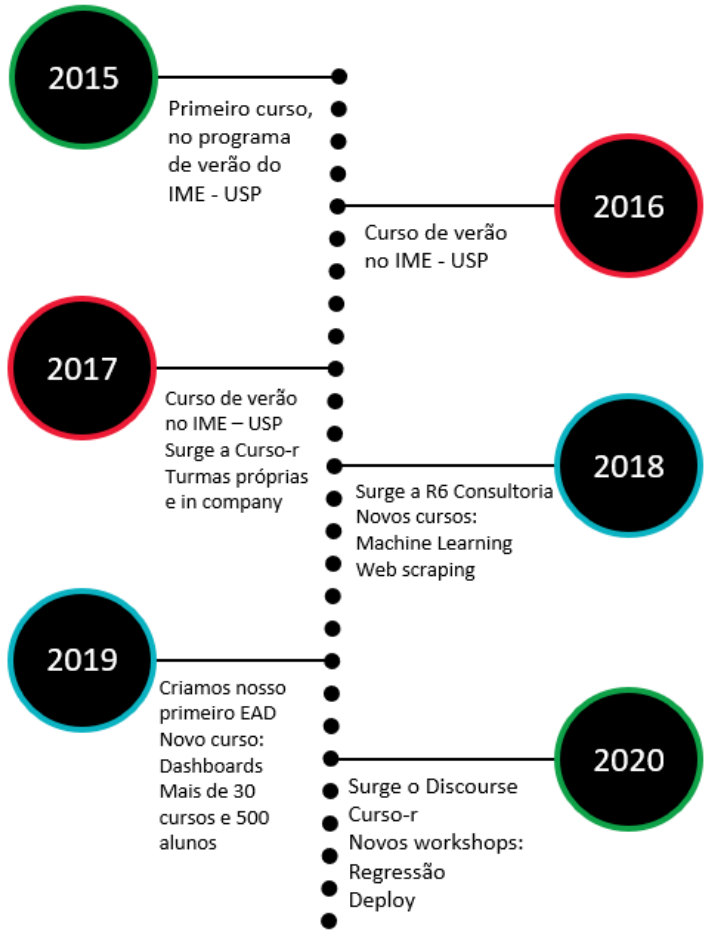
Workshop: Deploy

2020-06-20

Prólogo

Linha do tempo



Sobre o curso

- Alguns lembretes:
  - O curso ocorre das **9:00 às 18:00** com uma hora de **almoço**
  - Teremos um intervalo durante o **teste de incêndio**
  - A **gravação** do curso ficará disponível para todos por 1 ano
  - Todos se tornarão membros preferenciais no nosso **Discourse**
- Algumas sugestões:
  - O curso é longo, então reserve **água e comida** para não cansar
  - **Lave as mãos** sempre que puder e fique em casa
  - Levante para se **alongar** regularmente durante a aula

Na aula de hoje

- O que é deploy (implantação)
- O que é uma API
- O pacote `{plumber}`
- O que é uma máquina virtual
- O que é docker
- Como empacotar um dashboard
- O pacote `{golem}`
- Como automatizar um deploy

Está tudo preparado?

- Conta Google
- Cadastro no Google Cloud
- Conta GitHub
- Conta Docker Hub
- Instalação R e RStudio
- Instalação `{plumber}`, `{tidyverse}`, `{golem}`

Introdução

O que significa "deploy"?

> Implantação de software são todas as atividades que tornam um sistema disponível para uso

- No geral, colocar um software em produção envolve uma série de passos e técnicas simples e complexos
  - Tirar o código do seu computador e colocá-lo em um **servidor**
  - Permitir que o software seja **atualizado** sempre que necessário
  - Garantir a **estabilidade** do serviço levando em conta a quantidade de usuários
  - **Disponibilizar** o software de forma útil para o usuário final
  - Não perder a cabeça no caminho...

Exemplos de implantação

- Disponibilizar uma API
  - **Produto**: código que realiza uma tarefa específica dada uma entrada
  - **Objetivo**: permitir que um usuário faça uma chamada para o software e receba a resposta desejada
  - **Implantação**: servir a API em uma máquina remota
- Transformar um dashboard em um site:
  - **Produto**: código que, quando executado, exibe um dashboard interativo
  - **Objetivo**: ter um endereço fixo que, quando acessado, exibe o dashboard
  - **Implantação**: servir o dashboard em uma máquina remota

APIs

O que é uma API?

