1.3.2 View

Em poucas palavras, a *view* é a entidade do MVC que apresenta informações na tela para o usuário final. Porém, a *view* desempenha muitos outros papéis; na verdade, ela é a ponte entre o usuário e o sistema. Dessa forma, a *view* é responsável por captar o *input* do usuário e transformá-lo em comandos para o sistema processar. Outro aspecto importante é o escopo que uma *view* pode atuar. No desenvolvimento de aplicações com o modelo MVC, objetos de *view* podem ser generalistas ao ponto de apenas saberem renderizar e lidar com textos, imagens e mídias, ou podem ser específicos ao ponto de saber renderizar apenas objetos específicos do sistema definidos no código pelo programador (MODEL-VIEW-CONTROLLER..., 2009).

1.3.3 Controller

A *controller* é a entidade do MVC que implementa as ações do sistema. Como já mencionado, as *models* sabem gerenciar seus dados a nível de persistência e abstração, as *views* são responsáveis por coletar o *input* do usuário para disparar as ações, estas que residem nas *controllers*. Uma ação realizada por uma *controller*, por exemplo, seria o comando "salvar como" de um dado armazenado dentro de uma *model* (MODEL-VIEW-CONTROLLER..., 2009). Em geral, a comunicação realizada pelas entidades no modelo MVC pode ser visualizada na Figura 1.

1.4 Ruby on Rails

O *Ruby on Rails*, ou simplesmente *Rails*, é um *framework* de aplicação *web* que fornece tudo o que é necessário para a criação de aplicações que seguem o modelo MVC e que utilizam de bancos de dados para armazenamento de suas estruturas e dados. O *Rails* traz para cada uma das entidades do modelo MVC uma interface que provê as funcionalidades necessárias para que estas sejam implementadas em uma aplicação *web*.

O *Rails* traz duas interfaces para que o desenvolvedor consiga criar as *models* no domínio de sua aplicação: o *Active Record* e o *Active Model*. Para as *views*, o *Rails* fornece a *Action View*, que disponibiliza funcionalidades de geração de novas *views* na aplicação. Para as *controllers*, o *Rails* fornece o *Action Controller*. Cada uma dessas interfaces são exploradas em mais detalhes a seguir ([RUBY..., 2023](#)).

1.4.1 Active Record

O *Active Record* é o "M" do modelo MVC no contexto do *Rails*. De fato, o *Active Record* é um padrão de projeto que precede o *Rails*, este foi descrito por *Martin Fowler* em seu livro *Patterns of Enterprise Application*. Porém, no contexto de aplicações *Rails*, o *Active Record* é caracterizado por sua responsabilidade de representar regras de negócio e dados, sendo uma ponte direta entre as *models* da aplicação e o banco de dados, disponibilizando uma interface de *Object Relational Mapping* (ORM) ([ACTIVE..., 2023b](#)).

1.4.1.1 Active Record como uma interface de ORM

O *Active Record* provê, por padrão, uma interface de comunicação entre os objetos da aplicação com bancos de dados relacionais. Esta interface permite que o programador não necessite de escrever instruções SQL diretamente no código fonte da aplicação para buscar e persistir dados das *models* no banco de dados ([ACTIVE..., 2023b](#)). Além de prover esse mecanismo de interação com o banco, o *Active Record* fornece diversas outras funcionalidades que permitem a configuração de validações sobre atributos da *model*, além de prover mecanismos de *callbacks* que são ativados automaticamente para os eventos básicos do ciclo de vida do objeto: criar, salvar, atualizar e destruir.

1.4.2 Active Model

O *Active Model* possibilita incorporar funcionalidades do *Active Record* em uma classe *Ruby* pura, sem a intenção de persistir seus atributos no banco de dados. O *Rails* provê maneiras de se incluir separadamente cada uma dessas funcionalidades, permitindo com que a classe herde apenas os comportamentos desejados do *Active Record* ([ACTIVE..., 2023a](#)).

