

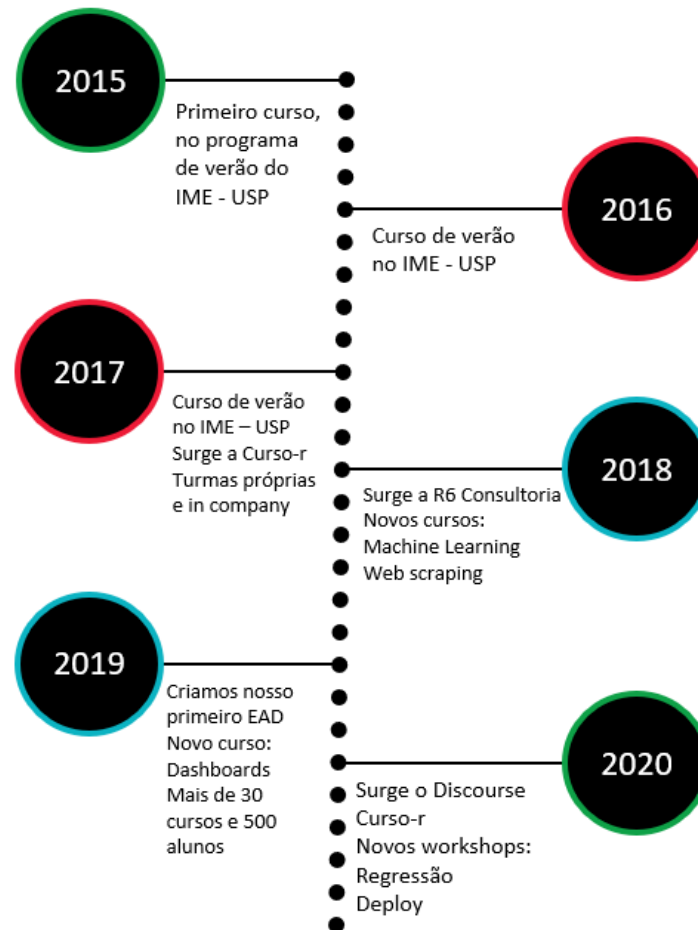
# Workshop: Deploy



Dezembro 2022

# Prólogo

# Linha do tempo



# Sobre o curso

- Alguns lembretes:
  - O curso ocorre das **19:00 às 22:00**
  - 30 minutos de monitoria antes das aulas, das **18:30 às 19:00**
  - A **gravação** do curso ficará disponível para todos por 1 ano
  - Todos se tornarão membros preferenciais no nosso **Discourse**
  - Intervalos de 10 a 15 minutos ao decorrer da aula

# Conteúdo

- O que é deploy (implantação)
- O que é uma API
- O pacote `{plumber}`
- O que é Docker
- Deploy usando **Render**
- O que é GitHub Actions
- O que é GitHub Packages
- O pacote `{golem}`
- Deploy na Google Cloud

# Está tudo preparado?

- Conta GitHub
- Conta Render
- Conta Google
- Cadastro no Google Cloud
- 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