> *Application Programming Interface* (API) é uma interface de computação que define interações entre múltiplos softwares intermediários

- Essencialmente uma API é uma forma de um computador falar com outro sem precisar de um humano
- Uma API define:
  - As **chamadas e requisições** que podem ser feitas (e como fazê-las)
  - Os **formatos** de dados que podem ser utilizados
  - As **convenções** a serem seguidas
- Hoje falaremos especificamente de APIs REST em HTTP, ou seja, **APIs para serviços web**

Exemplo de API

- Um exemplo de API **sem autenticação** é a PokéAPI: https://pokeapi.co/docs/v2
- A **documentação** é provavelmente o melhor lugar para entender uma API:

Pokemon

Pokémon are the creatures that inhabit the world of the Pokémon games. They can be caught using Pokéballs and trained by battling with other Pokémon. Each Pokémon belongs to a specific species but may take on a variant which makes it differ from other Pokémon of the same species, such as base stats, available abilities and typings. See Bulbapedia for greater detail.

GET https://pokeapi.co/api/v2/pokemon/{id or name}/

- Uma API não deixa de ser um "link" que aceita parâmetros e retorna dados
    - Qual a diferença entre um site e uma API?

PokéAPI

- Este **endpoint** recebe o nome de um Pokémon e retorna uma lista de dados

```
library(httr)

resposta <- GET("https://pokeapi.co/api/v2/pokemon/ditto")
resposta
```

```
#> Response [https://pokeapi.co/api/v2/pokemon/ditto]
#>   Date: 2020-06-20 01:53
#>   Status: 200
#>   Content-Type: application/json; charset=utf-8
#>   Size: 10.4 kB
```

```
content(resposta)$moves[[1]]$move$name
```

```
#> [1] "transform"
```

Exemplo de API com autenticação

- exemplos de APIs **com autenticação** são as da NASA: https://api.nasa.gov/
- APIs podem receber parâmetros que alteram o seu comportamento (p.e. chave)

`GET https://api.nasa.gov/planetary/apod`

concept_tags are now disabled in this service. Also, an optional return parameter *copyright* is returned if the image is not public domain.

Query Parameters

Parameter	Type	Default	Description
date	YYYY-MM-DD	*today*	The date of the APOD image to retrieve
hd	bool	False	Retrieve the URL for the high resolution image
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

APOD API

- Este **endpoint** retorna a "foto astronômica do dia" para uma certa data

```
params <- list(
  date = "2019-12-31",
  api_key = NASA_KEY # Guardada no meu computador
)

resp <- GET("https://api.nasa.gov/planetary/apod", query = params)
content(resp)$url
```

```
#> [1] "https://apod.nasa.gov/apod/image/1912/M33-HaLRGB-RayLiao1024.jpg"
```

- Neste caso, ainda podemos utilizar a resposta da API para exibir uma imagem
    - Poderíamos, por exemplo, implementar um **site que consulta** essa API

O pacote {plumber}

> Um pacote R que converte o seu código R pré-existente em uma API web usando uma coleção de comentários especiais de uma linha

- Qualquer função que recebe uma entrada bem definida e retorna uma saída estruturada pode se tornar uma API
- Casos de uso:
  - Retornar entradas de uma **tabela**
  - Aplicar um **modelo** (vide https://decryptr.netlify.app/)
  - Inicializar um **processo externo**
  - Muito mais...

