

UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Sharpener: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.

Pedro Morello Abbud



São Carlos – SP

Sharpener: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.

Pedro Morello Abbud

Orientador: Prof. Dr. Dilvan de Abreu Moreira

Coorientador: Prof. Dr. Orlando de A. Figueiredo

Monografia final de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.

Área de Concentração: Sistemas Computacionais

USP – São Carlos

Outubro de 2019

Abbud, Pedro Morello

Sharpen: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido. / Pedro Morello Abbud. - São Carlos - SP, 2019.

37 p.; 29,7 cm.

Orientador: Dilvan de Abreu Moreira.

Coorientador: Orlando de A. Figueiredo.

Monografia (Graduação) - Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos - SP, 2019.

1. Ensino de programação. 2. Sistemas *Web*. I. Moreira, Dilvan de Abreu. II. Instituto de Ciências Matemáticas e de Computação (ICMC/USP). III. Título.

AGRADECIMENTOS

Aqui vão meu sinceros agradecimentos ao professor Orlando de Andrade Figueiredo, que, mesmo sendo docente de outra universidade, me acolheu tão bem. Agradeço as tantas horas que investiu me orientando e aos incontáveis assuntos e livros interessantes que tem me mostrado. Guardarei estas memórias com carinho.

Também é merecedor de muito agradecimento meu grande amigo Guilherme de Freitas Perinazzo, o qual tem sido, desde meu primeiro emprego, um grande mentor. Eterna gratidão às tantas oportunidades que tive de aprender técnicas e conceitos novos com você.

RESUMO

ABBUD, P. M.. *Sharpeners: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido..* 2019. 37 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Propõe-se a construção de uma prova de conceito para uma ferramenta de ensino a programação, que seja moderna e de código aberto.

Palavras-chave: Ensino de programação, Sistemas *Web*.

ABSTRACT

ABBUD, P. M.. *Sharpeners: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido..* 2019. 37 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Proposal for the creation of a proof of concept on a programming languages teaching tool, that is both modern and open source.

Key-words: Programming language education, *Web* systems.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama que explica o funcionamento do fluxo de concessão do código de autorização, definido na seção 4.1 da norma <i>RFC 6749</i> . Fonte: <i>API Gateway OAuth 2.0 Authentication Flows</i>	21
Figura 2 – Diagrama que explica a arquitetura do sistema <i>Sharpener</i> , o qual se desenvolveu uma prova de conceito.	24
Figura 3 – Modelo entidade relacional gerado a partir da ferramenta <i>ERAlchemy</i>	26
Figura 4 – Página de <i>Login</i> da prova de conceito do sistema <i>Sharpener</i>	28
Figura 5 – Página de turmas da prova de conceito do sistema <i>Sharpener</i>	29
Figura 6 – Página de turmas da prova de conceito do sistema <i>Sharpener</i> , em que o professor inscreve suas turmas em trilhas.	29
Figura 7 – Página de trilhas da prova de conceito do sistema <i>Sharpener</i>	30
Figura 8 – Página de trilhas da prova de conceito do sistema <i>Sharpener</i> , em que uma nova trilha é criada.	30
Figura 9 – Página de trilhas da prova de conceito do sistema <i>Sharpener</i> , em que <i>clusters de exercícios</i> são associados a passos de uma trilha.	31
Figura 10 – Página de exercícios da prova de conceito do sistema <i>Sharpener</i>	31

SUMÁRIO

1	INTRODUÇÃO	13
1.1	Motivação e Contextualização	13
1.2	Objetivos	14
1.3	Organização	14
2	MÉTODOS, TÉCNICAS E TECNOLOGIAS UTILIZADAS	15
2.1	Considerações Iniciais	15
2.2	Design de <i>APIs</i>	15
2.3	<i>Python</i> e <i>Flask</i>	18
2.4	O padrão <i>OAuth</i>	18
2.5	Integração e entrega contínua	20
3	DESENVOLVIMENTO	23
3.1	Considerações Iniciais	23
3.2	Projeto	23
3.3	Atividades Realizadas	24
3.3.1	<i>Desenvolvimento da API</i>	24
3.3.2	<i>Desenvolvimento da interface</i>	25
3.3.3	<i>Desenvolvimento da CLI</i>	27
3.4	Resultados	27
3.5	Dificuldades e Limitações	27
	REFERÊNCIAS	33
ANEXO A	LINKS RELACIONADOS	35

INTRODUÇÃO

1.1 Motivação e Contextualização

Nas últimas décadas, o interesse por programação cresceu de forma extraordinária e cursos introdutórios mostram-se cada vez mais populares. No entanto, cursos de programação ainda são considerados difíceis e, com frequência, possuem taxas de desistência elevada. Iniciantes sofrem de uma vasta gama de défices e dificuldades, desde a compreensão de constructos básicos de uma linguagem de programação à elaboração de planos para resolver um dado problema (ROBINS; ROUNTREE; ROUNTREE, 2003).

Um grande número de estudos concluem, de forma semelhante, que alunos, mesmo tendo aprendido a sintaxe e semântica de uma linguagem de programação, tendem a falhar em compor os blocos fundamentais estudados em programas válidos. Parte deste problema é abordado por Davies (1993), que faz uma distinção de duas partes essenciais no aprendizado em programação: conhecimento e estratégia. Denota-se conhecimento em programação como a parte declarativa do conhecimento, *e.g.* saber que existe um constructo chamado *for* na linguagem e seu propósito. Já estratégia em programação seria empregar de forma adequada o conhecimento em face de um problema, como o uso pertinente de um *for* em um programa.

Davies (1993) defende que grande parte das abordagens em ensino tendem a investir excessivamente na parcela de conhecimento em programação, muitas vezes deixando de lado o aspecto de estratégia.

Além desta falta de ênfase no ensino de estratégia em programação, Winslow (1996) afirma que, ainda que o aluno saiba resolver manualmente o problema, existe uma grande dificuldade na adaptação desta solução para seu equivalente em um programa de computador. Este fato corrobora com a teoria de Brooks (1983) em que se afirma que programação é o processo de construir mapeamentos entre o domínio problema, possivelmente passando por muitos domínios intermediários, para chegar finalmente no domínio da programação. Por esta teoria, o entendimento de um programa nada mais é que a reconstrução de ao menos uma parte destas relações entre domínios e que este processo é orientado a criação, confirmação e refinamento de hipóteses.

A complexidade da tarefa não só se concentra no ensino de conhecimento e estratégia em programação, nem na elaboração de habilidades que relacionem o domínio do problema ao domínio da programação, mas na flexibilidade que o iniciante deve alcançar na resolução

de problemas. Desta forma, o processo de escrita de um programa deve ser oportunística e incremental, em que o plano adotado e problema sempre estão em constante reavaliação, e que isto pode resultar em desvios e reformulações do plano original (VISSER, 1990).

Para que seja possível instrumentalizar esta escrita oportunística e incremental, o aluno deve adquirir outras habilidades como *debugging* de programas e familiaridade com um ambiente de programação.

Em vista ao imenso desafio que é o ensino de programação a iniciantes, diversas ferramentas foram desenvolvidas para dar apoio pedagógico ao professor nesta tarefa. Uma razoável parcela destas ferramentas são sistemas de juízes *online*, que consistem em sistemas capazes de receber códigos desenvolvidos pelos usuários como resposta a um determinado problema, mas com a capacidade de dar um *feedback* imediato ao usuário, informando se o código está correto ou não, através de análise sintática e testes automatizados.

Já existe uma grande variedade em juízes *online* disponíveis na literatura científica, tais como: *The Huxley* [Paes *et al.* (2013)], *CodeBench* [Galvão, Fernandes e Gadelha (2016)], *UVa Online Judge* [Revilla, Manzoor e Liu (2008)], *feeper* [Alves e Jaques (2014)], *URI Online Judge* [Bez, Tonin e Rodegheri (2014)], *BOCA* [Campos e Ferreira (2004)], *RunCodes*, *Sphere Online Judge*, *HackerRank*, *CodeChef*, *InterviewBit*, *Kattis*, *LeetCode*, *CodinGame*, *CodeSignal*, *CodeWars*, *Exercism*, entre tantos outros.

Apesar da grande variedade e maturidade destas plataformas existentes, poucas têm como foco ser uma ferramenta de auxílio didático ao professor em sala de aula e praticamente nenhuma é de código aberto. Assim sendo, existe uma lacuna que pode ser preenchida por um sistema que tenha estes dois focos e dê liberdade ao professor a implementar técnicas de ensino que o convenha.

1.2 Objetivos

Pretende-se neste trabalho desenvolver uma prova de conceito de um sistema para apoio ao ensino de programação em salas de aula, que seja de código aberto e que forneça ferramentas necessárias para o professor conduzir e monitorar exercícios práticos dados em sala de aula.

1.3 Organização

No Capítulo 2 pretende-se fazer uma revisão bibliográfica das técnicas, métodos e tecnologias empregadas na implementação da prova de conceito do sistema *Sharpenner*. A seguir, no Capítulo 3, descreve-se a especificação funcional do sistema tais como seus detalhes de implementação. Finalmente no Capítulo 4 apresentam-se os resultados e possíveis trabalhos futuros.

MÉTODOS, TÉCNICAS E TECNOLOGIAS UTILIZADAS

2.1 Considerações Iniciais

Este capítulo tem como propósito fornecer o embasamento teórico necessário para o entendimento da construção do sistema proposto. Aqui é contido uma descrição detalhada das técnicas, métodos e tecnologias utilizadas, assim preparando o leitor para o conteúdo dos próximos capítulos.

