

**Instituto de Computação**

Departamento de Ciência da Computação

Universidade Federal da Bahia (UFBA) - Salvador, BA - Brasil

**MATA62 - Engenharia de Software I**

Prof. Eduardo Almeida

**Alunos grupo 1:** Igor Sobral ([igor.sobral@ufba.br](mailto:igor.sobral@ufba.br)); João Lucas Lima de Melo ([joaollm@ufba.br](mailto:joaollm@ufba.br)); Matheus Guimarães ([guimaraes.matheus@ufba.br](mailto:guimaraes.matheus@ufba.br)); Natan Moura ([natan.moura@ufba.br](mailto:natan.moura@ufba.br))

* [Sala Virtual do Grupo](https://meet.google.com/agq-pczq-ikv)
* [Especificações do trabalho Fase 2](https://drive.google.com/file/d/1k1R8oQnK210az6YqeR3rHNFc_ANeeMqa/view?usp=sharing)
* [Pasta Drive do Grupo](https://drive.google.com/drive/folders/1kKr1xcZSHNlDeWIQ-Op87Gt7qZiIkdPm?usp=sharing)
* [Trabalho Fase 1](https://docs.google.com/document/d/1QtfJyn9rRSQmxkkTFxIbagiX22hRdOcfV163lrcFKAw/edit?usp=sharing)
* [Apresentação Fase 1](https://docs.google.com/presentation/d/1ghFM8NwqrWUAPvH8Ahm5Nh0kj7DKn8y5YmjqllcDdHQ/edit?usp=sharing)
* [Notas Fase 1](https://drive.google.com/file/d/1NA6mHJPd4V6VdfSDrGcbWYwqRNvs2Ijv/view?usp=sharing)
* [Doc para Entrega da Fase 2](https://docs.google.com/document/d/1hauvKzRwDCwchcYR-vylFrTJxRAEuKzYH432sdMFIcc/edit?usp=sharing)
* [Apresentação Fase 2](https://docs.google.com/presentation/d/1ButAm4vtQYD_plqwlcBHJwzBcIBPmQz0JZUKnCtsA8g/edit?usp=sharing) ([vídeo](https://drive.google.com/file/d/1b27J18YDKsnhr3TzVfIzv9SUsR-wsElE/view?usp=sharing))

**Projeto Incremental de Engenharia de Software**

**PARTE II**

**O trabalho envolve analisar a arquitetura de software de aplicações open source para:**

a. Identificar uma aplicação exemplo para cada estilo arquitetural (layer, pipeline, microkernel, service-based). O projeto deve ter no mínimo 24 commits nos últimos dois anos.

b. Para essa arquitetura, mostrar os componentes da arquitetura, as características arquiteturais usadas pela aplicação, o estilo sendo usado e a sua análise crítica sobre o uso do estilo pela aplicação (se o estilo teve alguma variação na topologia e a sua concordância geral).

c. A documentação da aplicação (site, mensagens de email do projeto, lista de discussão, ferramentas de comunicação, ferramentas de bug reports) deve ser consultada.

* ***Layer***

**Projeto:** Typical REST Server

**Repositório:** [Github](https://github.com/typical-go/typical-rest-server)

**Linguagem usada:** Go

**Descrição da arquitetura:** Segundo *Fundamentals of Software Architecture*, a Arquitetura em Camadas (Layered Architecture) consiste em uma concepção de implementação por componentes organizados em camadas, performando diferentes funções na aplicação (como representação e dados).

**Descrição do projeto:** Implementação de um servidor REST típico através da linguagem de programação GO, desenvolvido seguindo os princípios da arquitetura em camadas. É utilizado ainda princípios do layout padrão de projetos em Go além de respeitar os princípios SOLID. A aplicação recebe chamadas via REST API e retorna os estados correspondentes às chamadas.

**Características de arquitetura:**

Projeto desenvolvido em torno de três camadas: *apresentação* (internal/app/controller), *lógica* (internal/app/service) e *dados* (internal/app/repo).