Exemplo de {plumber}

- Para criar uma **API local** com o `{plumber}`, basta comentar informações sobre o endpoint usando #*

```
library(plumber)

#* Escreve uma mensagem
#* @param msg A mensagem para escrever
#* @get /echo
function(msg = "") {
  paste0("A mensagem é: '", msg, "'")
}
```

- A função precisa estar salva em um arquivo para que possamos invocar os poderes do `{plumber}` no mesmo

Invocando a API

- Para implantar a API **localmente**, basta rodar os dois comandos a seguir

```
api <- plumb("arqs/exemplo_api.R")
api$run(port = 8000)
```

- A função `run()` inicializa a API em http://localhost:8000 (dependendo da **porta** escolhida)

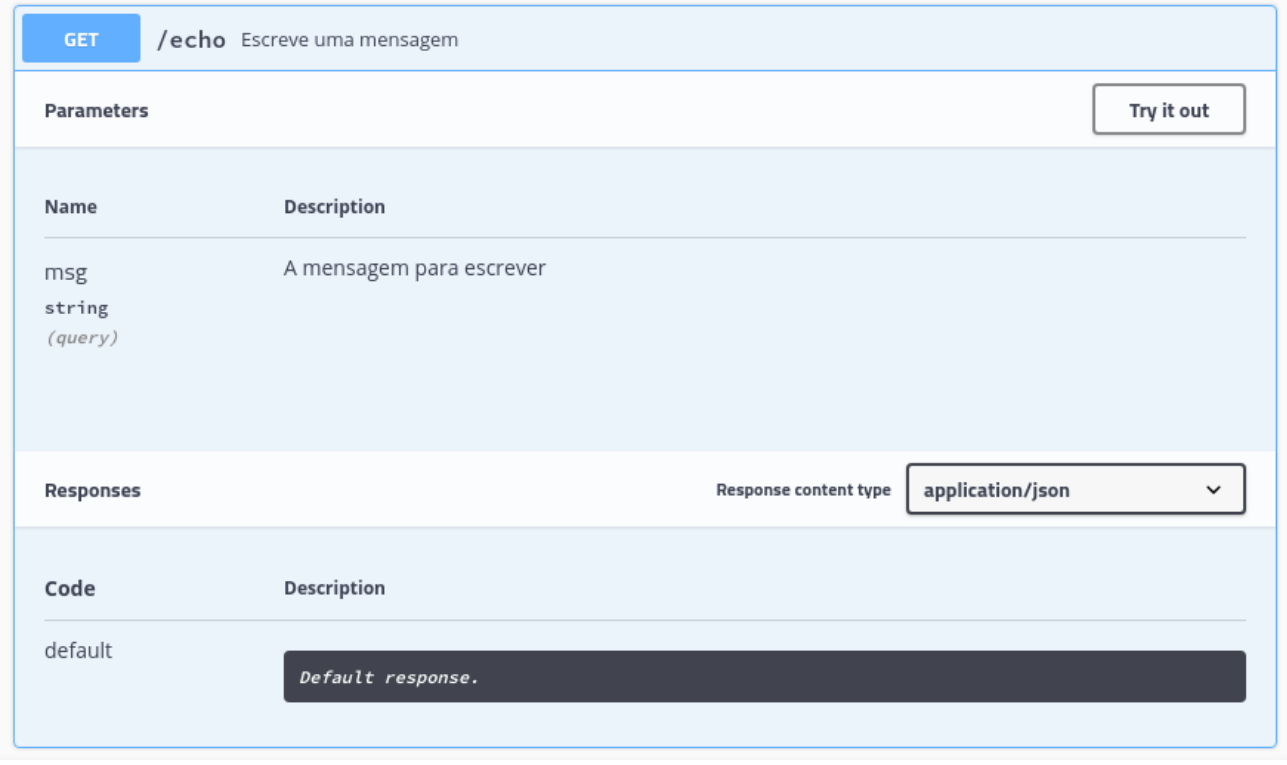
```
params <- list(msg = "Funciona!")
resp <- GET("http://localhost:8000/echo", query = params)

content(resp)[[1]]
```

```
#> [1] "A mensagem é: 'Funciona!'"
```

Swagger

- Swagger é essencialmente uma API que ajuda a criar APIs, incluindo uma interface com **documentação** em http://localhost:8000/__swagger__/



Uma nota sobre REST

> *Representational State Transfer* (REST) é um estilo de arquitetura de software que define um conjunto de restrições a serem utilizadas para criar um serviço web

