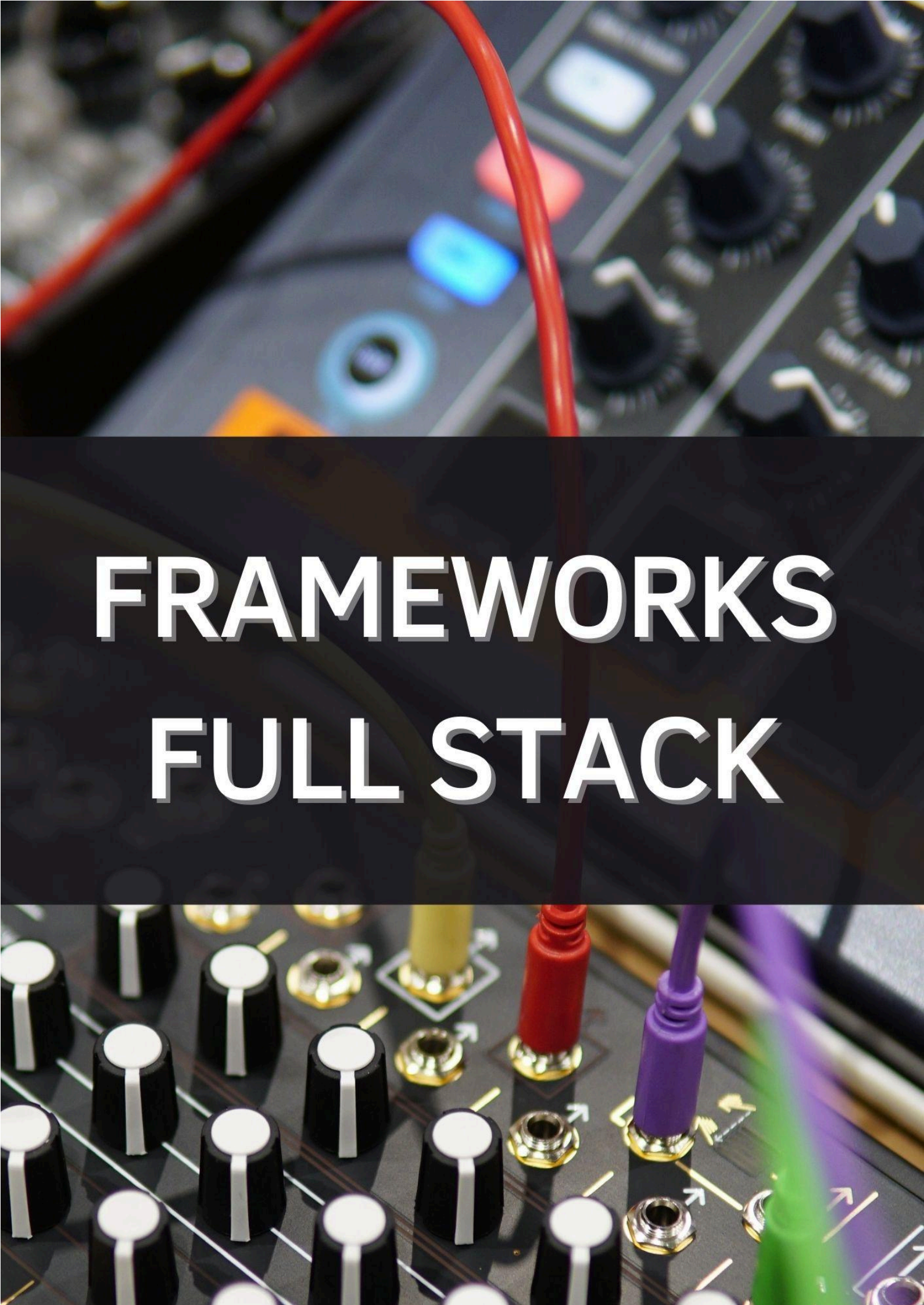


FRAMEWORKS FULL STACK

Texto base

2

Levantamento de requisitos e System Design

Prof. André Nascimento Maia
Prof. Caio Nascimento Maia

Resumo

Nesta aula, entenderemos os conceitos sobre levantamentos de requisitos e System Design, pontos cruciais para o bom andamento de projetos de software modernos. Além disso, serão apresentadas algumas técnicas e perguntas-chave para se fazer em períodos iniciais dos projetos.