*Dados:*

**.** Responsável pela criação e estabelecimento dos dados da aplicação e gerenciamento do banco de dados.

**.** É definida uma estrutura Book, onde cada instância possui os atributos id, title, author updatedAt (horário da atualização) e createdAt (horário da criação).

**.** É criada uma estrutura em banco de dados para receber Book’s.

**.** As operações referente ao banco de dados são declaradas nessa camada da arquitetura. Consistem em *count* (retorna a quantidade de Book’s no banco de dados)*, find* (recebe uma lista de Book’s e a retorna com o dado buscado, caso exista)*, insert* (insere um Book no banco de dados e retorna seu id)*, bulkInsert* (insere um Book no banco de dados e retorna as linhas afetadas)*, update* (atualiza os valores de um Book no banco de dados e retorna as linhas afetadas)*, patch* (corrige dados de um Book no banco de dados e retorna as linhas afetadas)e *delete* (remove um Book no banco de dados e retorna as linhas afetadas).

*Lógica:*

**.** Responsável por definir as operações sobre as estruturas de dados da aplicação, os Book’s.

**.** É criada uma estrutura auxiliar para a comunicação com o banco de dados.

**.** As operações sobre Book são declaradas. Consistem em *create* (um Book é criado, validado, inserido na estrutura auxiliar e seu id é retornado)*, validateBook* (confirmação se os campos *title* e *author* estão inseridos), *find* (usa a estrutura auxiliar para comunicar com o servidor e fazer a consulta e retorno de uma lista de Book’s existentes no banco)*, findOne* (usa uma chamada auxiliar, utilizando a estrutura que se comunica com o banco de dados para consultar a existência de um Book)*, delete* (utilizando a estrutura auxiliar, um Book é consultado, encontrado, e removido do banco de dados)*, update* (utilizando a estrutura auxiliar, um Book tem seus atributos atualizados no banco de dados)e *patch* (utilizando a estrutura auxiliar, os dados de um Book são corrigidos).

*Apresentação:*

**.** Responsável pela comunicação entre as chamadas de API e operações solicitadas.

**.** É criada uma estrutura auxiliar para a comunicação com a camada lógica, além da comunicação com a memória cache.

**.** Para cada requisição HTTP pela API é atribuída uma operação da aplicação da camada apresentação, que se comunica com a camada lógica para realizar as devidas operações. A função responsável por essa atribuição é *SetRoute*, onde GET é associada a operação *find* (por id ou todos os elementos), HEAD é associada a operação *findOne* (por id), POST é associada a operação *Create*, PUT é associada a operação *update*, PATCH é associada a operação *patch* e DELETE é associada a operação *delete*.

*Adendo - Infraestrutura (internal/app/infra):*

**.** Não consiste, necessariamente, de uma camada da aplicação. No entanto, é responsável pela implementação, declaração e definição de propriedades de estruturas como memória cache, solicitações e banco de dados.

**Características arquiteturais relacionadas:**

*Overall Cost* - O projeto possui um escopo pequeno. Itera sobre um único tipo de estrutura de dados, BOOK’s. A arquitetura em camadas fornece um baixo custo de implementação e manutenção da aplicação.

*Simplicity* - Ainda em função da baixa complexidade da aplicação, a abordagem do uso de componentes monolíticos, as camadas, possibilita uma abordagem mais direta e fácil de gerir a nível de implementação e manutenção do projeto.

* ***Pipeline (pipe and filter)***

**Projeto:** TeeTime

**Repositório:** [Github](https://github.com/teetime-framework/TeeTime)

**Linguagem usada:** Java e C++

**Descrição da arquitetura:** Segundo [Shaw (1989)](https://dl.acm.org/doi/pdf/10.1145/75199.75222), a arquitetura *pipeline (pipe and filter)* é constituída de módulos que recebem e emitem fluxos de dados, e geralmente o fluxo de saída envolve a transformação do fluxo de entrada, o que torna eles *filters*, esses módulos são conectados pelo que chamamos de *pipes***.**

**Descrição do projeto:** A aplicação escolhida para o estilo arquitetural é a [Teetime,](https://github.com/teetime-framework/TeeTime) criada por Wulf Christian, Hasselbring Wilhem e Ohlemacher Johannes. O intuito deste framework é prover ferramentas úteis e simples para criação de aplicações genéricas com arquitetura em *pipeline* (pipe and filter). O próprio framework disponibiliza de estruturas básicas que permitem a criação de *filters*, instanciação de *pipes*, sincronização de processos e configurações necessárias para a criação das aplicações desejadas.