1.4.3 Action View

As *Action Views* no *Rails* são usadas para renderizar o resultado de uma ação ou requisição processada pelo sistema. Em termos dos fundamentos de uma aplicação cliente/servidor, a função de uma *Action View* é processar as informações a serem retornadas pelo sistema e compilá-las em um formato compreensível pelo usuário. O *Rails* facilita essa renderização por meio de três componentes chave: *templates*, *partials* e *layouts*.

1.4.3.1 Templates

Os *templates* atuam como um guia para a exibição das informações resultantes de uma ação. Por padrão, um *template* da *Action View* pode ser renderizado em diversos formatos, destacando-se, entre eles, o *HyperText Markup Language* (HTML), *JavaScript Object Notation* (JSON) e *Extensible Markup Language* (XML).

1.4.3.2 Partials

As *partials*, ou o termo mais completo *template partials*, são exatamente o que o nome sugere: *templates* parciais, ou seja, fragmentos de código que são extraídos para um arquivo separado e podem ser reutilizados em diversas partes. O propósito das *partials* é dividir o processo de renderização em partes para um melhor controle e permitir a reusabilidade, o que fomenta a filosofia do *Don't Repeat Yourself* (DRY) (ACTION..., 2023b).

1.4.4 Action Controller

O *Action Controller* é uma parte essencial do padrão de arquitetura MVC, representando a letra "C". Ele gerencia as solicitações, processa dados provenientes da *model* e gera a saída apropriada para exibição ao usuário. Em aplicações *Rails* convencionais, o *Action Controller* recebe a solicitação, interage com a *model* para obter ou salvar dados, e utiliza a *view* para criar uma saída em HTML ou qualquer outro formato necessário. Ou seja, o *Action Controller* atua como intermediário entre os dados das *models* e a apresentação ao usuário por meio das *Action Views*. No *Rails*, é possível definir *actions* dentro das *controllers* por meio de funções que recebem os dados da requisição e realizam as operações necessárias. Cada *action* tem um *template* de *view* correspondente, fortalecendo a cooperação entre o *Action Controller* e a *Action View* (ACTION..., 2023a).

1.5 Performance e Escalabilidade

A performance de uma aplicação *web* pode ser avaliada de várias maneiras. Duas das maneiras mais conhecidas de aferir o nível de performance de uma aplicação é através

do número de requisições processadas por segundo, também chamado de *throughput* ou *requests/second*, e o tempo gasto pela aplicação para responder a uma requisição. Uma boa aplicação visa manter seu *throughput* alto e seu *response time* baixo, de tal forma que a experiência do usuário seja positiva, a carga de uso seja suportada e a aplicação seja confiável em momentos de pico de acesso, evitando indisponibilidade. Sabe-se que 80% do tempo gasto por uma aplicação no processamento de uma requisição é na execução operações no banco de dados, com consultas SQL, e na montagem da resposta para o usuário, com a renderização de *templates* e criação do conteúdo compilado. (JUGO; KERMEK; MEŠTROVIĆ, 2014).

A aplicação de soluções web em diversas áreas de negócio, como vendas online, destaca o conceito de escalabilidade em contextos de software. Com o aumento da demanda e do acesso de usuários, é crucial que uma aplicação web possa expandir-se mediante a adição de recursos computacionais. Embora não haja um consenso claro sobre a definição de escalabilidade, geralmente é compreendida como a capacidade de uma aplicação aumentar seu *throughput* por meio da incorporação eficiente de recursos de hardware. Essa capacidade é uma propriedade do sistema de software, fortemente influenciada pela arquitetura adotada. Se a arquitetura não permitir o aumento do *throughput* em um determinado nível de demanda com a incorporação de mais recursos, ela é considerada não escalável. (WILLIAMS; SMITH, 2004).

1.6 Cache com Ruby on Rails

Em aplicações web, especialmente no contexto do *Rails*, o conceito de cache refere-se à prática de armazenar conteúdos gerados durante o processamento de resposta a requisições para posterior utilização em requisições subsequentes. O *Ruby on Rails* fornece em sua API uma interface única e genérica que possibilita maneiras de integrar-se com sistemas terceiros que atuam como bases de dados para armazenamento de cache. Isso oferece ao programador a flexibilidade de escolher estratégias de como um dado específico será armazenado, tornando também a tarefa de definir a expiração dos dados armazenados mais simples, algo que, quando feito manualmente, é bastante suscetível a erros. Na seção 1.6.1 é explorada as maneiras que o *Rails* permite com que o programador utilize sua interface de cache, e na seção 1.6.2 são explicadas as principais interfaces disponibilizadas pelo *Rails* para comunicação com bases de dados para *cache* (RAILS, 2023c).