- O *Hypertext Transfer Protocol* (HTTP) é a base para toda a **Web** (≠ Internet)
  - Ele define uma série de **métodos de requisição** para que um computador seja capaz de "pegar" e "mandar" conteúdo da/para a Internet
  - `GET` pega, `POST` envia e assim por diante
- REST usa os comandos HTTP para definir as mesmas operações, mas **sem estado**
  - Um site requer uma interação permanente com o usuário, enquanto uma API realiza **operações instantâneas**

Exemplo de POST

- Um **endpoint** POST normalmente recebe dados, esse é um exemplo simples

```
#* Retorna a soma de dois números
#* @param a O primeiro número
#* @param b O segundo número
#* @post /sum
function(a, b) {
  as.numeric(a) + as.numeric(b)
}
```

```
params <- list(a = 2, b = 4)
resp <- POST("http://localhost:8000/sum", body = params, encode = "json")

content(resp)[[1]]
```

```
#> [1] 6
```

Docker

O que é Docker?

> Docker é uma *platform as a service* (PaaS) que usa virtualização de sistemas operacionais para implantar softwares em "contêineres"

- O Docker não passa de um programa que roda no seu computador e permite criar e usar **contêineres**
- Contêineres são máquinas virtuais (mais sobre isso a seguir) "superficiais", acessíveis somente pela linha de comando
- Contêineres são **isolados** entre si e empacotam seu próprio **software**, bibliotecas e configuração
- Contêineres são construídos em cima de **imagens**, modelos que descrevem os componentes da máquina virtual
- Para testar, acesse https://labs.play-with-docker.com/

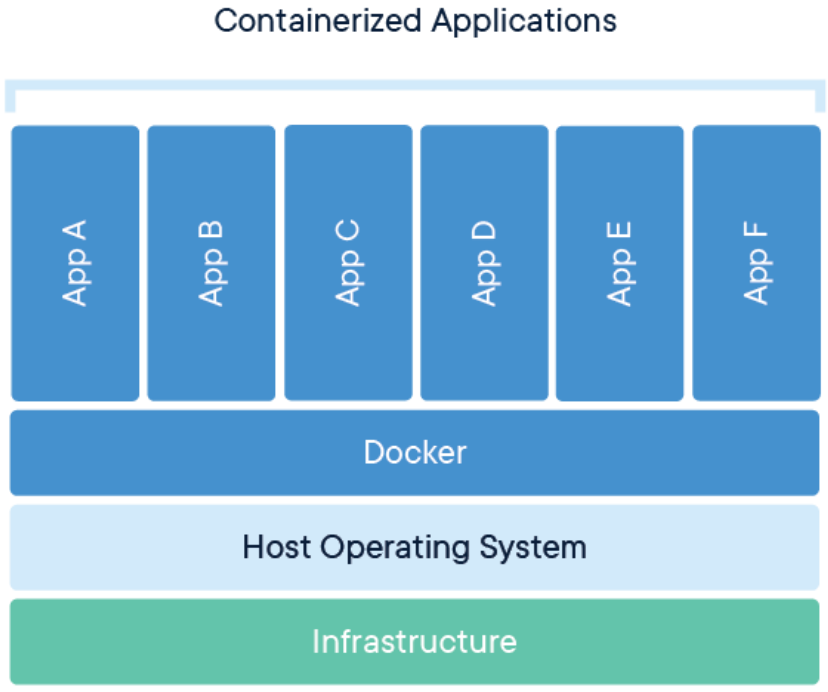
O que é uma máquina virtual?