**Características da arquitetura:**

O projeto divide sua estrutura em quatro entidades principais: *pipes (teetime.framework.pipe), filters (teetime.stage), ports (teetime.framework)* e *configurations (teetime.framework)*.

*Pipes*: Dividem-se entre síncronos e assíncronos e são responsáveis pela conexão entre os *stages*.

*Ports*: Definem as saídas e entradas dos *stages*, protegendo-os de entradas de tipos inválidos.

*Filters:* Definidos como *stages* realizam operações sobre o fluxo de entrada e produzem uma saída correspondente.

*Configurations:* Representam configurações de conexões entre *stages*, indicando conexão entre portas, *pipes* relacionados, entre outros.

**Características arquiteturais relacionadas:**

*Simplicity* - O *framework* TeeTime tem como um dos principais objetivos permitir a criação de aplicações pipe-and-filter de maneira simples e eficaz, além de que a implementação do próprio framework é simples, legível e de fácil entendimento.

*Performance* - O TeeTime provê de paralelismo entre os *filters*, além de fornecer *Pipes* síncronos e assíncronos, que permitem melhor seleção de *threads* para execução dos *filters*, com um melhor aproveitamento do *multithreading*.

*Testability* - É possível testar cada componente da aplicação individualmente (principalmente os *filters* por funcionarem de modo independente) , além disso diversos testes estão disponíveis no próprio Git da aplicação;

*Modularity* - É possível identificar e separar os componentes da aplicação facilmente, entre *filters, sinks, source, ports, configurations ,* entre outros. Como característica marcante temos a independência dos *filters* e como eles podem ser utilizados de modo independente.

*Overall Cost* - O custo de incluir novas funções, fluxos, ordem das ações, entre outros, é bem baixo, já que só é necessário configurar um *filter* existente ou criar outros *filters* a partir das definições base.

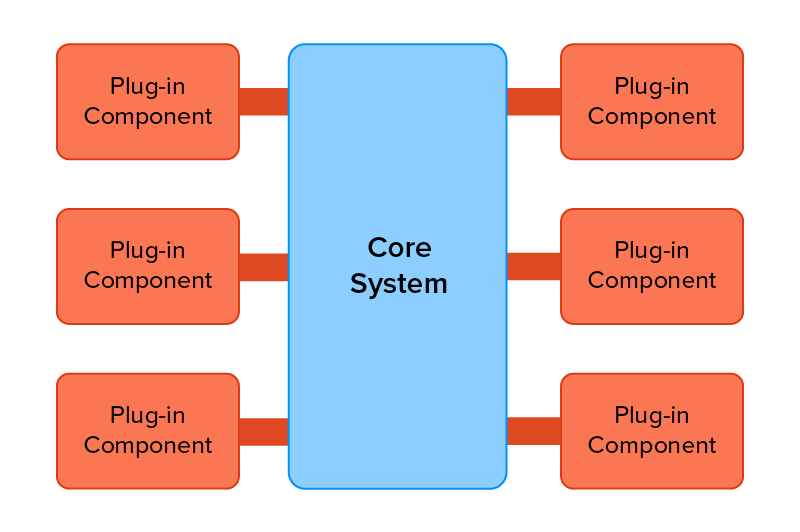
* ***Microkernel***

**Projeto:** seL4

**Repositório:** [Github](https://github.com/seL4/seL4)

**Linguagem usada:** C (Principalmente) e Python

**Descrição da arquitetura:** A arquitetura em microkernel é baseada em dois tipos componentes: o núcleo, que contém as funcionalidades mínimas necessárias, e os componentes de plug-in, que são independentes entre si e possuem novos recursos e uma maior customização. Desta forma, a representação de uma arquitetura em microkernel é dada por:



**Descrição do projeto:** A aplicação escolhida para o estilo arquitetural foi o [seL4](https://github.com/seL4/seL4), um microkernel e hipervisor (responsável por criar e executar máquinas virtuais) focado em segurança e performance, possuindo uma verificação formal sobre a corretude de sua implementação da sua especificação,sendo uma opção para sistemas de segurança crítica. Note que o seL4 não é propriamente um sistema operacional mas apenas o microkernel, uma vez que não possui diversos serviços usuais de um sistema operacional, geralmente atuando em modo usuário . É possível encontrar [plugins](https://docs.sel4.systems/projects/user_libs/) para o sel4, além de sistemas operacionais que o usam como base, por exemplo o [seL4 Core Platform](https://github.com/BreakawayConsulting/sel4cp), um sistema operacional para IoT e sistemas embarcados, e o [Neptune OS](https://github.com/cl91/NeptuneOS), um sistema operacional implementação de componentes do kernel do Windows NT, com o objetivo de fornecer suporte para executar aplicativos do Windows.

O uso de tal arquitetura em um sistema operacional ao invés de um kernel monolítico,que tenta colocar todas as funções principais do sistema dentro do kernel, tende a criar uma aplicação mais confiável e segura,uma vez que, caso ocorra um erro em modo privilegiado, não há nada no sistema para impedi-lo.

A parte de código mais relevante do sistema operacional se encontra na pasta [/src](https://github.com/seL4/seL4/tree/master/src), onde contém várias pastas importantes,como o [/kernel](https://github.com/seL4/seL4/tree/master/src/kernel), que possui informações importantes como o boot do sistema, assim como lidar com threads e erros,o [/drivers](https://github.com/seL4/seL4/tree/master/src/drivers), que gerencia a memória e timer, o [/plat](https://github.com/seL4/seL4/tree/master/src/plat), necessário para portar o sistema em diferentes plataformas, e o [/api](https://github.com/seL4/seL4/tree/master/src/api), que cuida de chamadas ao sistema e erros.

**Características da arquitetura:**

*Simplicity, Overall Cost* - Assim como as arquiteturas em Layers e em Pipeline, a arquitetura em microkernel não possui a complexidade e os custos de manutenção que arquiteturas distribuidas costumam ter.

*Performance* - Em projetos de arquitetura microkernel, a performance tende a ser um ponto positivo, uma vez que os projetos não costumam crescer muito e ser possível remover plug-ins que não estejam com funcionamento ativo com facilidade. De fato, o seL4 é o microkernel de sistema operacional com maior performance.

*Testability* - Uma vez que plu-gins são independentes, a arquitetura em microkernel possui uma boa testabilidade, já que pode-se testar um plug-in por vez.

*Scalability* - Uma vez que o núcleo é responsável por todos os processos de coordenação e comunicação entre plug-ins, um dos pontos fracos da arquitetura em microkernel é a escalabilidade, já que para, projetos muito grandes, a quantidade e complexidade dos plug-ins pode impactar o desempenho da aplicação.

Fault Tolerance - Na arquitetura em microkernel, tudo passa pelo núcleo e nada funciona sem ele. Por isso , a tolerância a falhas é um ponto negativo de um projeto, sendo importante o estudo para verificar se não há vulnerabilidade no núcleo.

* ***Service-based***

**Projeto:** [Eventide](https://eventide-project.org/)

**Repositório:** [Github](https://github.com/search?q=eventide&type=commits)

**Linguagem usada:** Ruby

**Descrição da arquitetura:** Microsserviços é um estilo arquitetural para construir aplicativos distribuídos usando contêineres. Eles recebem esse nome porque cada função do aplicativo funciona como um serviço independente. Essa arquitetura permite que cada serviço seja dimensionado ou atualizado sem interromper outros serviços no aplicativo.