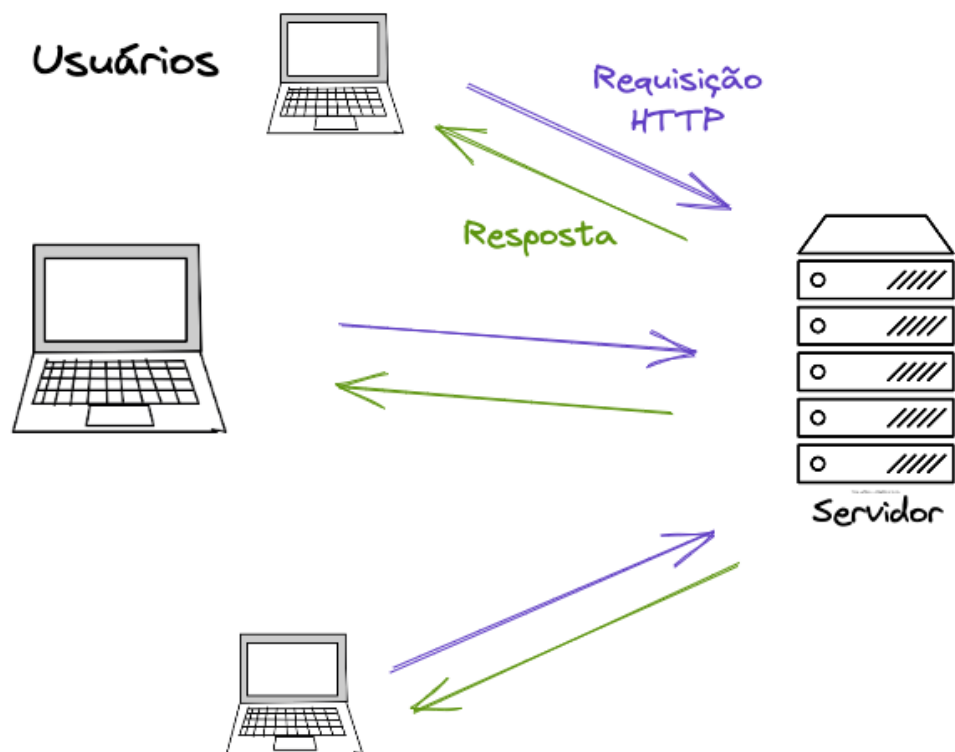
APIs

# 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**

# API HTTP



# Vantagens e desvantagens

- 👍 O usuário não precisa entender nada sobre a linguagem de programação em que ela foi desenvolvida, apenas saber fazer requisições HTTP.
- 👍 Elas podem ser executadas em servidores separados da aplicação que está consumindo-a de forma muito simples.
- 👍 Em geral, são fáceis de escalar horizontalmente, basta adicionar mais máquinas p/ atender as requisições.
- 👎 Quando a latência (tempo p/ responder) é muito importante: neste caso ter que fazer uma requisição HTTP pode ser muito caro.
- 👎 Quando você quer passar uma grande quantidade de dados. Eg, transferir arquivos com alguns GB's. Neste caso, protocolos como FTP/FTPS podem ser mais adequados.

# 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"))
```

```
#> Response [https://pokeapi.co/api/v2/pokemon/ditto]
#>   Date: 2022-12-06 21:56
#>   Status: 200
#>   Content-Type: application/json; charset=utf-8
#>   Size: 22.7 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	<i>today</i>	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
```

#> NULL

- 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/01_exemplo_api.R")  
pr_run(api, port = 8000)
```