2.2 Design de *APIs*

Separação de conceitos, traduzido do termo *Separation of Concerns*, é uma temática chave na arquitetura cliente-servidor da internet. Parte do porquê a internet funciona tão bem foi a preocupação, desde sua concepção, de uma interface uniforme que, desde que respeitada, daria liberdade aos desenvolvedores de implementar, em qualquer linguagem ou tecnologia, um de seus componentes.

Apesar do sucesso na separação de conceitos entre as responsabilidades que o navegador e o servidor empregam, a mesma preocupação não foi replicada entre interfaces e funcionalidades que o servidor expõe. Com frequência existe um alto acoplamento entre a interface, *Frontend*, e o servidor, *Backend*, o que resulta em uma interface não uniforme para interação dos recursos que deveriam estar expostos. Define-se aqui um recurso como qualquer conceito na internet que pode ser referenciado por um identificador único e manipulado por uma interface uniforme (MASSE, 2011).

A solução, que leva ao desacoplamento, é a interação com estas funcionalidades através de uma *API*, *Application Programming Interface*, exposta pelo *backend*, que provê formas padronizadas de acesso. Aqui não somente ganhamos portabilidade, já que possibilitamos que não só um tipo de cliente, navegadores, saibam como acessar nossos recursos, mas ganhamos espaço para criar uma camada de abstração do serviço *Web*, modelando-o em recursos. Estes recursos não serão projetados como uma cópia da organização de dados ou funcionalidades presentes no *backend* mas em representações que sejam de fácil consumo e de entendimento ubíquo pelo lado do cliente.

O desafio de criar bons serviços na internet pode ser facilitado se empregarmos estilos arquiteturais já existentes. Aqui definimos estilos arquiteturais como um conjunto coordenado de restrições arquiteturais.

Um destes estilos arquiteturas que tem ganhado cada vez mais tração se chama *REST*, *Representational State Transfer*, ou em português, *Transferência Representacional de Estado*. Cunhado por [Fielding e Taylor \(2000\)](#), o termo evoca como um sistema na *Web* deveria se comportar, uma máquina de estados virtuais em que o usuário progride através da seleção de identificadores únicos, que identificam recursos, e verbos *HTTP* que operam sobre estes recursos.

As restrições na arquitetura impostas pelo estilo *REST* são agrupadas em seis categorias:

Cliente-servidor: A separação dos papéis do cliente e servidor deve ser clara para que estes possam ser projetados e implementados de forma independente. A interação entre eles só acontece na forma de requisições, que são iniciadas pelo cliente. Servidores devem mandar respostas apenas como reações de requisições dos clientes.

Interface uniforme: interfaces *REST* possuem quatro restrições: identificação de recursos, manipulação de recursos através de representações, mensagens auto-descritivas e hipermídia como motor de estado da aplicação, *HATEOAS*.

A primeira dita que cada recurso precisa ser endereçável por um identificador único, *URI*, *Unique Resource Identifier*.

Este recurso pode ser representado de diversas formas, seja em formato *HTML*, que é mais adequado para um navegador, ou formato *JSON*, que é mais apropriado para consumo de outro programa. Percebe-se aqui que a representação é apenas uma forma de interagir com o recurso, não o próprio recurso. Isto é o que dita a segunda regra.

No consumo de uma *API*, o cliente especifica um recurso e seu estado desejado, enquanto o servidor deve responder com o recurso e seu estado real. Esta troca de mensagens deve ser feita utilizando *headers* e códigos de estado *HTTP* que descrevam o estado do recurso e metadados correspondentes, de forma que as mensagens sejam auto-descritivas.

A última restrição, *HATEOAS*, ajuda a aumentar a visibilidade e descoberta de recursos relacionados na *API*, que ajudam o cliente a navegar dinamicamente nos recursos expostos. Assim, quando referenciam-se outros recursos, também são fornecidos seus identificadores únicos e verbos aceitos para aquela rota.

Sistema em camadas: Restringe-se o comportamento dos componentes em camadas, em que cada camada só pode interagir com subjacentes. Entre as chamadas do cliente que requisita uma representação de um estado de um recurso, e o servidor que processa a requisição, pode haver vários servidores entre eles. Estes servidores podem prover um camada de segurança, *cache*, balanceamento de carga ou outras funcionalidades. Estas camadas não devem afetar a requisição ou resposta e nem o cliente nem servidor precisam estar cientes se elas existem ou não.

Protocolo sem estado: O servidor não deve lembrar absolutamente nada do usuário que utiliza sua *API*. Isto implica que toda requisição individual precisa conter toda informação necessária para que o servidor processe e retorne uma resposta. Esta restrição visa aumentar a escalabilidade, visibilidade e confiabilidade do sistema. Escalabilidade pois permite que haja *caching* de respostas e que o servidor possa desalocar recursos físicos entre requisições, visibilidade pois sistemas que monitoram requisições não precisam olhar além de uma requisição, pois elas contêm todo o dado necessário para entender a natureza desta, e a confiabilidade pois a recuperação de falhas parciais se torna mais simples (KENDALL *et al.*, 1994).

Cache: O servidor deve declarar quais dados podem ser guardados em *caches*. *Caches* ajudam a reduzir a latência percebida pelo cliente, aumentam a disponibilidade e confiabilidade de serviços e mitigam a carga de trabalho do servidor. Em resumo, *caches* reduzem todos os custos associados a um serviço na *internet*.

Código sob demanda: A internet faz bastante uso de código sob demanda, que possibilita que o servidor transfira executáveis, tais como *scripts* e *plugins*, para a execução do lado do cliente. Apesar de proposto por Fielding e Taylor (2000), esta restrição tende a estabelecer um acoplamento de tecnologias entre servidor e cliente. Por este motivo “código sob demanda” é a única restrição imposta pelo estilo arquitetural *REST* que é considerada opcional.

O maior desafio no projeto de uma *API* é abstrair componentes do sistema em recursos. O modelamento destes recursos estabelece os aspectos chave da sua *API* e é semelhante ao processo de modelar um banco de dados relacional ou mesmo o modelamento clássico de um sistema orientado a objetos.

No processo de modelagem de recursos, rotineiramente começa-se pensando em arquétipos de recursos. Estes arquétipos nos ajudam a comunicar de forma consistente as estruturas que são frequentemente encontradas em *Designs* de *REST APIs*. Idealmente, cada recurso pertence a apenas um dos seguintes arquétipos: *document*, *collection*, *controller* e *store*.

Document é o arquétipo base de todos os outros, todos os outros arquétipos são especializações deste. Sua representação tipicamente inclui campos com dados ou *hiperlinks* para outros recursos. *Collection* é um diretório de outros recursos em que clientes podem recuperar ou, possivelmente, adicionar novos itens à coleção. Já *Stores* são recursos que permitem que usuários guardem novos recursos, de forma que o próprio cliente escolha o identificador daquele recurso. Para recursos que se parecem mais como procedimentos que não se encaixam nos métodos padrões *CRUD* (*create*, *retrieve*, *update*, *delete*), o arquétipo *Controller* é adequado. Todos estes arquétipos contêm recomendações de como o identificador único deve ser escolhido, para que seja claro a quem consome a *API* de qual arquétipo se trata aquele recurso.

2.3 Python e Flask

Python é uma linguagem com sintaxe clara e concisa, favorecendo a legibilidade do código-fonte, o que a torna uma linguagem extremamente produtiva. A linguagem inclui diversas estruturas e módulos de alto nível e, por ser, segundo [a pesquisa anual de desenvolvedores do site StackOverflow](#), uma das linguagens mais populares e amadas do mundo, possui uma infinidade de excelentes *Frameworks* e bibliotecas de terceiros, que podem ser instalados de forma simples e gratuita.

A linguagem inclui vários recursos de outras linguagens modernas, tais como geradores, instropecção, persistência, metaclasses e unidades de teste. Multiparadigma, a linguagem é feita para suportar programação modular e funcional, tal como orientação a objetos. A linguagem é interpretada através de *bytecodes* pela máquina virtual do *Python*, tornando seu código excepcionalmente portátil, e, como conta com tipagem dinâmica, é muito adequada para prototipação (BORGES, 2014).

Sua sintaxe clara torna fácil seu aprendizado, mesmo por iniciantes. Além de ser largamente utilizada no desenvolvimento de sistemas, a linguagem é muito utilizada para automação de tarefas, por meio de *scripts*. Também se integra muito bem com outras linguagens, como *C* ou *Fortran*, fazendo-se uma ótima linguagem para a escrita de *wrappers*. *Python* é um *software* de código aberto e é largamente utilizado na indústria, por grandes empresas como: *Microsoft*, *Yahoo*, *Google*, *Disney*, entre tantas outras.

Python é largamente utilizado para a construção de aplicações *Web*, normalmente desenvolvidos em algum tipo de *framework*, sendo *Django* a escolha mais comum. *Django* conta com todas as ferramentas necessárias para o desenvolvimento embutidas em seu *framework*, como *ORM*, *Object Relation Mapper*, soluções para autenticação e autorização, e um sistema de administração. Ao oferecer uma solução completa, perde-se em desempenho na aplicação e inflexibiliza-se seu desenvolvimento. Para aqueles que querem construir uma aplicação enxuta e de forma menos opinada, sugere-se um *microframework Web*.