“Tanto a arquitetura de microsserviços quanto a SOA são consideradas arquiteturas baseadas em serviços, o que significa que são padrões de arquitetura que enfatizam fortemente os serviços como o principal componente de arquitetura usado para implementar e executar funcionalidades comerciais e não comerciais. Embora microsserviços e SOA sejam estilos de arquitetura muito diferentes, eles compartilham muitas características.” (O’Reilly, 2022)

**Características da arquitetura:** Componentização via Serviços; Organizada em torno de Business Capabilities; Produtos (não projetos); Smart Endpoints e dumb pipes; Governança descentralizada; Gerenciamento de dados descentralizado; Automação de infraestrutura; Design for failure.

**Descrição da Aplicação:** Dentre o grupo das arquiteturas baseadas em serviço, aplicações que fazem invocações implícitas, chamadas feitas indiretamente como resposta para notificações ou outros eventos, os autores escolheram analisar o projeto *Eventide. Pôde-*se observar a utilização de diferentes estilos ao se constatar os documentos sobre o histórico do projeto.

O projeto *Eventide*, um *backbone* operacional para uma variedade de implementações, abrangendo cenários que incluem automação de marketing, tecnologia de anúncios, blockchain, mobile banking do consumidor, gestão de patrimônio, investimentos, programas afiliados, folha de pagamento, análise e transformação de dados, dentre outros, trata-se de um projeto que ampliou seu leque de serviços e se reestruturou oferecendo uma variedade de implementações com variados estilos arquiteturais em especial aplicações de microsserviços.

O projeto tem mais de 700 commits, com mais de 200 nos últimos dois anos. Ao se analisar a *Eventide* mais a fundo, vê-se um projeto do mundo real onde não há purismo de estilos, mas sim uma combinação de estilos para o que o framework se propõe a ser, adotando estilos de:

* Publish-Subscribe, com os clientes da aplicação atuando como assinantes de um conteúdo, onde os *subscribers* se registram ou cancelam o interesse e os publicadores mantém uma lista de interessados oferecendo um *broadcast* de mensagens de forma assíncrona.
* Event-sourcing, entidades de lógica de negócios projetadas a partir de fluxos de eventos com armazenamento em cache e instantâneo.
* Microsserviços, serviços baseados em mensagens hospedados em qualquer número de processos ou máquinas do sistema operacional, com consumidores pub-sub baseados em atores, grupos de consumidores, despacho de mensagens e manipuladores.

Vale relembrar que existe diferenças bem definidas nesse universo das arquiteturas baseadas em serviço, sendo a de microsserviços uma arquitetura com funções de propósito único implantadas como unidades separadas de software --- onde cada serviço possui seus próprios dados, dividindo serviços dentro de um domínio em subdomínios, com contextos restritos entre o serviço e os dados que ele possui.

Já o SBA que, embora tenha um estilo muito parecido com o de microsserviços, tem domínios de aplicativos bem definidos implantados como unidades separadas de software. Isso confere ao sistema maior granularidade. Cada serviço contém toda a funcionalidade do domínio.

A análise crítica sobre a *Eventide* feita pelos autores observa que trata-se de um projeto do mundo real, que elegeu seus estilos como uma base sólida estrutural para sua arquitetura, separando eles em fluxos, serviços, mensagens pub/sub e fornecimento de eventos.

Para os autores, o aspecto mais interessante sobre analisar a *Eventide* foi o fato de podermos observar vários usos de estilos arquiteturais em um único projeto, possibilitando uma visão mais ampla do que esse framework se propõe fazer – um armazenamento de eventos completo e um armazenamento de mensagens implementado no PostgreSQL para aplicativos Pub/Sub, além de Event Sourcing, SBAs e Microsserviços.

Conclui-se que a *Eventide* faz uma implementação minimalista dos recursos essenciais de ferramentas como Event Store ou Kafka, sendo uma fonte de eventos e um armazenamento de mensagens Pub/Sub criado no Postgres pra hospedagem simples em nuvem ou local, com suporte integrado para padrões de mensagens Pub/Sub e padrões de consumidor como grupos de consumidores.