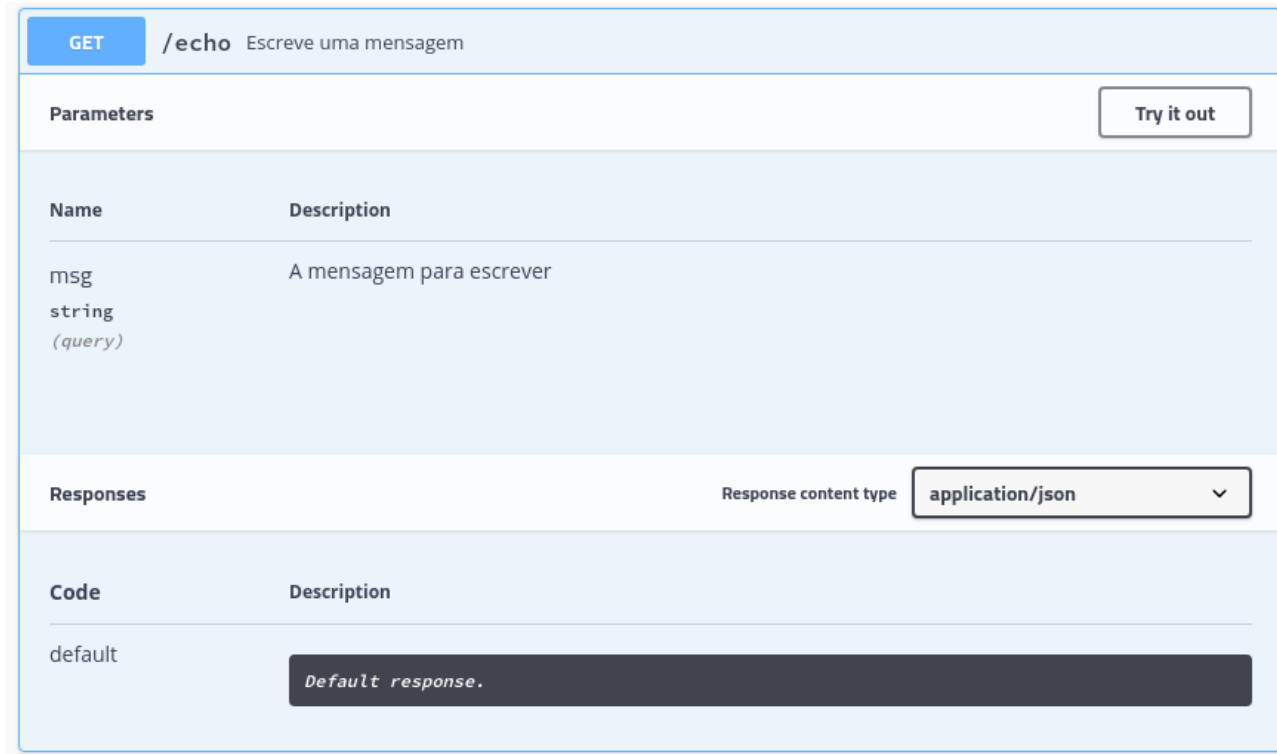
- A função `pr_run()` inicializa a API em `http://localhost:8000` (dependendo da **porta** escolhida)

```
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/\\_\\_docs\\_\\_/](http://localhost:8000/__docs__/)



The image shows a Swagger UI interface for an API endpoint. At the top, there is a blue button labeled 'GET' followed by the endpoint path '/echo' and a description 'Escreve uma mensagem'. Below this, there is a 'Parameters' section with a 'Try it out' button. The parameters table has two columns: 'Name' and 'Description'. It lists a parameter named 'msg' of type 'string (query)' with the description 'A mensagem para escrever'. Below the parameters, there is a 'Responses' section with a 'Response content type' dropdown menu set to 'application/json'. The responses table has two columns: 'Code' and 'Description'. It shows a 'default' response with a description 'Default response.' displayed in a dark box.

Name	Description
msg string (query)	A mensagem para escrever

Code	Description
default	Default response.

# 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** ( $\neq$  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 0 primeiro número  
/* @param b 0 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

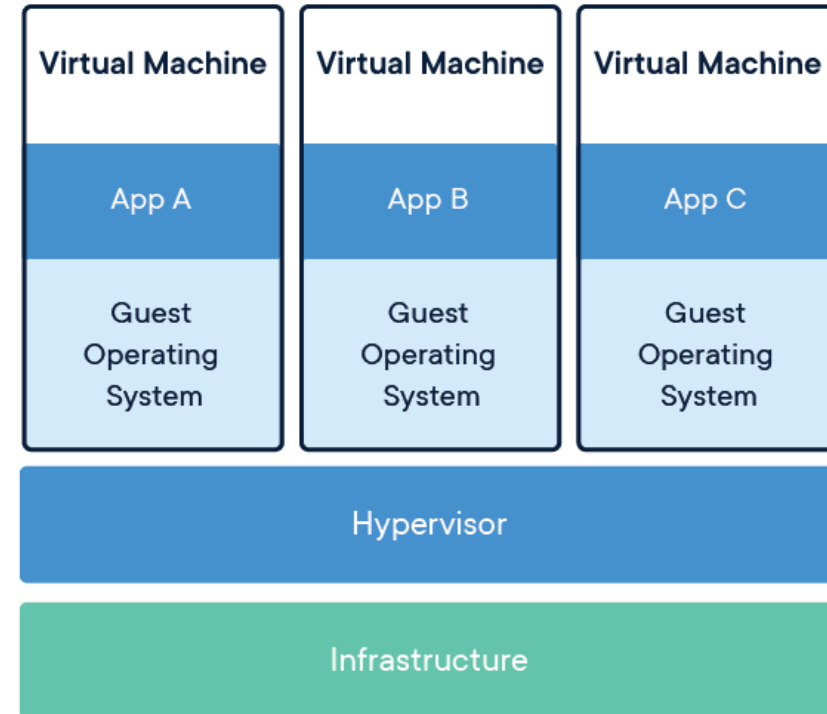
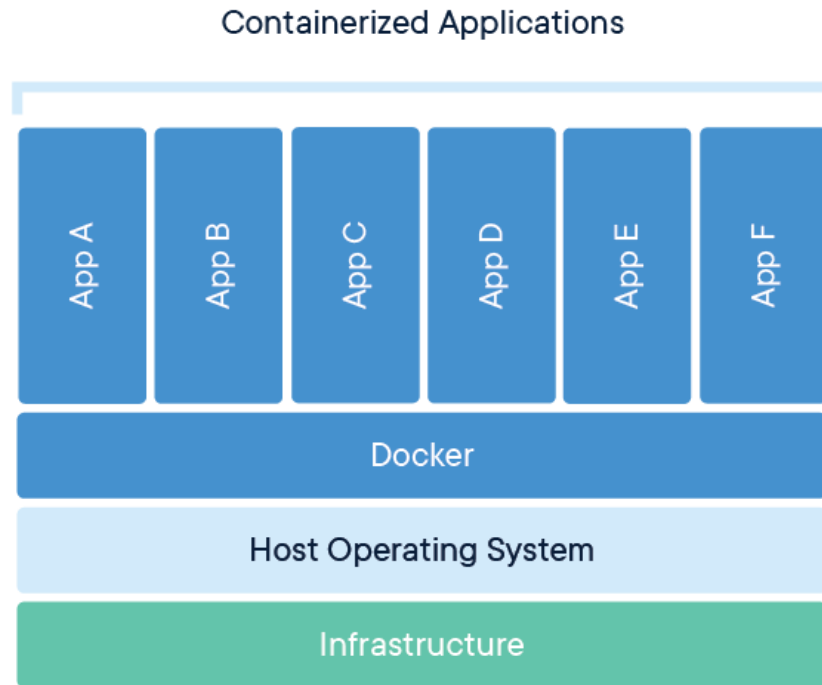
# O que é Docker?

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/>

# Docker vs. VM

- Note as vantagens e desvantagens de cada arquitetura



# Docker e o sistema operacional

- Docker roda em uma camada acima do sistema operacional hospedeiro e portanto usa recursos deste
- A principal diferença quando comparado a uma VM é que imagens construídas em um OS não são necessariamente compatíveis com outros OSs
  - **Lembre:** Dockerfiles e imagens construídos no Windows são bem diferentes dos que são construídos para sistemas Unix (Linux/macOS)
- Em geral, queremos Dockerfiles de Unix porque é muito mais prático/barato na hora de alugar um servidor na nuvem.
  - Quem quiser fazer isso no Windows > 10 pode utilizar o *Windows Subsystem for Linux* que permite executar essas imagens

# 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, incluindo 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 rstudio/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/02_exemplo_docker/  
  
docker build -t exemplo .  
  
docker run -p 8000:8000 --rm exemplo
```