1.6.1 Tipos de Cache no Ruby on Rails

O *Ruby on Rails* fornece maneiras de se utilizar cache com funcionalidades embutidas, ou com *gems* que se integram com o *framework* sem muita dificuldade. A seguir são descritas cada uma dessas maneiras com suas particularidades, pontos positivos e

negativos.

1.6.1.1 Page Caching

O *page caching* é uma estratégia de cache fornecida pela *gem actionpack-page_caching*, que consiste em salvar o conteúdo de uma requisição em um arquivo estático que será servido como resultado nas próximas requisições. Dessa maneira, uma vez salvo o resultado de uma requisição específica, quando realizada uma nova requisição ao mesmo *endpoint*, o arquivo salvo será servido pelo servidor, de tal forma que a requisição sequer será processada pela aplicação *Rails*. Essa abordagem apesar de simples, possui potencial para reduzir bastante o *response time* da aplicação para os *endpoints* que forem armazenados.

Porém, esta estratégia funciona apenas para requisições *get* e *head* com retorno de código 200. A utilização desse mecanismo de cache também não é recomendada para páginas que requeiram autenticação, pois a aplicação *Rails* precisa decidir se o remetente da requisição está ou não autorizado a receber o conteúdo solicitado (RAILS, 2023b).

A *gem actionpack-page_caching* está disponível em seu repositório oficial no endereço https://github.com/rails/actionpack-page_caching.

1.6.1.2 Action Caching

O *action caching* é similar ao *page caching* no sentido de que a resposta inteira será armazenada. Essa estratégia de cache é fornecida pela *gem actionpack-action_caching*. Entretanto, nessa abordagem a requisição passa a ser recebida pela aplicação *Rails*, ao ponto que os filtros são executados antes do cache ser servido. Isto é, caso o *endpoint* consultado requeira autenticação para decidir se um determinado recurso será ou não servido, por exemplo, as verificações serão realizadas na *controller*, para só então o conteúdo em cache ser servido.

O *action caching* permite que o programador especifique diretamente o tempo de vida de um armazenamento de cache a nível de *actions* dentro da *controller*, além de permitir especificar condições para decidir se o resultado de uma *action* será ou não servido por cache (RAILS, 2023a).

1.6.1.3 Fragment Caching

Aumentando ainda mais o nível de granularidade do armazenamento de informação em cache, o *Rails* fornece um mecanismo chamado *fragment caching*. Com o *fragment caching* é possível armazenar fragmentos específicos do *template* de uma *view*, ao invés de armazená-la por inteira. Essa abordagem é especialmente utilizada em trechos da *view* onde objetos de *models* são renderizados.

Figura 2 – Exemplo de *template* que renderiza objetos com *caches* aninhados

```
1 <!-- No template utilizado para renderização do objeto product -->
2 <% cache product do %>
3   <%= render product.games %>
4 <% end %>
5
6 <!-- No template utilizado para renderização do objeto `game` -->
7 <% cache game do %>
8   <%= render game %>
9 <% end %>
10
```

Fonte: (RAILS, 2023c)


Quando o *Rails* se depara com um fragmento de *view* que exibirá informações de um objeto de uma *model*, e que há a instrução de armazenar o resultado em *cache*, uma chave única será criada com base na junção do nome da *view*, o nome da *model* do objeto em questão, o *id* e o atributo *updated_at* do objeto, que representam o a chave primária e o momento em que o objeto foi atualizado pela última vez, respectivamente. Caso não exista um valor já armazenado em *cache* para esta chave, o fragmento será renderizado pela aplicação e o resultado será armazenado na memória com associação direta à essa chave (RAILS, 2023c).