2.1. Levantamento de requisitos

Antes de começarmos a desenvolver um projeto de software, é necessário garantir que entendemos o que será feito ou quais os requisitos obrigatórios para que o projeto seja considerado terminado.

Para garantir um bom andamento de projeto, um planejamento adequado deve ser feito antes mesmo de começarmos a desenvolver códigos, criar a infraestrutura necessária ou decidir entre quais tecnologias devemos utilizar. Primeiro precisamos entender coisas como qual o problema que está sendo resolvido, seus desafios, os fluxos de sucesso e os de exceção, como deveria ser a jornada de usuário para as funcionalidades determinadas, qual a quantidade de usuários acessarão o sistema e até mesmo, requisitos mais técnicos como: quantas requisições devemos processar por segundo, por quanto tempo os dados devem ser mantidos, se existe alguma restrição de tecnologia, entre outros. Garantir que pensamos nesse assunto antes de iniciarmos a produção vai facilitar a jornada de desenvolvimento do projeto, culminando em um desenvolvimento previsível, facilitando na escolha das tecnologias e maior assertividade ao determinar prazos de entrega.

O tema Levantamento de Requisitos não é recente e existem diversas técnicas que podem ser empregadas para a resolução desse problema, porém, nessa aula iremos focar em um levantamento de requisitos de alto nível, no qual será possível focar nos principais problemas do projeto e de forma ágil prosseguir com as próximas etapas do planejamento para enfim, dar início ao desenvolvimento do software.

Fazer um levantamento de requisitos mínimo é especialmente importante para projetos em que o desenvolvedor atua de ponta-a-ponta, ou de forma *full-stack*, pois o mesmo precisa de uma visão do todo para garantir que os serviços que serão criados e as tecnologias que serão utilizadas são as melhores para a resolução do problema em questão, e que também vai garantir que o mesmo tenha um ciclo de desenvolvimento de software ágil e facilitado com ferramentas que se integram entre si.

Para fazer um levantamento de requisitos mínimo, onde possamos ter informação suficiente para o início do desenvolvimento, podemos seguir 4 passos:

1. Entender a necessidade do usuário.
2. Definir o objetivo e o escopo do problema.
3. Levantar os requisitos funcionais e não funcionais.
4. Entender as premissas do projeto.

2.1.1. Entender a necessidade do usuário

O primeiro passo para qualquer demanda de software é entender a necessidade da mudança para o usuário. Nesse momento, procuramos responder perguntas que foquem em como aquela mudança ou criação será benéfica para a pessoa que está utilizando o sistema, como aquilo pode ter um impacto positivo no fluxo de serviços que o usuário acessa ou como pode facilitar a utilização do seu sistema. Focando primeiramente nesse tópico, o desenvolvedor *full-stack* é capaz de entender o que realmente importa para o usuário em meio a diversos requisitos diferentes para focar os seus esforços em pontos que são prioritários, com o intuito de escolher as melhores tecnologias para a resolução do problema.

Algumas perguntas que podem surgir nesse momento são:

1. Como esse fluxo impacta o usuário final na utilização do sistema?
2. Por que o usuário precisa dessa funcionalidade?
3. Como será a jornada do usuário para acessar essa funcionalidade ou sistema?
4. O quão benéfico será a implementação disso para o usuário?

2.1.2. Definir o objetivo e o escopo do problema

Dado que entendemos a motivação da construção do projeto ou funcionalidade, como próximo passo definiremos os objetivos e o escopo do problema, ou seja, qual é o ponto que queremos alcançar para ter uma definição de projeto pronto, ou o que está dentro do escopo de atuação desse projeto ou parte dele.

Em geral, projetos de software podem não ter fim, existindo enquanto os mesmos forem mantidos e atualizados. Para garantir que projetos em que atuamos tenham um fim, ou que o mesmo possa ser dito como entregue em algum momento, é de extrema importância que o objetivo e escopo de atuação do problema seja bem definido desde o início. Dessa forma, será garantido que o time ou desenvolvedor terá um objetivo central de atuação no projeto e poderá perseguir isso para então começar um novo ciclo de desenvolvimento em novos projetos.