Deploy no Render



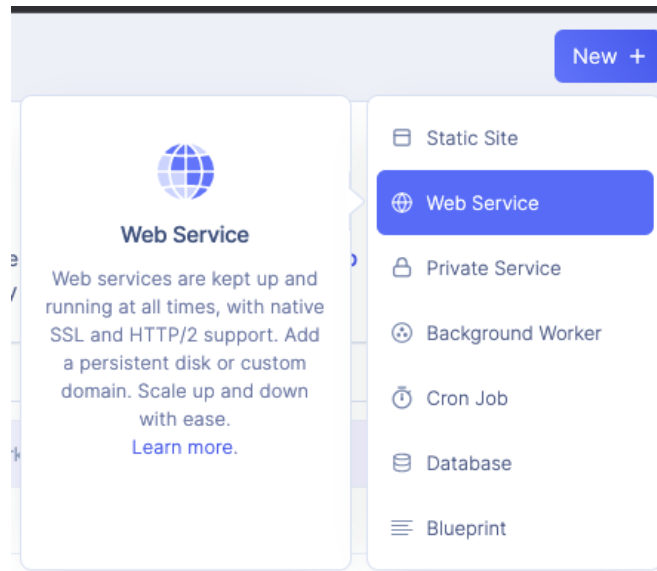
# Render

Render é uma *platform as a service* (PaaS) que permite hospedar serviços web, bancos de dados entre outros tipos de aplicações

- Tem várias funcionalidades extremamente interessantes:
  - Tem suporte para imagens **Docker**
  - Permite utilizar domínios customizados
  - Possui serviço de **autoscaling**
  - Tenta ser o mais simples possível para configurar
  - Tem bom suporte para *workflows* com Git/GitHub
- Alternativas: [Heroku](#), [Google Cloud Run](#), [Platform.sh](#) e [AWS Lambda](#)

# Implantação

- Criar um repositório no Git/GitHub com seu **Dockerfile** e código necessário para executar a sua API
- Clicar em **New > Web Service**