1.6.1.4 Russian Doll Caching

Ao utilizar a estratégia de *fragment caching*, nos cenários onde existem objetos cuja renderização acontece de forma aninhada com outros objetos em uma relação de *has_many/belongs_to*, se um dos objetos aninhados for alterado, ainda assim o fragmento armazenado em cache não expirará, pois o objeto mais externo não terá seu atributo *update_at* atualizado, conforme a Figura 2. Para resolver esse problema, o *Rails* oferece uma estratégia de armazenamento de cache chamada *Russian Doll Caching*. Essa estratégia consiste em adicionar no código da *model* aninhada a opção *touch: true* conforme ilustrado na Figura 3. Dessa maneira, sempre que um objeto aninhado for atualizado, o objeto mais externo também marcado como atualizado (RAILS, 2023c).

1.6.2 Bases de dados para Cache no Ruby on Rails

Como mencionado anteriormente, o *Ruby on Rails* provê uma interface para comunicação com diversas tecnologias de armazenamento de *cache*. Ele permite com que o programador especifique qual tecnologia deve ser utilizada na aplicação, e fornece uma interface que provê a fundação para interação com o sistema de *cache*. Essa interface é o *ActiveSupport::Cache::Store*, que fornece métodos básicos para o programador: *read*,

Figura 3 – Inclusão da opção *touch* na declaração de relacionamento entre *models*

```
1 class Product < ApplicationRecord
2   has_many :games
3 end
4
5 class Game < ApplicationRecord
6   belongs_to :product, touch: true
7 end
8
```

Fonte: (RAILS, 2023c)

write, *delete*, *exist?*, e *fetch*. Existem implementações dessa interface providas pelo *Rails*, chamadas de *Cache Store*, cujo propósito é operar com tecnologias de armazenamento conhecidas. Essas implementações são: o *Memory Store*, o *File Store*, o *Mem Cache Store*, o *Redis Cache Store*, e o *Null Store*; além de permitir o uso de *Cache Stores* customizados.

1.6.2.1 Memory Store

O *Memory Store* é uma tecnologia de armazenamento que utiliza a memória do próprio processo *Ruby* em execução para guardar os registros. Esse *cache store* possui as seguintes propriedades:

- O tamanho do armazenamento é fixo, e deve ser especificado na configuração da aplicação;
- Sempre que faltar espaço para inserir novos registros, uma limpeza será realizada removendo os registros que não foram utilizados recentemente;
- No caso da aplicação estar rodando em múltiplos processos, um processo não terá acesso ao *cache* do outro.

1.6.2.2 File Store

O *File Store*, como o próprio nome sugere, utiliza o próprio sistema de arquivos para guardar os registros de *cache*. Esse *cache store* possui as seguintes propriedades:

- O caminho do diretório utilizado para *cache* é especificado na configuração da aplicação;
- No caso da aplicação estar rodando em múltiplos processos no mesmo *host*, todos eles terão acesso ao mesmo cache.
- O volume de dados utilizado pelo armazenamento de *cache* cresce até que o disco se encontre cheio, sendo necessária a limpeza de registros antigos.

1.6.2.3 Mem Cache Store

O *Mem Cache Store* é uma tecnologia de armazenamento de *cache* baseada no *memcached server* da *Danga*. O *Rails* utiliza a *gem dalli* para operar com esse sistema de cache, por padrão. Esse *cache store* possui a seguinte propriedade:

- É possível utilizar um *cluster* de servidores *memcached*, porém, esses devem ser especificados na configuração da aplicação ou via variável de ambiente.

1.6.2.4 Redis Cache Store

O *Redis Cache Store*, como o próprio nome sugere, é um *cache store* que utiliza o *Redis* como tecnologia de armazenamento de cache. Esse *cache store* possui as seguintes propriedades:

- Dada a natureza do uso, é recomendado que se utilize um servidor *Redis* dedicado para *cache*.
- Servidores de *cache Redis* permitem utilizar políticas de expiração de registros por: registro utilizado com menor frequência, ou registro que não é utilizado a mais tempo.
- O *Redis Cache Store* permite especificar parâmetros como: *timeout* para conexão, *timeout* para leitura, *timeout* para escrita e tentativas de reconexão.
- O tempo gasto para criar ou reescrever um registro em *cache* é pequeno, sendo as vezes mais vantajoso reescrever um registro do que esperar muito tempo para buscá-lo.