O mais popular *microframework web* escrito em *Python* se chama *Flask*. Denomina-se “*micro*” devido as poucas escolhas que foram feitas em sua implementação, deixando a cargo do desenvolvedor a implementação dos módulos que se mostrem necessários. A abordagem minimalista possibilita o surgimento de um ecossistema de pacotes diverso, em que o desenvolvedor opta por pacotes que são mais adequados a necessidade do projeto, ao invés de seguir decisões tomadas pelo *framework*.

2.4 O padrão OAuth

Apesar de vivermos em um mundo que depende, cada vez mais, de serviços digitais, a maior parcela dos usuários ainda não se educaram ou não seguem boas práticas de segurança na

internet. Não é infrequente que se usem as mesmas credenciais em múltiplos serviços, mesmo que o usuário não tenha meios de averiguar que suas informações são manuseadas e armazenadas de forma adequada.

Uma das possíveis soluções para uma melhor gestão de identidade na *internet* é o uso de identidades federadas (CRITTENDEN, 2015). Uma identidade federada é um meio de uma pessoa vincular sua identidade eletrônica e atributos pessoais a múltiplas partes distintas. Esta solução traz benefícios como mais garantias de segurança no armazenamento de credenciais, menos sobrecarga de responsabilidade para desenvolvedores de aplicações e uma melhor experiência para o usuário, já que se diminui o atrito na criação e gerenciamento de múltiplas contas.

O modelo de serviço federado pode ser dividido em duas componentes necessárias: o provedor de identidade e o provedor de serviço. Ambas as partes cumprem papéis distintos e sem qualquer uma delas não temos uma solução de identidade federada. Para que a solução funcione, deve haver um relacionamento de confiança entre o provedor de identidade e o provedor de serviço. Caso o provedor de identidade não confie no provedor de serviço, este não o confiará os dados de usuários. Caso o provedor de serviço não confie no provedor de identidade, também não confiará nas informações advindas dele.

O provedor de identidade é quem mantém as informações de identidade. Usuários se autenticam diante ao banco de credenciais do provedor, que então libera as informações relativas àquele usuário ao provedor de serviço que iniciou o processo de autenticação. O provedor de serviço é aquele que provê serviços a usuários, estes sendo aplicações, serviços de infraestrutura ou serviços de dados. Com a popularização do modelo de computação em nuvem, cada vez mais usuários entram em contato com algum dos três principais modelos de serviço em nuvem, *Infrastructure as a Service (IaaS)*, *Platform as a Service PaaS* e *SaaS, Software as a Service*, todas as quais dependem de um modelo de serviço federado (ROUNTREE, 2012).

Existem vários padrões que implementam o modelo de serviço federado, as mais famosas sendo: *SAML* (Security Assertion Markup Language), *OAuth*, *OpenID* e *Windows Identity Foundation*. O padrão que tem ganhado mais adeptos, pela sua abordagem simples porém robusta, é o padrão *OAuth*, que já conta com quatro revisões, sendo a última *OAuth 2.0*.

OAuth 2.0 é um padrão aberto criado para suportar autorização de forma segura. Existem quatro papéis definidos pelo padrão: servidor de autorização, servidor de recurso, dono do recurso, comumente chamado de usuário, e o cliente, que no contexto do *OAuth 2.0* é uma aplicação que quer ser autorizada a acessar um recurso do usuário.

O padrão suporta vários fluxos de autorização dependendo das capacidades e grau de confiança que o servidor de autorização possui em um cliente. Caso o cliente tenha a capacidade de armazenar um segredo em comum com o servidor de autorização, o fluxo de concessão do código de autorização (*Authorization Code Grant*) pode ser utilizado. Se ele não possui essa capacidade, como no caso de aplicações que são executadas no computador do usuário, o fluxo

implícito (*Implicit Grant*) pode ser utilizado. Se o cliente pode ser confiado com as credenciais do usuário, comum em aplicações da mesma empresa, ele pode utilizar o fluxo de senha do dono do recurso (*Resource Owner Password Grant*).

Alem disso, o *OAuth 2.0* permite que clientes obtenham permissão para acessar recursos que não são relacionados a nenhum usuário utilizando o fluxo de credenciais de cliente (*Client Credentials Grant*). A maioria das aplicações Web, por possuírem servidores capazes de manter um segredo em comum com o servidor de autorização, fazem o uso do fluxo de código de acesso, que é ilustrado pela Figura 1 e definido por [Hardt \(2012\)](#).

O fluxo de concessão de código de acesso começa quando um usuário, dono de um recurso, escolhe fazer *login* ou atrelar sua conta do servidor de autorização em uma aplicação. A aplicação encaminha o usuário ao servidor de autorização e o usuário preenche suas credenciais e autoriza a aplicação a acessar um determinado escopo de recursos. Em seguida, o servidor de autorização emite um código de autorização, que geralmente expira em minutos. A aplicação, em posse do código de autorização, requisita diretamente ao servidor de autorização um código de acesso. Com o código de acesso, a aplicação agora pode acessar recursos protegidos do servidor de recursos, até que este código expire.

Existem alguns detalhes que garantem a segurança do fluxo anterior. Quando a aplicação requisita o servidor de autorização por um código de autorização, ela informa um identificador de cliente, que indica sua identidade, uma *url* de retorno, que é para onde o servidor de autorização mandará o código de acesso e uma cadeia de caracteres aleatórios que será usado para validar se a requisição veio do servidor de autorização. Caso na requisição a *url* de retorno informada seja diferente da cadastrada pela aplicação no servidor de autorização previamente, a requisição é abortada. No recebimento da resposta do servidor de autenticação, caso a cadeia de caracteres aleatórios não seja a mesma da enviada, aborta-se o processo. Para obtenção do código de acesso, a aplicação também não somente deve informar ao servidor de autenticação o código de autorização, mas também um segredo, *client secret*, previamente cadastrado em conjunto com seu identificador.

2.5 Integração e entrega contínua

Um dos grandes desafios no desenvolvimento de *software* é criar um processo repetível e confiável para entregas de aplicações. Frequentemente, lançamentos de *software* são tratados por seus desenvolvedores como momentos estressantes. Associa-se este *stress* a muitos processos manuais, infrequentes e propensos a erros, conduzidos em curtos prazos de tempo.

Pode-se, no entanto, se um processo rigoroso for seguido, tornar esta tarefa fácil, tão fácil quanto o apertar de um botão. Para atingirmos este nível de maturidade no desenvolvimento de *software*, [Humble e Farley \(2010\)](#) propõem que devemos seguir à risca dois princípios: automatização de todas etapas que sejam automatizáveis e manter todos os artefatos necessários

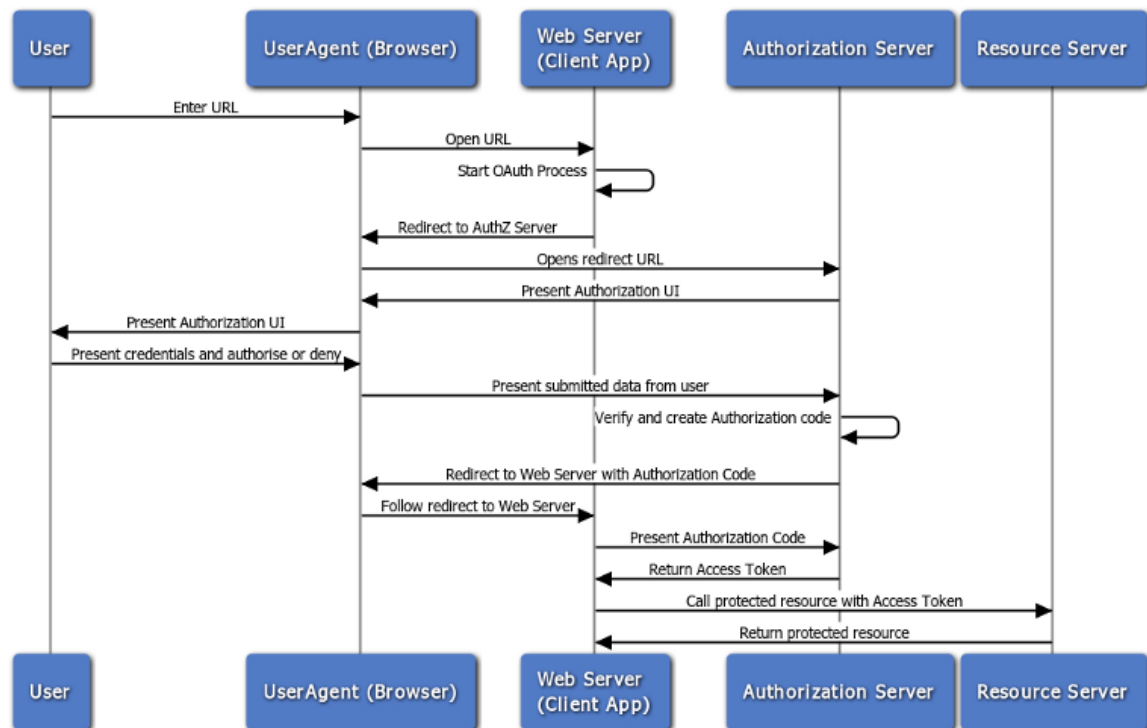


Figura 1 – Diagrama que explica o funcionamento do fluxo de concessão do código de autorização, definido na seção 4.1 da norma *RFC 6749*. Fonte: *API Gateway OAuth 2.0 Authentication Flows*.

para um lançamento em um sistema de controle de versões.

Idealmente, uma entrega de *software* é composta por três atividades: provisionar e gerenciar o ambiente em que a aplicação irá rodar (configuração de *hardware*, *software*, infraestrutura e serviços externos), instalar a versão correta da aplicação neste ambiente e configurar a aplicação, incluindo qualquer estado ou dado que ela possa requerer.