No momento da definição desses pontos, precisamos responder questões similares à essas:

1. Qual o objetivo central do projeto?
2. Qual problema está sendo resolvido?
3. O que faremos e o que não faremos para resolver esse problema?
4. O que está fora do escopo de atuação desse projeto?

2.1.3. Levantar os requisitos funcionais e não funcionais

Os requisitos de desenvolvimento de software podem ser classificados de duas formas diferentes: requisitos funcionais e requisitos não funcionais. Ambos atuam em conjunto para que sejam definidos requisitos como um todo para o projeto.

Requisitos funcionais são definidos como funções de sistemas ou componentes, enquanto que uma função é descrita como a especificação de um comportamento entre entradas e saídas (Functional Requirement, 2021). Em outras palavras, os requisitos funcionais são aqueles que especificam as funcionalidades do sistemas e que especificam como um produto ou projeto deve funcionar. Alguns exemplos de requisitos funcionais são: regras de negócio, funcionalidades administrativas, relatórios a serem gerados, cancelar uma transação financeira.

Requisitos não funcionais são definidos como critérios que podem ser usados para julgar a operação do sistema (Non-functional Requirement, 2021). Em geral, não existe consenso definitivo sobre requisitos não funcionais e entendimento concreto aceito na comunidade de computação como existe para requisitos funcionais, entretanto, existe um consenso unânime em que requisitos não funcionais são importantes e críticos para o sucesso de um projeto (Glinz, Martin. 2010). Para esse tipo de requisito, ao invés de descrevermos uma funcionalidade do sistema, devemos nos preocupar em como ele irá lidar com tópicos mais gerais, como:

1. Qual a latência máxima para as operações disponíveis no sistema?
2. Como será a usabilidade do sistema?
3. O quanto o sistema deve ser confiável?
4. Como serão tratados os pontos de segurança?
5. Qual o nível de disponibilidade do sistema?

2.1.4. Entender as premissas do projeto

As premissas são necessárias para nos guiar em relação ao escopo do que será desenvolvido. Elas determinam pontos de partida e regras básicas e fundamentais sobre os requisitos funcionais e não funcionais. Muitas vezes, as premissas nos indicam qual tipo de linguagem de programação devemos utilizar, tecnologias específicas, regras importantes para o negócio que devemos assumir durante todo o projeto e como resultado da solução. Alguns exemplos de premissa para um projeto de e-commerce seriam:

Premissas

1. Devemos usar o provedor de pagamento on-line MeuProvedor.com para todos os tipos de pagamentos no e-commerce. Este provedor é nosso parceiro e nos oferece vantagens competitivas de mercado.

2. A linguagem de programação para os serviços de *back-end* deve ser Python. Isso nos garantirá grande produtividade e aproveitamos o *know-how* dos nossos times de engenharia de software.
3. Nossa solução de e-commerce não criará um serviço de autenticação. Ele deve ser provido como serviço da empresa MinhaEmpresa.

Nos exemplos de premissas acima, fica claro o que será construído e o que não será construído na solução proposta, além de também deixar claras todas as regras importantes que assumimos para o desenvolvimento do software.

2.1.5. Prototipação

Após levantar todos os requisitos do sistema, um passo opcional mas altamente indicado é o de prototipação de sistemas.

Uma prática muito comum quando utilizamos padrões de projetos ágeis é o de prototipar e validar uma ideia antes de concluir o desenvolvimento completo de um projeto que eventualmente pode ser mais longo. Prototipar, nos permite visualizar o sistema como um todo, de ponta a ponta, facilitando entendermos como ele deve ser construído e como as diversas partes de software se conectam e se comunicam.

Na etapa de prototipação, focamos em elencar pequenas funcionalidades ou as funcionalidades principais, porém reduzidas de alguma forma para criar um protótipo e validar a ideia de atuação, seja essa validação uma decisão de utilização de tecnologia ou até mesmo para a validação da funcionalidade ou projeto em si.