1.7 Considerações Finais do Capítulo

Este capítulo visou esclarecer conceitos relevantes sobre aplicações web e seu funcionamento, a fim de permitir ser introduzida de maneira clara a abordagem de otimização de desempenho da plataforma Brasil Participativo. Essa plataforma foi desenvolvida em

Ruby on Rails e, atualmente, utiliza o PostgreSQL como recurso tecnológico para bancos de dados e o Redis para cache.

Para compreender os desafios enfrentados pela plataforma e perceber a aplicabilidade das soluções propostas, é fundamental entender o funcionamento de aplicações cliente/servidor e o fluxo de operação de uma aplicação MVC, especialmente no contexto do *framework Ruby on Rails*. Como observado, grande parte do tempo consumido por uma aplicação web está relacionada à execução de operações no banco de dados e à compilação de dados para fornecer uma resposta ao cliente, etapas essas que podem ser otimizadas com a utilização de recursos de cache fornecidos pelo próprio *framework*.

2 Proposta

Este capítulo tem como objetivo apresentar a proposta de intervenção na plataforma Brasil Participativo, esclarecendo os detalhes relevantes sobre as estratégias de otimização que podem vir a serem adotadas. A Seção ?? explica o que é a plataforma, sobre como ela nasceu, seus objetivos futuros e os já alcançados. A Seção ?? aborda os aspectos técnicos da plataforma em nível de código-fonte e arquitetura, revelando o funcionamento das principais partes e o uso do *framework Ruby on Rails*. Na Seção 2.4, são expostos os desafios enfrentados pela plataforma no ano de 2023, explorando as possibilidades de intervenção para otimização. A Seção 2.5 tem como objetivo listar potenciais intervenções para melhorar a performance, esclarecendo a abordagem a ser adotada e o grau de esforço envolvido.

2.1 Contextualizando a Plataforma Brasil Participativo

A Plataforma Brasil Participativo é uma iniciativa do governo federal voltada para a promoção da participação social. Desenvolvida em software livre com o apoio de diversos parceiros, incluindo a Dataprev, a comunidade Decidim-Brasil, o Ministério da Gestão e Inovação em Serviços Públicos (MGI) e a Universidade de Brasília (UnB), ela foi criada sob a responsabilidade da Secretaria Nacional de Participação Social da Secretaria Geral da Presidência da República (SNPS/SGPR).

A plataforma tem como propósito possibilitar que a população contribua ativamente na criação e melhoria das políticas públicas. Uma de suas primeiras iniciativas foi o Plano Plurianual Participativo, assinado pela SGPR e pelo Ministério do Planejamento e Orçamento (MPO). Durante o período de 11 de maio a 16 de julho de 2023, a plataforma permitiu a coleta de propostas da sociedade e a priorização de programas e propostas para o Plano Plurianual (PPA) 2024-2027.

A participação ativa na etapa digital do PPA atingiu mais de um milhão e quatrocentas mil pessoas (1.400.000), conquistando o título de maior experiência de participação social na internet realizada pelo governo federal. A plataforma continuará evoluindo, permitindo que conselhos nacionais criem suas páginas, ministérios realizem consultas públicas e órgãos federais promovam a participação da população na definição de decretos, portarias e outras ações.

Essa abertura à participação digital representa um marco importante para a democracia, possibilitando que cidadãos influenciem diretamente nas decisões governamentais. A Plataforma Brasil Participativo oferece a oportunidade de criação de perfis individuais

para facilitar a participação ativa dos interessados ([PARTICIPATIVO, 2023](#)).

2.2 Aspectos Técnicos da Plataforma Brasil Participativo

Nesta seção será abordada a maneira como foi desenvolvida a plataforma Brasil Participativo utilizando-se do *framework Ruby on Rails* e da *gem Decidim*. O objetivo dessa seção é elucidar como a plataforma foi inicialmente desenvolvida, o atual estado, e explorar em detalhes a *gem Decidim*.