De fato, até recentemente, vários destes passos para a entrega de *software* pareciam impossíveis ou ao menos complexas de aderirem os princípios citados por [Humble e Farley \(2010\)](#). Como poderíamos, por exemplo, versionar *hardware*? Com o advento de virtualização eficiente e barata, até esta tarefa que, *a priori*, parecia impossível, virou algo corriqueiro e ubíquo. Com advento e adoção da computação em nuvem, todos os passos necessários para automatização de testes, compilação ou de artefatos e lançamento do *software* estão acessíveis a qualquer desenvolvedor.

Define-se integração contínua como o processo no desenvolvimento de *software* em que membros de um time integram seu trabalho de forma frequente. Cada integração desencadeia etapas como verificação de estilo de código, testes automatizados e construções de compilados ou outros artefatos, com o objetivo de encontrar problemas nas integrações o mais rápido possível. Já entrega contínua leva a prática de integração contínua para outro patamar e automatiza o lançamento de versões do *software* dado que as etapas anteriores foram concluídas com sucesso e o código foi revisado ([FOWLER; FOEMMEL, 2006](#)).

DESENVOLVIMENTO

3.1 Considerações Iniciais

Este capítulo apresentará o projeto como um todo, explicando escolhas que foram feitas em seu desenvolvimento, tais como ferramentas e tecnologias adotadas. Como muitas tecnologias diferentes são citadas, supõe-se aqui que os conceitos e técnicas descritas no capítulo anterior já pareçam mais familiares ao leitor.

3.2 Projeto

Com o intuito de desenvolver um ambiente de ensino que fosse igualmente conveniente para professores estruturarem suas disciplinas de programação, tanto quanto agradável para alunos participarem de atividades, desenvolveu-se nesta monografia uma prova de conceito do que se envisinou poder ser esse sistema. Espera-se que, a partir desta prova de conceito, atraía-se interessados pela construção de uma solução e que, de forma coletiva, construa-se uma plataforma de excelência, por meio de colaborações em código aberto.

O professor na plataforma será capaz de criar e gerenciar exercícios, trilhas e turmas. Alunos se inscrevem na plataforma por meio de um *login* de identidade federada, e ingressarão nas turmas por meio de um *link* específico, que será fornecido pelo professor nos primeiros dias de aula.

Trilhas de exercícios são preparadas por professores, que as associarão às suas turmas. Estas trilhas são compostas de alguns *clusters* de exercícios, que contêm exercícios de assuntos correlatos e, idealmente, de mesmo nível de dificuldade. Para cada aluno, será sorteado um exercício de cada *cluster* de forma aleatória. Caso o aluno apresente dificuldade na resolução deste exercício, o aluno pode solicitar dicas relativas ao exercício, ou, em último caso, a resolução do mesmo. Caso a solução seja requisitada, outro exercício do mesmo *cluster* é sorteado a este aluno.

Para diminuir o escopo inicial do projeto e incentivar o estudante a se familiarizar com o ferramental que envolve desenvolvimento de *software*, escolheu-se que o desenvolvimento do código acontecesse localmente no computador do aluno. Para que, ainda sim, proporcionemos ao aluno uma boa experiência, desenvolveu-se uma interface por linha de comando, que nos

referiremos a partir de agora por *CLI* (*Command Line Interface*). Por meio desta *CLI*, o aluno poderá fazer *downloads* e submissões dos exercícios propostos de maneira rápida e prática.

Dessa forma particionou-se o sistema em três grandes partes: uma interface gráfica, *Frontend*, em que professores e alunos possam interagir com exercícios, trilhas e turmas, uma *CLI* que possibilita *download* e submissão de exercícios, e um servidor com uma *API* que seja capaz de abstrair os recursos necessários por ambos os clientes. Na Figura 2 apresenta-se um diagrama da arquitetura do sistema.

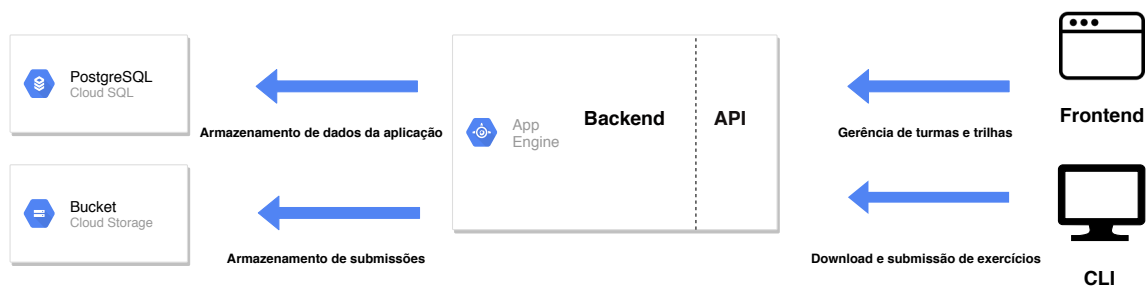


Figura 2 – Diagrama que explica a arquitetura do sistema *Sharpener*, o qual se desenvolveu uma prova de conceito.

3.3 Atividades Realizadas

3.3.1 Desenvolvimento da API

Para o desenvolvimento da *API*, escolheu-se a linguagem *Python*, por sua clareza, e sua alta produtividade, que é crucial no processo de prototipação. Em conjunto com *Python*, o *micro framework web Flask* também foi escolhido. Justifica-se essa escolha pela inerente simplicidade, clareza e produtividade do *framework*.

Antes de podermos modelar recursos para a *API*, estudou-se quais dados eram necessários para os casos de uso propostos. Este estudo gerou um *MER*, modelo entidade relacional, que relacionava todas as entidades do sistema. A partir do diagrama gerado, criou-se “*models*” que representavam estas entidades. A classe que possibilitou a criação destes *models* provêm da *ORM* que abstrai bancos relacionais chamada *SQLAlchemy*. *SQLAlchemy* é uma biblioteca estável com mais de treze anos de maturidade, mas que continua sendo a escolha padrão dos desenvolvedores.

Optou-se por utilizar *PostgreSQL*. A escolha foi feita por se tratar de um sistema gerenciador de banco de dados relacional de código aberto e por este ser referência em desempenho e funcionalidades.

No mapeamento do *MER*, previamente esboçado, em classes utilizou-se a biblioteca de visualização *ERAlchemy*. *ERAlchemy* é capaz de gerar diagramas *MER* de forma automática a partir das classes modelo ou de uma conexão com o banco de dados. Tal ferramenta se provou excepcionalmente útil pois permitiu o desenvolvimento incremental e iterativo das classes modelo

e garantiu que o modelo conceitual fosse implementado sem divergências. A Figura 3 mostra o modelo entidade relacional do sistema, gerado a partir da ferramenta.

Para que nosso sistema fosse capaz de armazenar submissões de alunos de forma confiável, um serviço de intervalos de armazenamento, traduzido do inglês *Bucket Storage*, se fez necessário. Um intervalo de armazenamento nada mais é que uma abstração de provedores de computação em nuvem para oferecer armazenamento de objetos. Uma das grandes vantagens associadas a este tipo de serviço é o baixo custo por *gigabyte*, além de sua alta escalabilidade, tanto um aumento no volume de dados armazenados, quanto na velocidade recuperação destes. O provedor de computação em nuvem escolhido foi a [Google Cloud Platform](#).

A alternativa a adotar um serviço de armazenamento por intervalos seria armazenar arquivos diretamente no banco de dados, o que traria um impacto na performance do mesmo, já que *SGBDs* não são otimizados para este tipo de dado.

Modelou-se recursos seguindo o estilo arquitetural *REST*. Coleções e documentos de exercícios, turmas, tópicos, entre outros recursos foram implementados, e um recurso do arquétipo *controller* foi necessário para prover autenticação a interface por meio do fluxo de concessão do código de autorização. Também disponibilizou-se uma rota para *health check*, que permite que uma sonda externa consulte se a aplicação continua funcionando adequadamente. Caso esta não esteja, envia-se um sinal para que um novo container da aplicação seja iniciado e o tráfego é redirecionado a ela.

Para que não seja necessário que professores criem um banco de exercícios do zero, um “conector” foi implementado para a plataforma *exercism*, em que seus exercícios são extraídos e guardados no banco de dados. A implementação aborda apenas duas linguagens *Python* e *Rust*, mas pode ser facilmente estendido para outras linguagens, dado que se informe a estrutura de arquivos que os exercícios daquela linguagem são armazenados.

Também foi configurado uma plataforma de *CI/CD* para o projeto. A toda nova versão enviada ao repositório remoto do sistema de controle de versões, uma tarefa rodava a ferramenta *autopep8* que checava se o código *commitado* era sintaticamente válido e não seguia más práticas. Caso o código fosse reprovado na tarefa anterior, este não poderia ser aprovado e mesclado na *branch* principal de desenvolvimento. Se a contribuição passar na verificação e depois de revisão for aceita, uma tarefa para lançamento da nova versão é disparada automaticamente, que coloca uma nova versão no ar. A plataforma de *CI/CD* empregada foi o [Github Actions](#) e as novas versões eram colocadas no ar através da *Platform as a Service* do Google, [AppEngine](#).