Quando implementado um protótipo ou forma reduzida do projeto completo, diversos problemas podem ser encontrados e resolvidos ainda nesse primeiro momento. Em alguns casos mais complexos, pode até mesmo acontecer do projeto não ter continuidade pela validação não ter tido resultados satisfatórios.

Fazendo prototipação de projetos garantimos que estamos seguindo um conceito muito utilizado nas empresas modernas, o conceito *Fail-Fast*. O conceito, em tradução livre, significa "falhar rápido". Esse conceito é especialmente importante para garantir que os problemas são encontrados rapidamente, ao invés de serem descobertos ao final de longos projetos. Garantindo que estamos falhando rápido, também garantimos que resolveremos os problemas rapidamente, antecipamos situações inesperadas, aprendendo mais sobre o que estamos construindo e asseguramos maior confiabilidade no processo de desenvolvimento de software, entregando mais valor em menos tempo.

Ideias similares à prototipação também podem ser encontradas nos conceitos de *mininum viable product* (*MVP)* ou *proof of concept* (*POC)*.

2.1.6. Aplicando o levantamento de requisitos ao projeto da disciplina

Ao longo da disciplina iremos acompanhar o projeto de um *e-commerce* de canecas personalizadas. Podemos aplicar o levantamento de requisitos à esse projeto da seguinte forma:

1. **Entender a necessidade do usuário**: clientes que gostam de canecas personalizadas têm a necessidade de comprar suas canecas on-line através de um

e-commerce. Esses clientes acessarão um website que pode ser visualizado de forma agradável quando acessado pelo seu *smartphone*, *tablet* ou pelo seu computador pessoal por um *browser*.

2. **Definir o objetivo e escopo do problema**: o objetivo será o de construir um site que venda canecas personalizadas como seu único produto, podendo pagar através de serviços de pagamento on-line, como por exemplo, o PagSeguro.
3. **Levantar os requisitos funcionais e não funcionais**:
    a. Requisitos Funcionais:
        i. Listar as canecas disponíveis;
        ii. Permitir a adição de canecas em um carrinho de compras;
        iii. Pagar um pedido de canecas;
        iv. Notificar o usuário sobre as etapas do seu pedido.
    b. Requisitos não funcionais:
        i. Latência: a loja deve exibir os produtos com baixa latência nas requisições de leitura no servidor para evitar que usuários desistam das suas compras, enquanto que para pagamentos, o usuário pode aguardar o processamento da sua requisição por no máximo uma hora;
        ii. Usabilidade: a loja deve ter fluxos simples para adição dos produtos no carrinho, bem como fluxo de pagamento facilitado e com o mínimo de passos e telas possíveis;
        iii. Confiabilidade: as compras dos clientes devem ter forte consistência de armazenamento de dados, garantindo que os clientes consigam ver seus pedidos e o status de processamento dos mesmo sempre que acessarem a página, além disso, é necessário que a loja seja resiliente e se recupere de possíveis falhas de infraestrutura de forma automática e também que se auto ajuste aos picos de tráfego de usuários, garantindo que os desenvolvedores não precisarão criar novas instâncias da aplicação de forma manual quando a quantidade de cliente comprando aumentar ou quando ocorreram falhas sistêmicas;
        iv. Segurança: os dados do cliente nunca devem ser expostos, além de serem armazenados em ambiente seguro com criptografia adequada. O sistema também deverá ter camada anti-DDoS para evitar possíveis ataques à sua integridade;
        v. Disponibilidade: a loja deve estar disponível 24 horas por dia, durante 7 dias por semana.

Diversos pontos foram levantados (e muitos ficaram de fora) para que possamos planejar de forma adequada e termos informações disponíveis para a tomada de decisões técnicas para garantir o atingimento do objetivo.

Apesar de diversos requisitos terem sido levantados, o projeto de *e-commerce* de canecas é um projeto acadêmico que servirá para ilustrar a utilização de ferramentas comuns ao *framework full-stack*, e portanto, não iremos implementar todos os requisitos levantados acima.