2.2.1 Como foi Desenvolvida a Plataforma Brasil Participativo

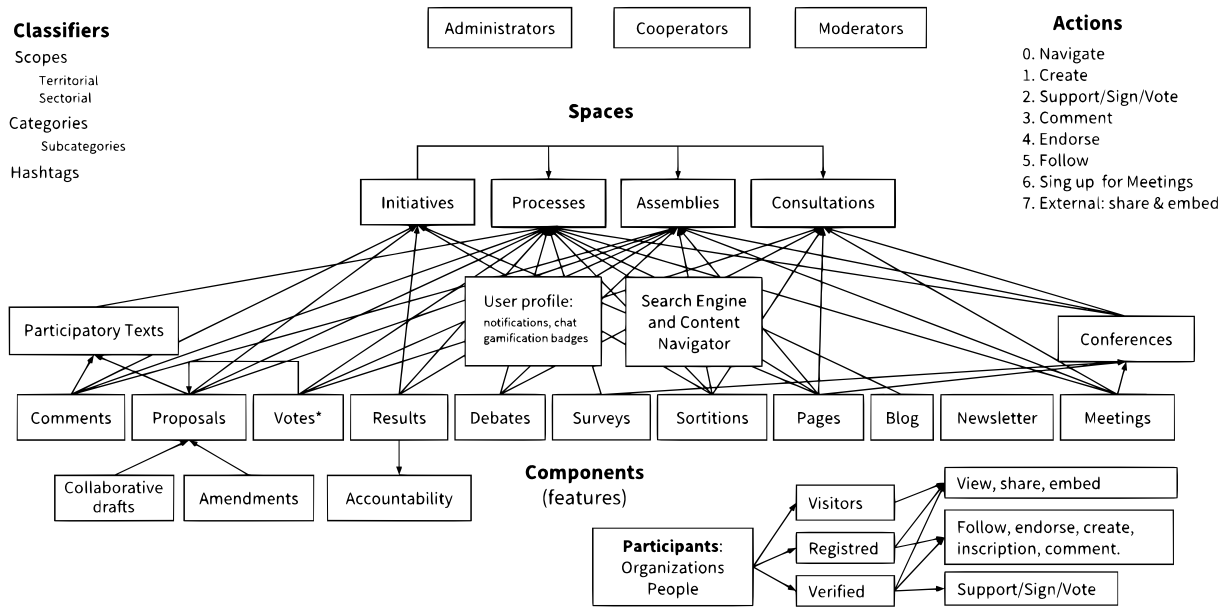
A plataforma Brasil Participativo foi desenvolvida com o *framework Ruby on Rails* utilizando a *gem Decidim*. Segundo a própria documentação disponibilizada pela comunidade de contribuidores do *Decidim*, a *gem* é descrita como um *framework* que visa possibilitar que qualquer pessoa possa criar uma plataforma *web* para ser utilizada como rede política para participação democrática. O *Decidim* fornece a organizações a capacidade de criar processos para planejamento estratégico, orçamento participativo, concepção colaborativa de regulamentos, espaços urbanos e processos eleitorais ([DECIDIM, 2023a](#)).

A plataforma no presente momento se apoia totalmente nas funcionalidades disponibilizadas pela *gem Decidim*, preocupando-se apenas em implementar em sua instância correções de *bugs*, customizações de estilo e comportamento de regras de negócio, e funcionalidades extras. De fato, a maior parte do código executado na plataforma não é visto em seu repositório oficial, pois este está abstraído dentro da *gem Decidim*. O repositório oficial do Brasil Participativo está disponível no *Gitlab* no endereço: <https://gitlab.com/lappis-unb/decidimbr/decidim-govbr>. Já o repositório oficial do *Decidim* está disponível no *Github* no endereço: <https://github.com/decidim/decidim>.