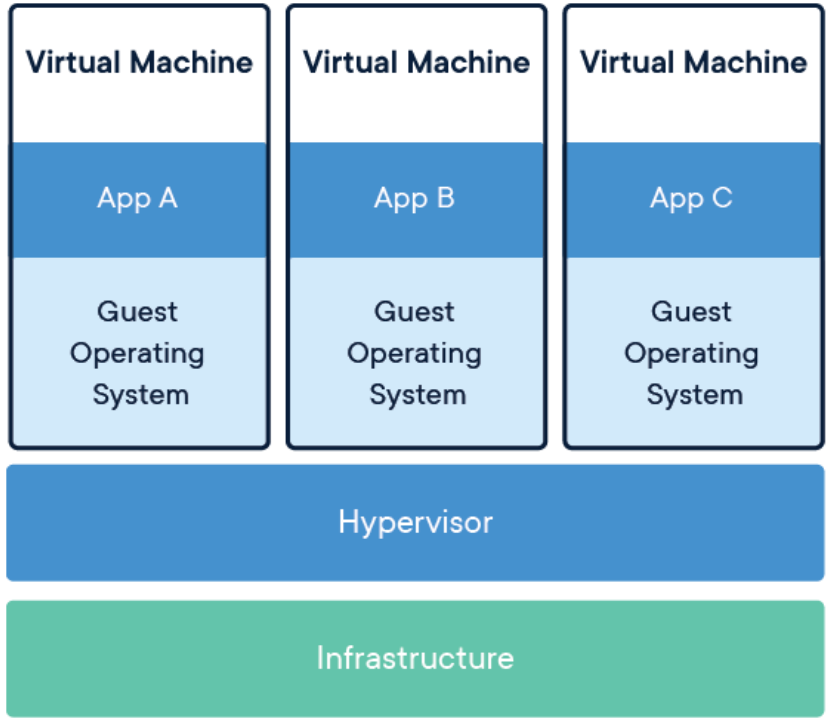
> Máquina virtual (VM) é um software que provém a funcionalidade de um computador físico, mas apenas através de emulação

- Normalmente uma máquina virtual emula um **sistema operacional** completo, desde um monitor até entradas USB
- Um hipervisor usa software nativo para simular **hardware virtual**, permitindo que código seja executado sem saber que está em uma VM
- Com uma VM é possível "criar" um computador Ubuntu dentro de um Windows e vice-versa, por exemplo
- Diferentemente de um contêiner, VMs são pesadas e "profundas", dependendo de uma imagem (ISO) para instalar o sistema operacional

Docker vs. VM

- Note as vantagens e desvantagens de cada arquitetura





Dockerfile

- Grande parte das imagens Docker já estão disponíveis no **Docker Hub** (como um CRAN do Docker)
    - Inclusive, lá estão várias imagens específicas para R, incluíndo RStudio Server, Shiny, etc. https://hub.docker.com/u/rocker
- Podemos criar uma imagem nova com um **Dockerfile**, um arquivo que especifica como ela deve ser construída
    - O primeiro componente é sempre a **imagem base** (muitas vezes um sistema operacional)
    - A seguir vêm os comandos de **configuração**
    - Por fim, o **comando** a ser executado pelo contêiner

Exemplo de Dockerfile

- A base já foi feita pelo autor do `{plumber}` e tem tudo que precisamos
- Copiamos o arquivo para **dentro do contêiner** de modo a utilizá-lo
- **Expor a porta** 8000 é necessário porque ela é onde a API será servida
- O **comando** de execução deve ser o caminho para o arquivo fonte da API (isso está descrito na documentação)

```
FROM trestletech/plumber

COPY exemplo_api.R /

EXPOSE 8000/tcp
CMD ["/exemplo_api.R"]
```

Exemplo de imagem e contêiner

- Para criar a imagem, é necessário estar dentro do diretório do Dockerfile
- O comando `docker build` monta uma imagem a partir do Dockerfile e seus arquivos associados e dá um nome para a mesma (argumento `-t`)
- O comando `docker run` executa uma imagem, criando um contêiner
  - O argumento `-p` indica a porta a ser servida no hospedeiro e a porta original
  - O argumento `--rm` limpa o armazenamento depois que tudo acaba

```
cd arqs/exemplo_api/

docker build -t exemplo .

docker run -p 8000:8000 --rm exemplo
```

Implantação contínua

> Em engenharia de software, CI/CD refere-se genericamente à combinação das práticas de integração contínua (CI) e implantação contínua (CD)

- Dado um certo código e um método consistente de implantá-lo, faz todo sentido **automatizar** o processo
- Implantação contínua normalmente envolve transferir a versão mais recente/**estável** do software e colocá-la em produção
  - O CD de um serviço encapsulado em Docker necessita automatizar o **build**
  - Existe uma série de serviços que detectam uma nova versão de um **repositório** e automaticamente criam atualizam a sua imagem
- Hoje vamos falar sobre o **Google Cloud Build** porque ele se conecta em outros serviços que vamos usar