2.2. System Design

Como um desenvolvedor *full-stack*, após levantar os requisitos do sistema, entramos em uma etapa técnica mais complexa, onde os componentes e suas interações devem ser descritos, geralmente em forma de diagrama ou documentação escrita. Essa descrição é comumente chamada de *System Design*, ou projeto de sistema (em tradução livre).

Diversas empresas modernas que detém arquitetura complexas de sistemas, geralmente com diversas interações entre diferentes serviços e sistemas, praticam o *System Design* como passo antes do desenvolvimento de projetos. A arquitetura de sistemas deixa claro para os desenvolvedores onde eles devem atuar, como eles devem se comunicar com os sistemas vizinhos e como os fluxos deverão percorrer e sensibilizar os sistemas desenvolvidos. Em muitos casos, e principalmente em empresas consideradas *Big Techs* (Facebook, Apple, Amazon, Netflix, Google, entre outros), o *System Design* é uma das etapas mais importantes do processo de entrevistas para cargos em Engenharia de Software, para qualquer nível de senioridade.

Fazer um bom *System Design* garante que o projeto seguirá de forma previsível, tecnicamente falando, deixando claro as responsabilidades dos sistemas, arquitetura utilizada, domínio do problema e interação entre componentes.

Existem diversas formas de fazer o *System Design*, como a utilização de notação UML (*Unified modeling language*) para prototipação dos componentes e comunicação de objetos da arquitetura. Também é muito comum nos deparamos com notação livre para demonstrar a arquitetura, onde, sem padronização alguma, os desenvolvedores desenham caixas e setas para indicar o fluxo de requisições ou informações em um sistema. Para a demonstração do *System Design* do projeto da disciplina, iremos utilizar um padrão bem difundido entre as maiores empresas de tecnologia, o *C4 Model* (BROWN, 2021).

2.2.1 C4 Model

O *C4 Model* é dividido em quatro camadas que seguem uma ordem de importância, onde começamos de forma mais abstrata e conforme são descritas as próximas camadas, é especificado cada vez mais a fundo o sistema em questão. Algumas de suas camadas são opcionais, dependendo da complexidade do projeto. De forma geral, quase todas as arquiteturas de sistemas construídas precisarão ao menos das camadas 1 e 2. O nome é acrônimo para *context, containers, components and code* que é a descrição das camadas especificadas no padrão, sendo respectivamente:

1. **Contexto:** o diagrama de contexto de uma aplicação específica, onde o escopo é apenas aquele software ou aplicação em questão. É um bom ponto de partida para a diagramação e documentação de software, onde o mesmo dá a oportunidade de dar um passo para trás e visualizar o panorama geral do sistema. A indicação geral é desenhar um diagrama onde o sistema em questão é uma caixa no centro, cercado por seus usuários ou outros sistemas que têm integração. Nesse momento, detalhes não são tão importantes, já que o objetivo central é ver um diagrama do sistema de forma geral, o foco deve ser nos atores,