2.2.2 Arquitetura do Decidim

O *Decidim* possui várias entidades em sua arquitetura, mas as mais importantes para entendimento da regra de negócio são os espaços participativos e os componentes, referidos na documentação como *participatory spaces* e *components*, respectivamente. Uma visão macro das entidades do *Decidim* pode ser visto na figura 4. O *Decidim* armazena dados de cada uma dessas estruturas utilizando a abordagem padrão de aplicações *Rails like*, isto é, utilizando-se do *Active Record* e persistindo os dados em um banco de dados relacional, o *PostgreSQL*.

Figura 4 – Arquitetura do *Decidim*



Fonte: Retirado de (DECIDIM, 2023b)

Espaço Participativo	Módulo
Assembleias	decidim-assemblies
Consultas ou Eleição	decidim-elections
Iniciativas	decidim-initiatives
Processos Participativos	decidim-participatory_processes

Tabela 1 – Módulos dos espaços participativos padrões do *Decidim*

2.2.2.1 Participatory Spaces

Os *participatory spaces* são *frameworks* que definem como a participação será realizada, os canais ou meios pelos quais cidadãos ou membros de uma organização podem processar solicitações ou coordenar propostas e tomar decisões. Iniciativas, processos, assembleias e consultas são todos espaços participativos, estes referidos na documentação como *initiatives*, *participatory processes*, *assemblies* e *consultations*, respectivamente. Exemplos específicos incluem: uma iniciativa cidadã para alterar diretamente uma regulamentação; uma assembleia geral ou conselho de trabalhadores; um orçamento participativo, planejamento estratégico ou processo eleitoral; um referendo ou uma convocação para votar "Sim" ou "Não" para mudar o nome de uma organização (DECIDIM, 2023b).

A nível de código, cada implementação de um novo *participatory space* é feita em um sub-módulo separado em uma *gem* própria. Uma relação entre cada um dos espaços participativos já existentes por padrão no *Decidim* e o seu respectivo sub-módulo pode ser vista no quadro 1

Componente	Módulo
Comentários	decidim-comments
Propostas	decidim-proposals
Debates	decidim-debates
Pesquisas	decidim-surveys
Sorteios	decidim-sortitions
<i>Blogs</i>	decidim-blogs
Reuniões	decidim-meetings

Tabela 2 – Módulos dos espaços participativos padrões do *Decidim*

2.2.2.2 Components

Os *components* são os mecanismos participativos que permitem uma série de operações e interações entre os usuários da plataforma dentro de cada um dos espaços participativos. No *Decidim* há disponível por exemplo os seguintes componentes: comentários, propostas, debates, pesquisas, sorteios, blogs e reuniões. Outros componentes que se baseiam nos componentes básicos são: textos participativos, responsabilidade e conferências ([DECIDIM, 2023b](#)).

Da mesma forma que ocorre com os espaços participativos, os componentes são implementados através de sub-módulos em formato de novas *gems*. No quadro 2 é possível conferir o nome do módulo de cada componente fornecido por padrão pelo *Decidim*.

2.3

2.4 Principais Problemas de Performance da Plataforma Brasil Participativo

2.5 Proposta de Intervenção na Plataforma Brasil Participativo

Referências

ACTION Controller Overview. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: https://guides.rubyonrails.org/action_controller_overview.html. Citado na página 32.

ACTION View Overview. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: https://guides.rubyonrails.org/action_view_overview.html. Citado na página 32.

ACTIVE Model Basics. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: https://guides.rubyonrails.org/active_model_basics.html. Citado na página 31.

ACTIVE Record Basics. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: https://guides.rubyonrails.org/active_record_basics.html. Citado na página 31.

ADAM, B. M.; BESARI, A. R. A.; BACHTIAR, M. M. Backend server system design based on rest api for cashless payment system on retail community. In: IEEE. *2019 International Electronics Symposium (IES)*. [S.l.], 2019. p. 208–213. Citado na página 27.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR 14724: Informação e documentação — trabalhos acadêmicos — apresentação. Rio de Janeiro, p. 15, mar. 2011. Citado na página 3.

CARLSON, J. L. *Redis in Action*. Shelter Island, NY: Manning Publications Co., 2013. ISBN 9781935182054. Citado na página 29.