3.3.2 Desenvolvimento da interface

A interface foi desenvolvida utilizando *Javascript*, *CSS* e *HTML*. Como desejávamos uma plataforma que fosse bastante interativa, uma *single page application* foi implementada, utilizando a tecnologia [React](#). Seguiu-se o *design system* chamado *Material Design* por ser

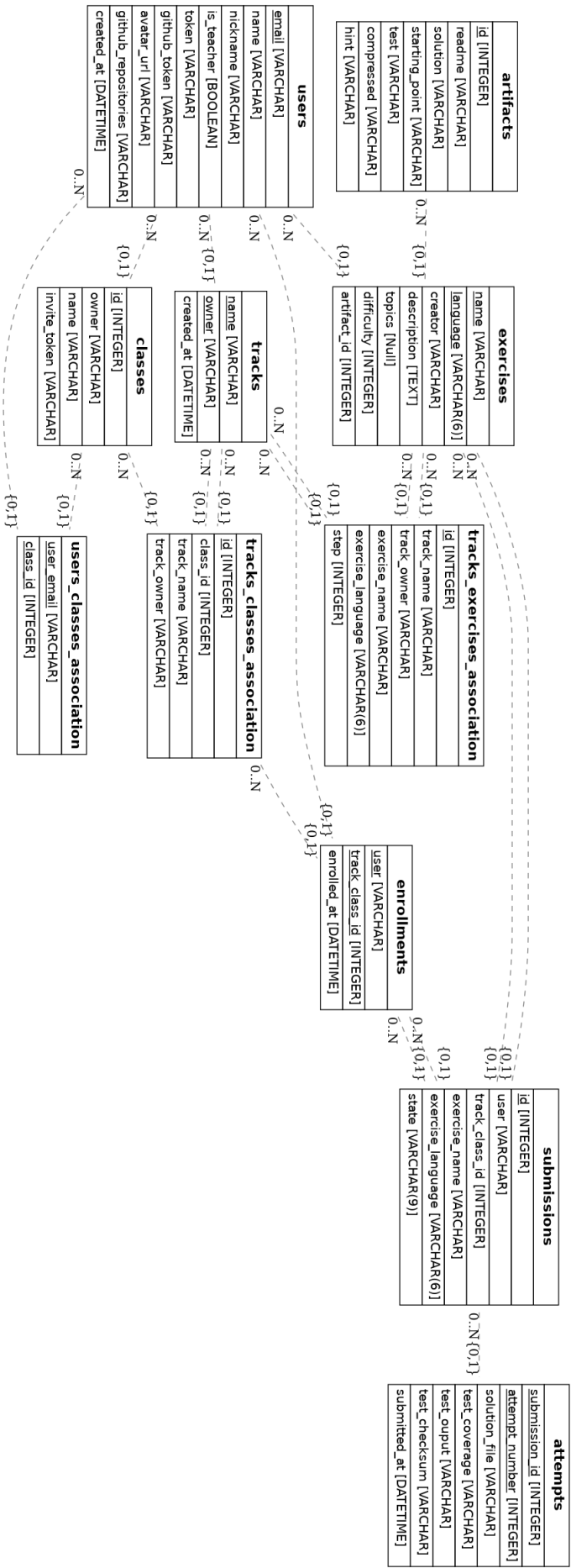


Figura 3 – Modelo entidade relacional gerado a partir da ferramenta *ERAlchemy*.

bastante intuitivo para novos usuários. A biblioteca *Material-UI* forneceu vários componentes que serviram de base para os componentes customizados. A comunicação com a *API* foi feita utilizando o cliente *HTTP axios* e os dados são persistidos numa *store* local, utilizando o gerenciador de estados *Redux*.

3.3.3 Desenvolvimento da CLI

O desenvolvimento da *CLI* foi feito na linguagem *Rust*, utilizando a biblioteca *StructOpt*. *StructOpt* é uma biblioteca que permite construção de interfaces por linhas de comando a partir da anotação de macros em *structs* ou *enums*. Com uma simples anotação ganha-se um *parser* dos comandos e mensagens que instruem o usuário como utilizar sua *CLI*.

Aceitam-se três comandos na *CLI*, “*download*” que a partir de um identificador busca o exercício proposto, *submit* que manda uma tentativa de solução do problema ao servidor e *config*, necessário que seja executado, ao menos uma vez, com uma chave que identifica qual aluno está utilizando o programa.

3.4 Resultados

A Figura 4 mostra a página inicial da interface *web* da ferramenta *Sharpen*. O *login* do usuário é feito por contas previamente cadastradas na plataforma *Github*, por meio fluxo de código de acesso. O professor quando logado, será direcionado a página de suas turmas, retratada na Figura 5, em que poderá gerenciá-las. Nesta página o professor também inscreve suas turmas em trilhas previamente criadas, como podemos observar em Figura 6.

Pode-se acessar a página que possibilita a criação de trilhas através do menu lateral, que o leva a página retratada pela Figura 7. Nesta página pode-se criar novas trilhas como mostrado na Figura 8. Uma trilha é composta por vários “passos”, que são os *clusters* de exercícios discutidos anteriormente. Ao clicar no botão de adicionar exercícios, surge um novo componente em que pode-se buscar e selecionar exercícios que farão parte daquele passo, conforme podemos observar em Figura 9. Sabemos que a elaboração de exercícios é uma tarefa que toma muito tempo, portanto na página de exercícios, retratada pela Figura 10, professores podem buscar por novos exercícios, que foram criados por outros professores ou extraídos de alguma repositório público.

3.5 Dificuldades e Limitações

Encontraram-se duas grandes dificuldade na condução deste trabalho. A primeira delas foi a orquestração de diferentes tecnologias e partes do sistema. Houve um alto custo associado a dominar, utilizar e integrar diferentes linguagens, *frameworks* e serviços. Ao longo do desen-

volvimento deste projeto, percebeu-se que o escopo do projeto abordado foi inadequado para desenvolvimento no decorrer de um semestre.

Apesar, desde a concepção do projeto, o desenvolvimento proposto era de apenas uma prova de conceito de um sistema, esperava-se que, ao final da disciplina, um protótipo já adequado para testes em salas de aula fosse alcançado. Infelizmente, tal expectativa não foi cumprida. Apesar de uma parte significativa do sistema ter sido contemplada, ainda faltam muitas funcionalidades essenciais para que um “teste de campo” seja realizado.