Deploy

Google Cloud Platform

> Google Cloud Platform (GCP) é um conjunto de serviços na nuvem, incluindo processamento, armazenamento, analytics e machine learning

- A "**nuvem**" é um nome bonito para uma coleção de armazéns ao redor do mundo com computadores que podem ser alugados
    - Um **servidor** é um computador com um programa que o permite receber requisições de outros computadores
    - Um **site** é um conjunto de código sendo servido em um servidor, que pode ser convertido para uma página visual
- A Google oferece sua **infraestrutura** para ser alugada por usuários comuns
    - O GCP é a plataforma onde podemos controlar esses recursos sem nos preocuparmos com a **manutenção** do hardware e do software

Exemplo de CD no CGP

1. Menu Lateral
2. Cloud Build
3. **Acionadores**
4. Conectar repositório
5. GitHub
6. Criar **gatilho**
7. Editar gatilho
8. Verificar progresso
9. Garantir sucesso

Exemplo de deploy no CGP

1. Menu Lateral
2. IAM e administrador
3. Contas de **serviço**
4. Criar conta de serviço
5. Administrador do **Storage** + Administrador do **Compute**
6. Menu Lateral
7. Google Compute Engine
8. Criar **instância**
9. Implante uma **imagem** de contêiner nesta instância de VM

Exemplo de deploy no CGP (cont.)

1. Menu Lateral
2. Rede VPC
3. Firewall
4. Criar regra de **firewall**
5. Intervalos de IP de origem: **0.0.0.0/0**
6. Menu Lateral
7. Rede VPC
8. Endereços **IP externos**
9. Tipo: Temporário > **Estático**

Testando um deploy

> DevOps (desenvolvimento + operações de TI) tem por objetivo acelerar o ciclo de desenvolvimento e prover CD com software de alta qualidade

- Depois que o deploy estiver pronto (máquina virtual rodando, configurações realizadas) é essencial testar
- Em um ambiente corporativo em que os riscos são altos, os testes precisam ocorrer **antes** do deploy
- Muitas vezes é vital ter um **ambiente de testes** bem configurado que simule todos os problemas pelo qual o programa pode passar
    - Estamos usando a metodologia **XGH**, então testamos só depois de implantar
- Alguns testes: corretude, carga, responsividade, etc.

Testando a API

```
params <- list(msg = "Testado!")
resp <- GET("http://34.66.246.102:8000/echo", query = params)

content(resp)[[1]]
```

```
#> [1] "A mensagem é: 'Testado!'"
```

```
params <- list(a = 2, b = 6)
resp <- POST("http://34.66.246.102:8000/sum", body = params, encode = "json")

content(resp)[[1]]
```

```
#> [1] 8
```

- Ainda seria possível associar um domínio a esses IPs, mas isso (configuração de CDN) foge do tópico da aula de hoje

Shiny

Shiny empacotado

- Apps começam com uma ideia simples, mas vão **crescendo** até o ponto que não conseguimos mais entender onde estão os seus pedaços
- Com **módulos**, é possível separar pedaços de um shiny em scripts separados, que são adicionados como funções dentro do app principal
    - Um módulo pode usar funções de certo pacote, e às vezes esquecemos de checar se ele está instalado quando o app for colocado em produção
- Uma alternativa muito útil é desenvolver o shiny dentro de um **pacote**
    - As **dependências** são checadas automaticamente
    - Os módulos se tornam **funções** do pacote
    - Tudo deve ficar **documentado** e organizado por padrão

O pacote {golem}

> `{golem}` é um framework opinionado para construir aplicações shiny prontas para produção https://engineering-shiny.org

- O `{golem}` cria **templates** estruturadas que facilitam o desenvolvimento, configuração, manutenção e implantação de um dashboard shiny
  - A template é um **pacote** R, importante pelos motivos destacados antes
  - Contém uma coleção de funções que **aceleram** tarefas repetitivas
  - Possui diversos **atalhos** para criar arquivos comuns
  - Traz funções que automatizam a preparação para o **deploy**
- Eu pessoalmente acho a template muito carregada, mas muita gente gosta