CONALLEN, J. Modeling web application architectures with uml. *Communications of the ACM*, ACM New York, NY, USA, v. 42, n. 10, p. 63–70, 1999. Citado na página 26.

DECIDIM. *Decidim Documentation*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://docs.decidim.org/en/v0.27/index.html>. Citado na página 40.

DECIDIM. *General description and introduction to how Decidim works*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://docs.decidim.org/en/v0.27/features/general-description>. Citado 2 vezes nas páginas 41 e 42.

FRY, J. P.; SIBLEY, E. H. Evolution of data-base management systems. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 8, n. 1, p. 7–42, 1976. Citado na página 28.

GONG, Y. et al. The architecture of micro-services and the separation of frond-end and back-end applied in a campus information system. In: IEEE. *2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. [S.l.], 2020. p. 321–324. Citado na página 27.

GOUGH, J.; BRYANT, D.; AUBURN, M. *Mastering API Architecture*. [S.l.]: "O'Reilly Media, Inc.", 2021. Citado na página 28.

GROUP, P. G. D. *About PostgreSQL*. 2023. Accessed on December 4, 2023. Disponível em: <https://www.postgresql.org/about/>. Citado na página 29.

JATANA, N. et al. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, v. 1, n. 6, p. 1–5, 2012. Citado na página 28.

JUGO, I.; KERMEK, D.; MEŠTROVIĆ, A. Analysis and evaluation of web application performance enhancement techniques. In: SPRINGER. *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings 14*. [S.l.], 2014. p. 40–56. Citado na página 33.

MODEL-VIEW-CONTROLLER Pattern. In: LEARN Objective-C for Java Developers. Berkeley, CA: Apress, 2009. p. 353–402. ISBN 978-1-4302-2370-2. Disponível em: https://doi.org/10.1007/978-1-4302-2370-2_20. Citado 2 vezes nas páginas 29 e 30.

OLUWATOSIN, H. S. Client-server model. *IOSR Journal of Computer Engineering*, IOSR Journals, v. 16, n. 1, p. 67–71, 2014. Citado na página 27.

PARTICIPATIVO, B. *Sobre o Brasil Participativo*. 2023. Acesso em: 10 de dezembro de 2023. Disponível em: <https://brasilparticipativo.presidencia.gov.br/processes/brasilparticipativo/f/33>. Citado na página 40.

RAILS, R. on. *Actionpack Action Caching*. 2023. Acesso em: 12 de dezembro de 2023. Disponível em: https://github.com/rails/actionpack-action_caching. Citado na página 34.

RAILS, R. on. *Actionpack Page Caching*. 2023. Acesso em: 12 de dezembro de 2023. Disponível em: https://github.com/rails/actionpack-page_caching. Citado na página 34.

RAILS, R. on. *Caching with Rails: An Overview*. 2023. Acesso em: 12 de dezembro de 2023. Disponível em: https://guides.rubyonrails.org/caching_with_rails.html. Citado 3 vezes nas páginas 33, 35 e 36.

RUBY on Rails API. 2023. Acesso em: 9 de dezembro de 2023. Disponível em: <https://api.rubyonrails.org/>. Citado na página 31.

TANENBAUM, A. S.; KLINT, P.; BOHM, W. Guidelines for software portability. *Software: Practice and Experience*, Wiley Online Library, v. 8, n. 6, p. 681–698, 1978. Citado na página 26.

WILLIAMS, L. G.; SMITH, C. U. Web application scalability: A model-based approach. In: *Int. CMG Conference*. [S.l.: s.n.], 2004. p. 215–226. Citado na página 33.

YU, S. Acid properties in distributed databases. *Advanced eBusiness Transactions for B2B-Collaborations*, p. 17, 2009. Citado na página 29.

Apêndices

APÊNDICE A – Primeiro Apêndice

Texto do primeiro apêndice.

APÊNDICE B – Segundo Apêndice

Texto do segundo apêndice.

Anexos

ANEXO A – Primeiro Anexo

Texto do primeiro anexo.

ANEXO B – Segundo Anexo

Texto do segundo anexo.