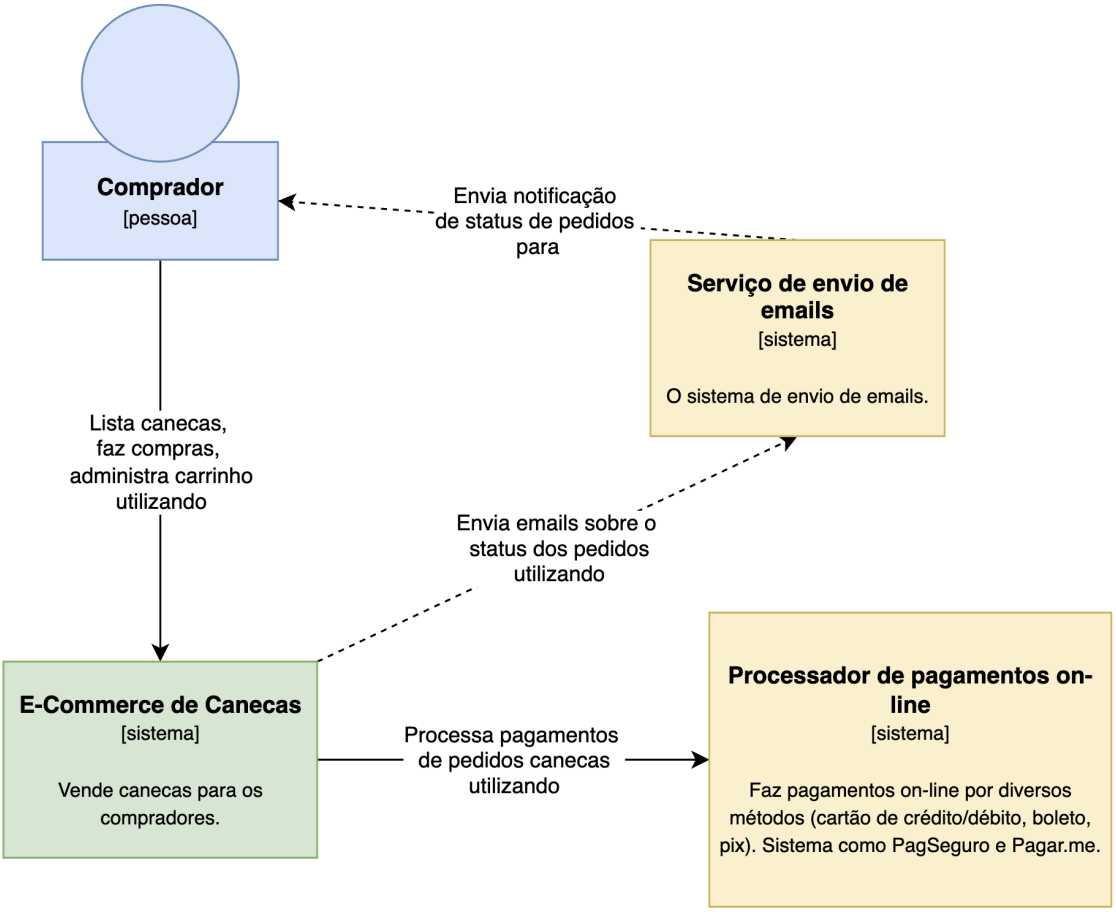
clientes, usuários, *personas* e sistemas, ao invés de tecnologias, protocolos, e outros detalhes específicos.

2. **Containers:** uma vez que é entendido como o sistema se encaixa numa estrutura geral de componentes, um próximo passo interessante é dar um "*zoom*" no diagrama de Contexto. Nesse momento, o "container" pode ser entendido como uma aplicação servidora, aplicação *mobile,* aplicação *desktop*, esquema de banco de dados, sistema de arquivos, ou qualquer outro que seja uma unidade que pode ser separada e entregue de forma unitária. O diagrama de containers mostra o formato de uma arquitetura de software em alto nível e como as responsabilidades desse sistema estão distribuídas. Também demonstra escolhas de tecnologias relevantes e como os containers se comunicam entre si.
3. **Componentes:** o diagrama de componentes é mais uma vez um "*zoom*" do diagrama de containers. Nesse é especificado separações, número de componentes, responsabilidades e camadas dentro de uma mesma aplicação.
4. **Código:** o último nível do *C4 Model* denota a diagramação alto de nível de código da aplicação, demonstrando quais classes existem, como elas se comunicam, como o código deve ser implementado, diagramas de relacionamento de entidades e outros. Esse é o maior nível de especificidade de diagramas e pode ser usado o UML (*Unified Modeling Language*) para denotar seus componentes.