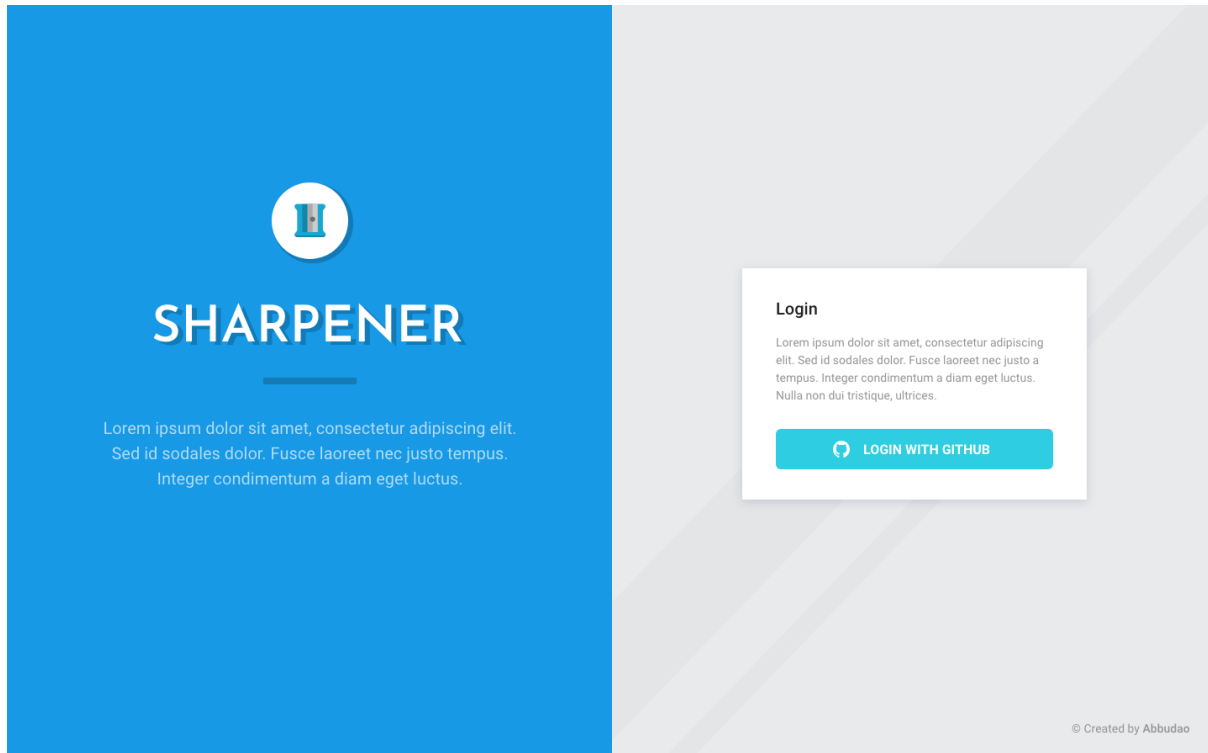
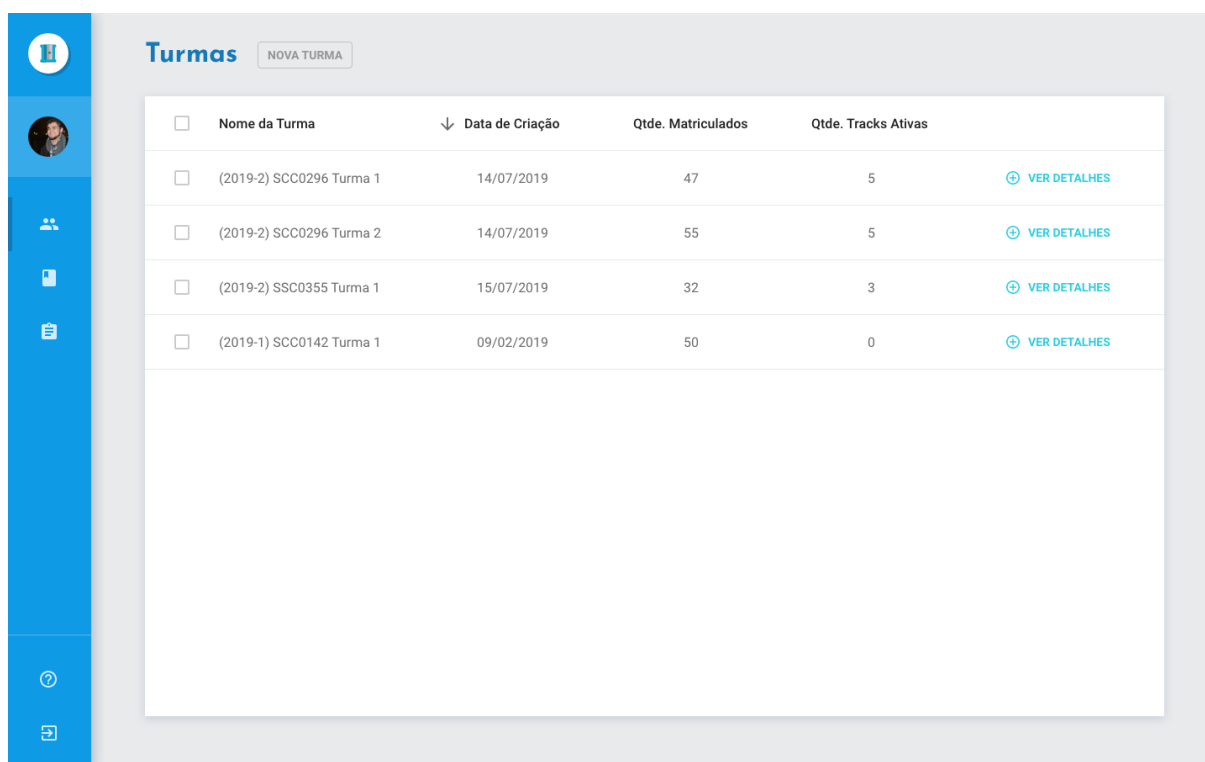
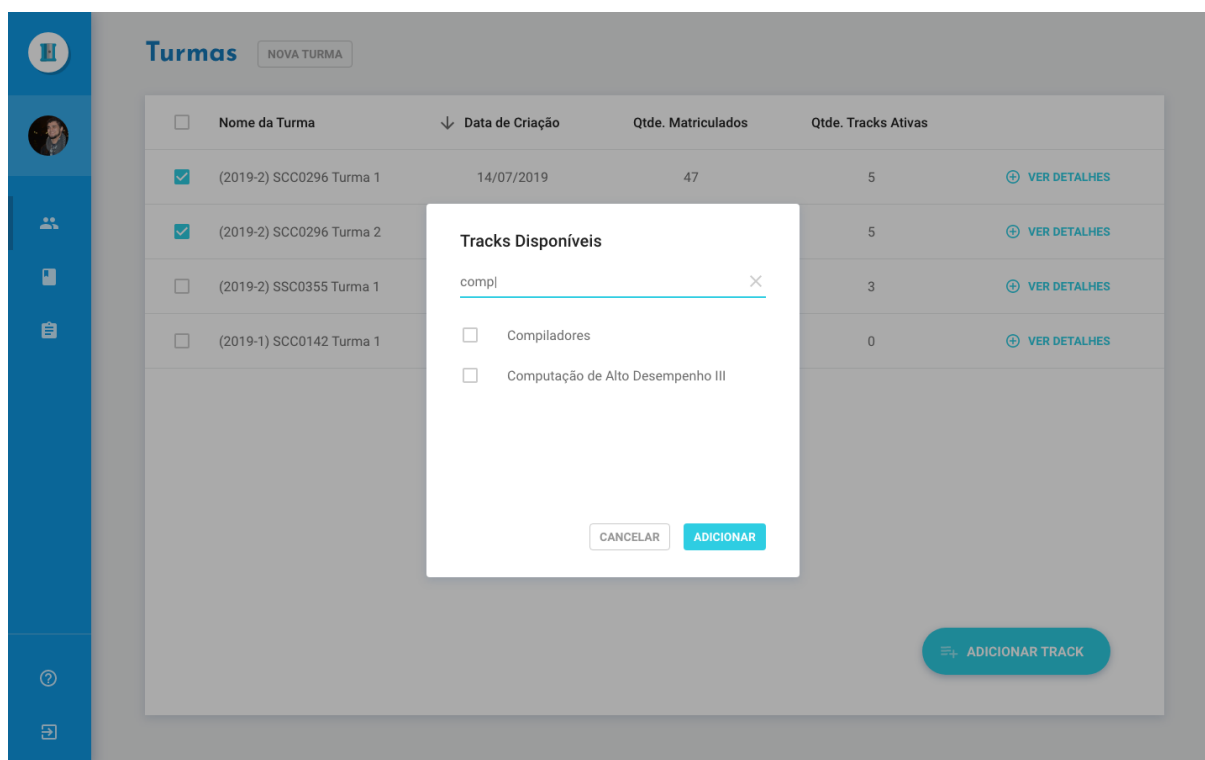


Figura 4 – Página de *Login* da prova de conceito do sistema *Sharpener*.



<input type="checkbox"/>	Nome da Turma	↓ Data de Criação	Qtde. Matriculados	Qtde. Tracks Ativas	
<input type="checkbox"/>	(2019-2) SCC0296 Turma 1	14/07/2019	47	5	+ VER DETALHES
<input type="checkbox"/>	(2019-2) SCC0296 Turma 2	14/07/2019	55	5	+ VER DETALHES
<input type="checkbox"/>	(2019-2) SCC0355 Turma 1	15/07/2019	32	3	+ VER DETALHES
<input type="checkbox"/>	(2019-1) SCC0142 Turma 1	09/02/2019	50	0	+ VER DETALHES

Figura 5 – Página de turmas da prova de conceito do sistema *Sharpener*.

<input type="checkbox"/>	Nome da Turma	↓ Data de Criação	Qtde. Matriculados	Qtde. Tracks Ativas	
<input checked="" type="checkbox"/>	(2019-2) SCC0296 Turma 1	14/07/2019	47	5	+ VER DETALHES
<input checked="" type="checkbox"/>	(2019-2) SCC0296 Turma 2			5	+ VER DETALHES
<input type="checkbox"/>	(2019-2) SCC0355 Turma 1			3	+ VER DETALHES
<input type="checkbox"/>	(2019-1) SCC0142 Turma 1			0	+ VER DETALHES