Exemplo de {golem}

- A função `create_golem()` cria um projeto-pacote com toda a estrutura
  - `R/` deve conter as funções, `dev/` ajuda a montar o shiny e `inst/` fica com os recursos auxiliares

```
library(golem)

create_golem("arqs/exemplo_shiny/", package_name = "exemplo")
```

- O primeiro passo é passar pelo arquivo `dev/01_start.R` para configurar o app
- O segundo é desenvolver o app (`dev/02_dev.R` pode ajudar)
- O último passo é criar a estrutura para deploy com `dev/03_deploy.R`
  - Nunca esquecer de instalar o app e testar com `exemplo::run_app()`

```
#> ../arqs/exemplo_shiny/
#> ├── DESCRIPTION
#> ├── Dockerfile
#> ├── NAMESPACE
#> ├── R
#> │   ├── app_config.R
#> │   ├── app_server.R
#> │   ├── app_ui.R
#> │   └── run_app.R
#> ├── dev
#> │   ├── 01_start.R
#> │   ├── 02_dev.R
#> │   ├── 03_deploy.R
#> │   └── run_dev.R
#> ├── exemplo_shiny.Rproj
#> └── inst
#>     ├── app
#>     │   └── www
#>     │       └── favicon.ico
#>     └── golem-config.yml
```

Deploy II

Preparação para deploy

- Como o shiny é um pacote, podemos seguir os passos de **desenvolvimento** de pacotes antes de colocá-lo em produção
    - Rodar `devtools::check()` para garantir que tudo está **em ordem**
    - **Instalar** o app com `devtools::install()`
    - **Executar o app** em uma sessão limpa com `exemplo::run_app()`
- Quando o shiny estiver pronto, adicionar um **Dockerfile** com `add_dockerfile()`
    - O Dockerfile **não é otimizado** para o Google Cloud e isso pode implicar em alguns problemas
    - Quando necessário, edite o Dockerfile para **adequá-lo** ao ambiente real onde ele será implantado

```
add_dockerfile()
```

```
FROM rocker/r-ver:4.0.1
RUN apt-get update && apt-get install -y  git-core libcurl4-openssl-dev libgit
RUN echo "options(repos = c(CRAN = 'https://cran.rstudio.com/'), download.file
RUN R -e 'install.packages("remotes")'
RUN R -e 'remotes::install_github("r-lib/remotes", ref = "97bbf81")'
RUN Rscript -e 'remotes::install_version("config",upgrade="never", version = '
RUN Rscript -e 'remotes::install_version("golem",upgrade="never", version = "0
RUN Rscript -e 'remotes::install_version("shiny",upgrade="never", version = "1
RUN Rscript -e 'remotes::install_version("attempt",upgrade="never", version =
RUN Rscript -e 'remotes::install_version("DT",upgrade="never", version = "0.13
RUN Rscript -e 'remotes::install_version("glue",upgrade="never", version = "1.
RUN Rscript -e 'remotes::install_version("htmltools",upgrade="never", version
RUN mkdir /build_zone
ADD . /build_zone
WORKDIR /build_zone
RUN R -e 'remotes::install_local(upgrade="never")'
EXPOSE 80
CMD R -e "options('shiny.port'=80,shiny.host='0.0.0.0');exemplo::run_app()"
```

Exemplo de CD no CGP

1. Menu Lateral
2. Cloud Build
3. **Acionadores**
4. Conectar repositório
5. GitHub
6. Criar **gatilho**
7. Editar gatilho
8. Verificar progresso
9. Garantir sucesso

Exemplo de deploy no CGP

1. Menu Lateral
2. Google Compute Engine
3. Criar **instância**
4. Implante uma **imagem** de contêiner nesta instância de VM
5. Menu Lateral
6. Rede VPC
7. Endereços **IP externos**
8. Tipo: Temporário > **Estático**

Testando o shiny

- **Navegar** para o link correpondente ao IP: http://104.198.249.27
  - A **porta 80** é a padrão para o tráfego HTTP, então não há necessidade de especificar nada

Fim!

https://forms.gle/LfiFsjTarLvt9FP46