2.2.2 Aplicando o C4 Model ao projeto da disciplina

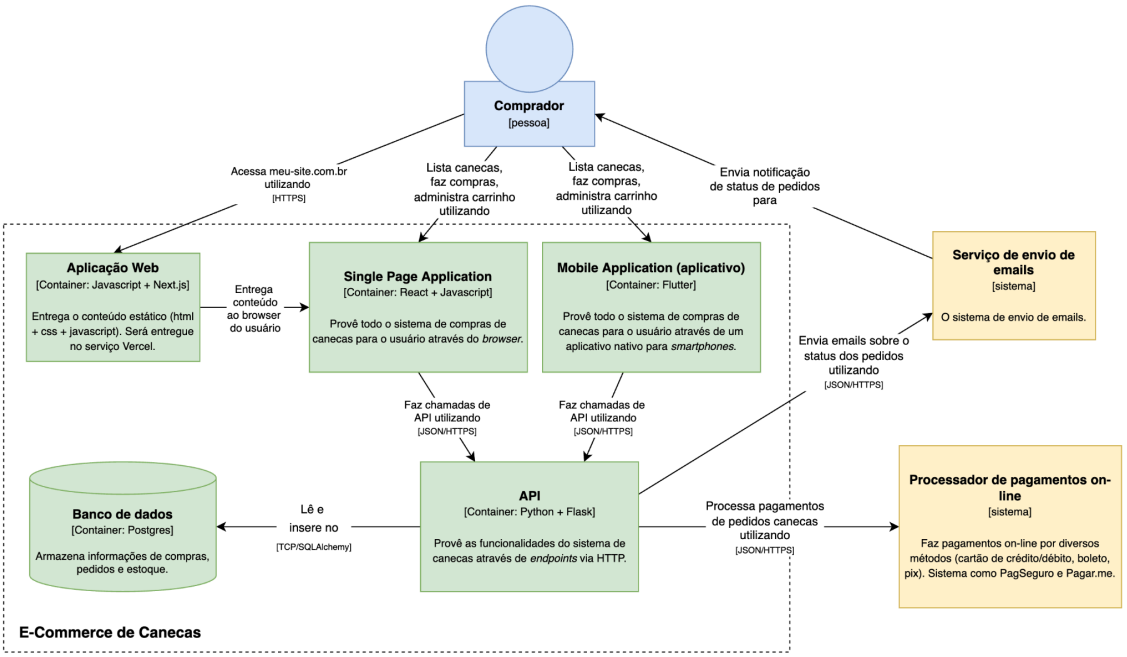
Os principais diagramas do *C4 Model* consistem nas camadas 1 e 2, respectivamente o diagrama de contexto e o de containers. As camadas 3 e 4 (diagrama de componentes e de código) são empregadas em sistemas maiores ou mais complexos, e podem servir de exercício para a prática de *system design*. Posto isso, é demonstrado na Figura 2.1 e Figura 2.2, exemplos dos dois diagramas principais do modelo, baseado nos requisitos comentados anteriormente.

Figura 2.1: Camada C1 do *C4 Model* baseado nos requisitos de exemplo



Fonte: do autor, 2022

Figura 2.2: Camada C2 do *C4 Model* baseado nos requisitos de exemplo



Fonte: do autor, 2022

Referências

GLINZ, Martin. **On Non-Functional Requirements**. 15th IEEE International Requirements Engineering Conference, 2010. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.470.5284&rep=rep1&type=pdf. Acesso em: 10 Dez. 2021.

SARAF, Pranay D.; BARTERE, Mahip M.; LOKULWAR, Prasad P. A Review on Evolution of Architectures, Services, and Applications in Computing Towards Edge Computing. In: **International Conference on Innovative Computing and Communications.** Springer, Singapore, 2022. p. 733-744.

ZAMMETTI, Frank. **Modern Full-Stack Development: Using TypeScript, React, Node. js, Webpack, and Docker**. Apress, 2020.

BROWN, Simon. **The C4 model for visualising software architecture**. Disponível em: https://c4model.com/. Acesso em: 23 dez. 2021.

React: Uma biblioteca JavaScript para criar interfaces de usuário. Disponível em: https://pt-br.reactjs.org/. Acesso em: 10 dez. 2021.

Flask: Web Development, one drop at a time. Disponível em: https://flask.palletsprojects.com/en/2.0.x/. Acesso em: 10 Dez de 2021.

Node.js: a JavaScript runtime built on Chrome's V8 JavaScript Engine. Disponível em: https://nodejs.org/en/. Acesso em: 10 Dez de 2021.

Functional Requirement. Disponível em: https://en.wikipedia.org/wiki/Functional_requirement. Acesso em: 10 Dez de 2021.

Functional Requirement. Disponível em: https://en.wikipedia.org/wiki/Non-functional_requirement. Acesso em: 10 Dez de 2021.