Tracks Disponíveis

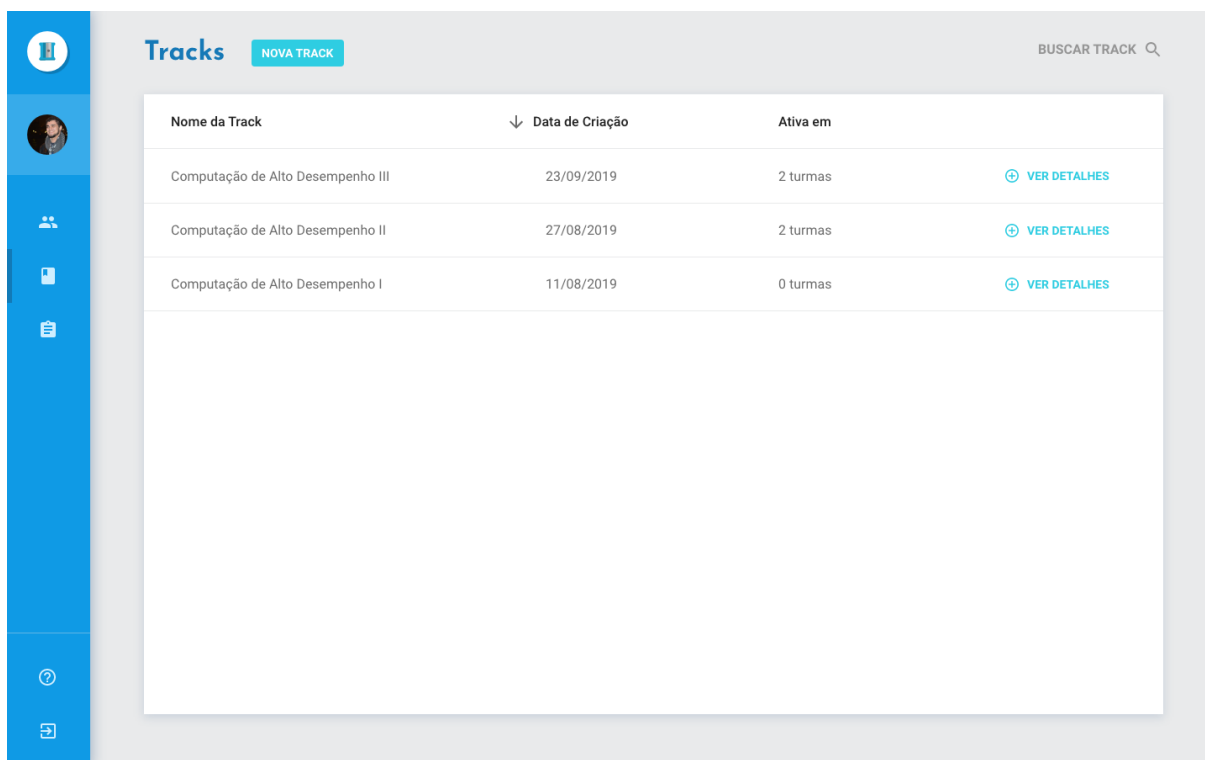
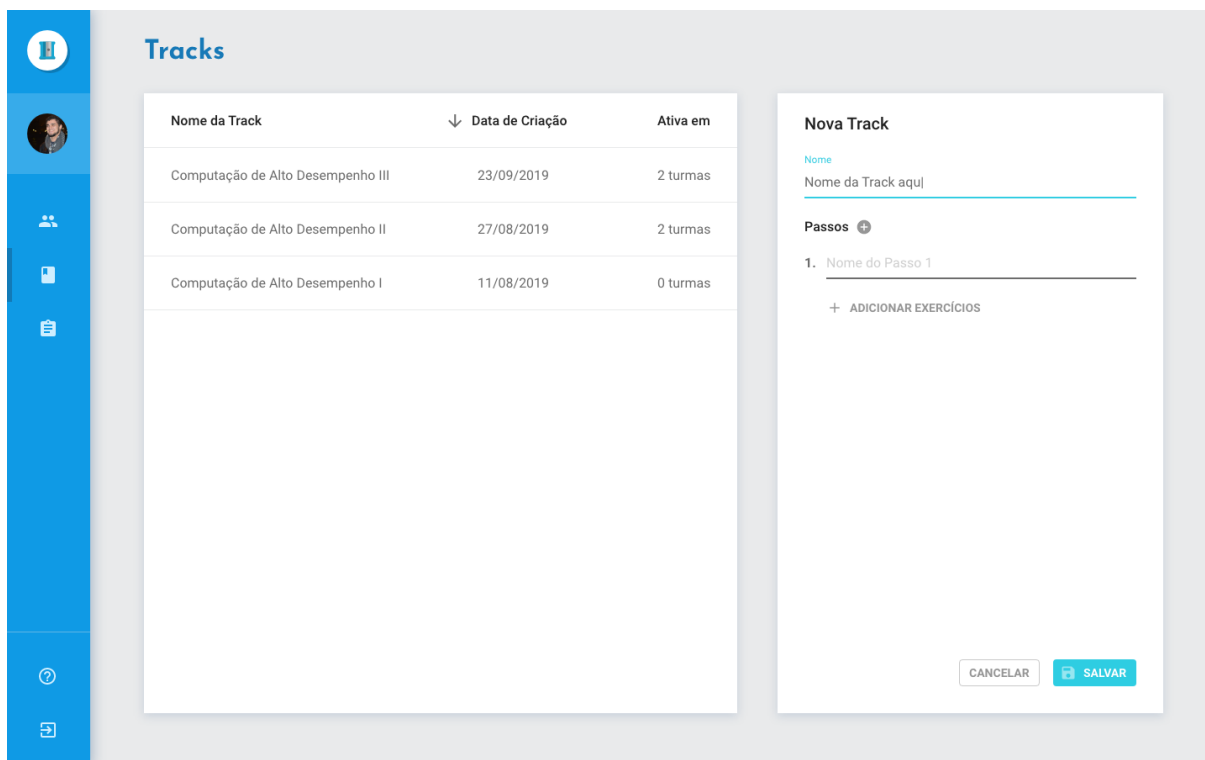
compl

- ☐ Compiladores
- ☐ Computação de Alto Desempenho III

[CANCELAR](#) [ADICIONAR](#)

[ADICIONAR TRACK](#)

Figura 6 – Página de turmas da prova de conceito do sistema *Sharpener*, em que o professor inscreve suas turmas em trilhas.

Figura 7 – Página de trilhas da prova de conceito do sistema *Sharpener*.Figura 8 – Página de trilhas da prova de conceito do sistema *Sharpener*, em que uma nova trilha é criada.

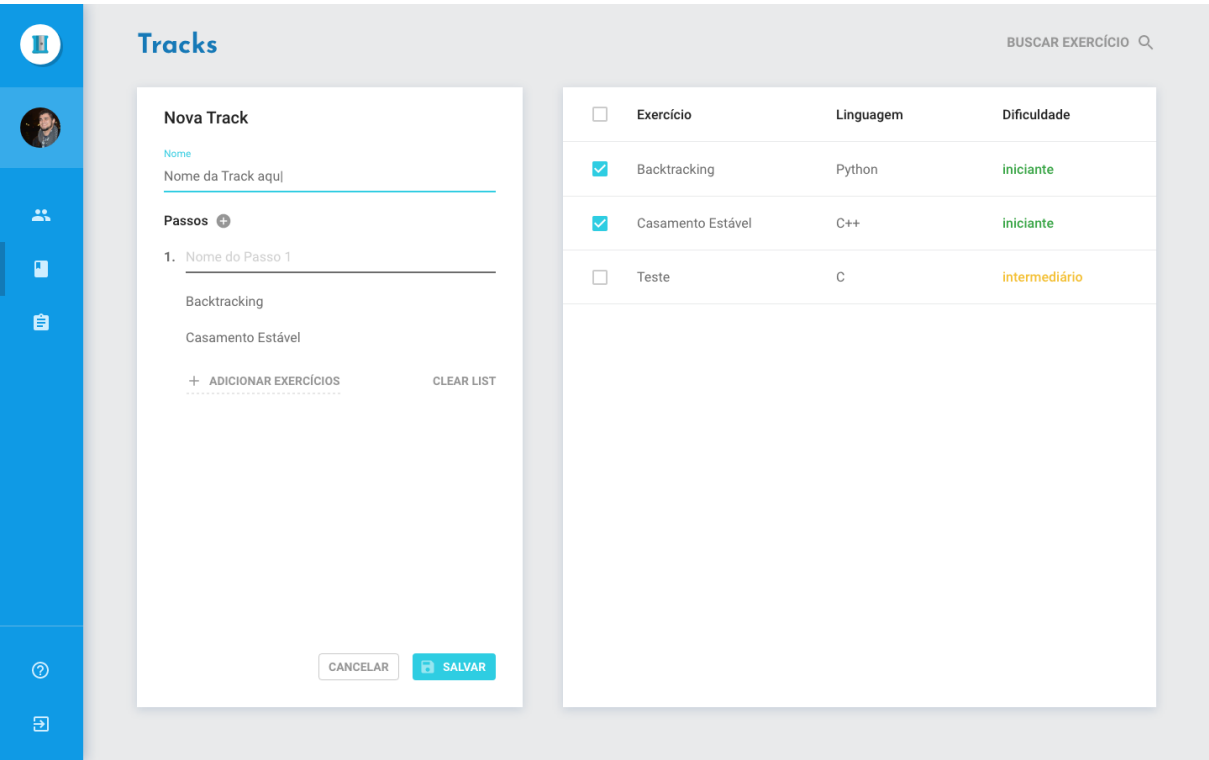


Figura 9 – Página de trilhas da prova de conceito do sistema *Sharpener*, em que *clusters de exercícios* são associados a passos de uma trilha.

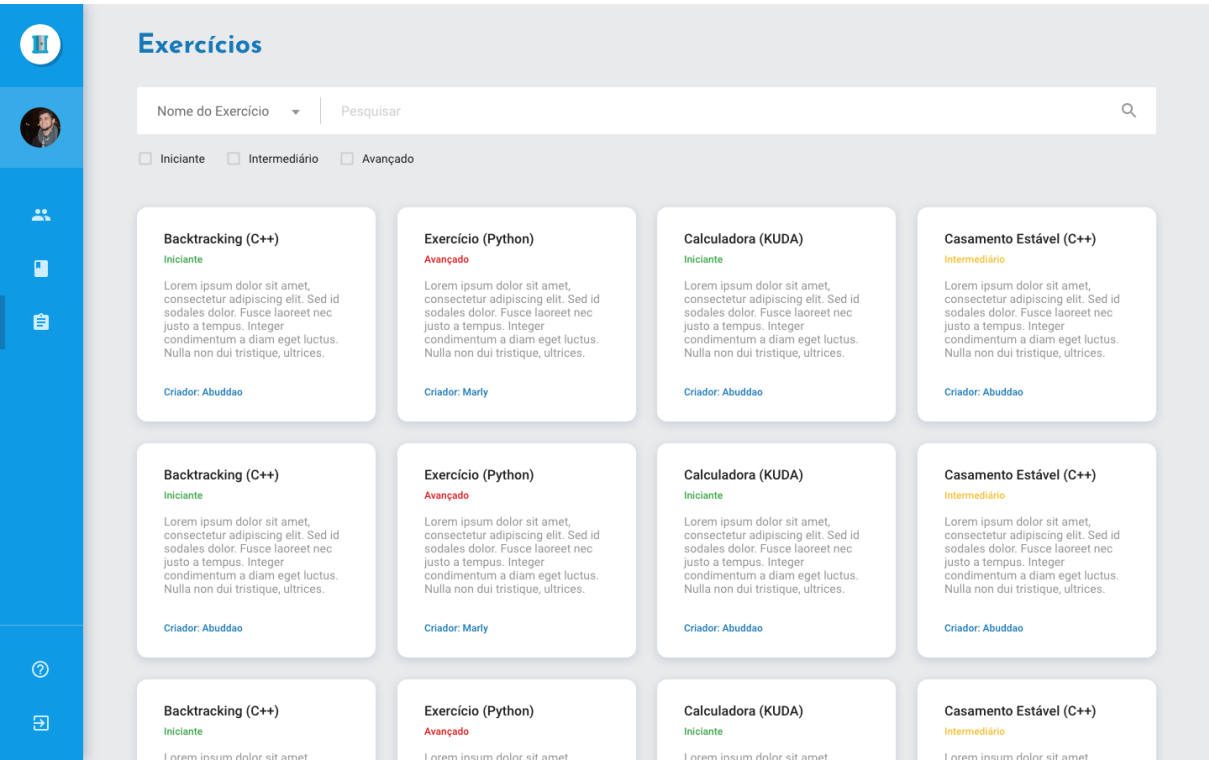


Figura 10 – Página de exercícios da prova de conceito do sistema *Sharpener*.

REFERÊNCIAS

ALVES, F. P.; JAQUES, P. Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2014. v. 3, n. 1, p. 51. Citado na página 14.

BEZ, J. L.; TONIN, N. A.; RODEGHERI, P. R. Uri online judge academic: A tool for algorithms and programming classes. In: IEEE. **2014 9th International Conference on Computer Science & Education**. [S.l.], 2014. p. 149–152. Citado na página 14.

BORGES, L. E. **Python para desenvolvedores: aborda Python 3.3**. [S.l.]: Novatec Editora, 2014. Citado na página 18.

BROOKS, R. Towards a theory of the comprehension of computer programs. **International journal of man-machine studies**, Elsevier, v. 18, n. 6, p. 543–554, 1983. Citado na página 13.

CAMPOS, C. P. de; FERREIRA, C. E. Boca: um sistema de apoio a competições de programação. 2004. Citado na página 14.

CRITTENDEN, R. What is federated identity and why should i care? In: . Washington, D.C.: USENIX Association, 2015. Citado na página 19.

DAVIES, S. P. Models and theories of programming strategy. **International Journal of Man-Machine Studies**, Elsevier, v. 39, n. 2, p. 237–267, 1993. Citado na página 13.

FIELDING, R. T.; TAYLOR, R. N. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine Doctoral dissertation, 2000. v. 7. Citado 2 vezes nas páginas 16 e 17.

FOWLER, M.; FOEMMEL, M. Continuous integration. **Thought-Works)** [http://www.thoughtworks.com/Continuous Integration. pdf](http://www.thoughtworks.com/Continuous%20Integration.pdf), v. 122, p. 14, 2006. Citado na página 21.

GALVÃO, L.; FERNANDES, D.; GADELHA, B. Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In: **Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)**. [S.l.: s.n.], 2016. v. 27, n. 1, p. 140. Citado na página 14.

HARDT, D. Rfc 6749: The oauth 2.0 authorization framework. **Internet Engineering Task Force (IETF)**, v. 10, 2012. Citado na página 20.

HUMBLE, J.; FARLEY, D. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)**. [S.l.]: Pearson Education, 2010. Citado 2 vezes nas páginas 20 e 21.

KENDALL, S. C.; WALDO, J.; WOLLRATH, A.; WYANT, G. A note on distributed computing. Sun Microsystems, Inc., 1994. Citado na página 17.

MASSE, M. **REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces**. [S.l.]: "O'Reilly Media, Inc.", 2011. Citado na página 15.

PAES, R. de B.; MALAQUIAS, R.; GUIMARÃES, M.; ALMEIDA, H. Ferramenta para a avaliação de aprendizado de alunos em programação de computadores. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2013. v. 2, n. 1. Citado na página 14.

REVILLA, M. A.; MANZOOR, S.; LIU, R. Competitive learning in informatics: The uva online judge experience. **Olympiads in Informatics**, Institute of Mathematics and Informatics, v. 2, n. 10, p. 131–148, 2008. Citado na página 14.

ROBINS, A.; ROUNTREE, J.; ROUNTREE, N. Learning and teaching programming: A review and discussion. **Computer science education**, Taylor & Francis, v. 13, n. 2, p. 137–172, 2003. Citado na página 13.

ROUNTREE, D. **Federated identity primer**. [S.l.]: Newnes, 2012. Citado na página 19.

VISSER, W. More or less following a plan during design: opportunistic deviations in specification. **International journal of man-machine studies**, Elsevier, v. 33, n. 3, p. 247–278, 1990. Citado na página 14.

WINSLOW, L. E. Programming pedagogy—a psychological overview. **ACM Sigcse Bulletin**, ACM, v. 28, n. 3, p. 17–22, 1996. Citado na página 13.

LINKS RELACIONADOS

<<https://www.thehuxley.com>>: Página da plataforma *The Huxley*, que utiliza um sistema de juiz *online* para auxiliar professores em aulas de programação.

<<http://codebench.icomp.ufam.edu.br>>: Página da plataforma *CodeBench*, que utiliza um sistema de juiz *online*, de forma gamificada, para auxiliar professores em aulas de programação.

<<https://uva.onlinejudge.org>>: Página da plataforma *UVa Online Judge*, um sistema de juiz *online* desenvolvido pela *University of Valladolid*.

<<http://feeper.unisinos.br>>: Página da plataforma *UVa Online Judge*, um sistema de juiz *online* desenvolvido pela *Universidade do Vale do Rio dos Sinos*.

<<https://www.urionlinejudge.com.br>>: Página da plataforma *URI Online Judge*, um sistema de juiz *online* desenvolvido pela *Universidade Regional Integrada*.

<<https://www.ime.usp.br/~cassio/boca>>: Página da plataforma *BOCA*, um sistema de juiz *online* desenvolvido pelo *Instituto de Matemática e Estatística* para apoio em competições.

<<https://we.run.codes>>: Página da plataforma *RunCodes*, um sistema de juiz *online* desenvolvido pelo *Instituto de Matemática e Estatística*, para auxiliar professores em aulas de programação.

<<https://www.spoj.com>>: Página da plataforma *Sphere Online Judge*, um sistema de juiz *online* desenvolvido pela *Sphere Research Labs*.

<<https://www.hackerrank.com>>: Página da plataforma *HackerRank*, que utiliza um sistema de juiz *online* para auxiliar na contratação de funcionários para empresas.

<<https://www.codechef.com>>: Página da plataforma *CodeChef*, que utiliza um sistema de juiz *online*, desenvolvida pela empresa *Directi*, para competições em programação.

<<https://www.interviewbit.com>>: Página da plataforma *HackerRank*, que utiliza um sistema de juiz *online* para auxílio na preparação de entrevistas para empresas.

<<https://open.kattis.com>>: Página da plataforma *Kattis*, um sistema de juiz *online* desenvolvido para promover competições.

<<https://leetcode.com>>: Página da plataforma *LeetCode*, que utiliza um sistema de juiz *online* para auxílio na preparação de entrevistas para empresas.

<<https://www.codingame.com>>: Página da plataforma *CodinGame*, que utiliza um sistema de juiz *online*, de forma completamente gamificada, para ensino e auxiliar empresas na contratação de funcionários.

<<https://codesignal.com>>: Página da plataforma *CodeSignal*, que utiliza um sistema de juiz *online* para auxiliar empresas na contratação de funcionários.

<<https://www.codewars.com>>: Página da plataforma *CodeWars*, que utiliza um sistema de juiz *online* para auxiliar professores em aulas de programação e empresas na contratação de funcionários.

<<https://exercism.io>>: Página da plataforma *Exercism*, que utiliza um sistema de juiz *online*, para ensino. As submissões contam com feedback de instrutores voluntários.

<<https://insights.stackoverflow.com/survey/2019>>: Resultado de 2019 da pesquisa anual de desenvolvedores do site *StackOverflow*. É a maior e mais abrangente pesquisa que envolve desenvolvedores.

<https://docs.oracle.com/cd/E39820_01/doc.11121/gateway_docs/content/oauth_flows.html> Documentação do *software API Gateway*, desenvolvido pela *Oracle Corporation*.

<<https://www.sqlalchemy.org>> : *SQLAlchemy* trás um conjunto de ferramentas e uma *ORM* para utilização de bancos de dados relacionais na linguagem *Python*.

<<https://www.postgresql.org>> : Página do site do banco relacional e de código aberto *PostgreSQL*.

<<https://pypi.org/project/ERAlchemy>> : Biblioteca para *Python*, chamada *ERAlchemy*, capaz de gerar diagramas entidade relacional através de *models* do *SQLAlchemy* ou de uma conexão com banco de dados.

<<https://cloud.google.com>> : Página do fornecedor de computação em nuvem, *Google Cloud Platform*.

<<https://github.com/features/actions>> : Página para a plataforma de *Continuous Integration* e *Continuous Delivery* do site *Github*.

<<https://cloud.google.com/appengine>> : Página para o *Platform as a Service* da *Google Cloud Platform*, chamado *AppEngine*.

<<https://pt-br.reactjs.org>> : *Framework javascript* para a construção de interfaces em *Single Page Applications*.

<<https://material-ui.com>> : Biblioteca que implementa o *design system* da Google, *Material Design*, em *React*.

<<https://github.com/axios/axios>> : Página do mais popular cliente *HTTP javascript* da atualidade.

<<https://redux.js.org>> : Página para a biblioteca *Redux*, que implementa parte da arquitetura *FLUX*. *Redux* é um gerenciador de estados previsível e centralizado para aplicações *Web*.

<<https://github.com/TeXitoi/structopt>> : Biblioteca em *Rust* que gera interfaces por linha de comando a partir de *structs* anotadas com macros.