# Implantação (cont.)

- Selecionar o repositório na lista.
- Aguardar o *build*! 🏆

November 26, 2021 at 3:30 PM ⚙️ In progress

a675b64 use newer image

```
Nov 26 03:30:57 PM ==> Cloning from https://github.com/curso-r/plumber-render...
Nov 26 03:31:03 PM ==> Checking out commit a675b64aebd4cdcf874f34d542a36e8e76c9211e in branch master
```

# GitHub Actions

# 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 **GitHub Actions** porque ele se conecta facilmente com o GitHub

# GitHub Actions

GitHub Actions ajuda a automatizar tarefas dentro de seu ciclo de vida de desenvolvimento de software <https://docs.github.com/pt/actions>

- Um **workflow** não passa de um processo bem-definido que será executado no repositório ao qual ele pertence
- Ele é definido a partir de um arquivo YAML dentro da pasta `.github/workflows`
  - É comum definir workflows para testagem de pacotes, geração de documentação, atualização de dados, etc.
- O workflow é, essencialmente, um duende mágico que baixa o nosso repositório em um **servidor do GitHub** e executa os comandos especificados
  - O plano gratuito já funciona para bastante coisa, mas cuidado com os **custos** das máquinas MacOS

# Estrutura

- Um workflow tem alguns componentes importantes:
  - **Event:** gatilhos que ativam o workflow, podendo ser desde um push ao repositório até uma hora do dia
  - **Job:** sequências completas de comandos que podem ser executadas paralelamente entre si
  - **Step:** uma tarefa dentro de um job, composta por ações
    - **Use:** passos importados de outro repositório (úteis para setup)
  - **Action:** o átomo do workflow, um comando a ser executado pelo "duende mágico"
- Também é comum definir **env**, variáveis de ambiente para o workflow

# Exemplo de workflow

```
on: [push]                                # Event
jobs:
  R-CMD-check:                             # Job
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2           # Use
      - uses: r-lib/actions/setup-r@v1      # Use
      - name: Install dependencies          # Step
        run: |                             # Action
          install.packages(c("remotes", "rcmdcheck"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}
      - name: Check                         # Step
        run: rcmdcheck::rcmdcheck(args = "--no-manual") # Action
        shell: Rscript {0}
```



# GitHub Packages

O Package Registry é um serviço de hospedagem que permite a publicação de pacotes <https://docs.github.com/pt/packages>

- Antigamente, a única forma de publicar imagens Docker era pelo Docker Hub, mas agora temos o **Package Registry**
- Com apenas um simples **Dockerfile na raiz** do repositório, é possível subir uma imagem Docker usando uma GH Action
- Depois que a imagem estiver publicada, basta usar o comando abaixo no terminal para utilizá-la onde for necessário

```
docker pull ghcr.io/USUARIO/REPO  
docker run ghcr.io/USUARIO/REPO
```

- Basta modificar o comando run conforme a necessidade (portas, etc.)

# Publicação de imagem

```
on: [push]
jobs:
  publish-image:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: docker/login-action@v1
        with:
          registry: ghcr.io
          username: ${GITHUB_ACTOR}
          password: ${GITHUB_TOKEN}
      - name: Build the Docker image
        run: |
          docker build -t ghcr.io/USUARIO/REPO .
          docker push ghcr.io/USUARIO/REPO
```

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

```
golem::create_golem("arqs/04_exemplo_golem/", package_name = "exemplo_golem")
```

- 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_golem::run_app()`

```
#> ../arqs/04_exemplo_golem/  
#> └─ 04_exemplo_golem.Rproj  
#> └─ DESCRIPTION  
#> └─ LICENSE  
#> └─ LICENSE.md  
#> └─ NAMESPACE  
#> └─ R  
#> |   └─ _disable_autoload.R  
#> |   └─ app_config.R  
#> |   └─ app_server.R  
#> |   └─ app_ui.R  
#> |   └─ run_app.R  
#> └─ app.R  
#> └─ dev  
#> |   └─ 01_start.R  
#> |   └─ 02_dev.R  
#> |   └─ 03_deploy.R  
#> |   └─ run_dev.R  
#> └─ inst  
#> |   └─ app  
#> |   |   └─ www
```

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

# 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 `exemplodeploy::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
  - Não precisamos do GH Package porque teremos o **Container Registry!**

```
golem::add_dockerfile()
```

```
FROM rocker/verse:4.2.0
RUN apt-get update && apt-get install -y git-core libcurl4-openssl-dev libgit2
RUN mkdir -p /usr/local/lib/R/etc/ /usr/lib/R/etc/
RUN echo "options(repos = c(CRAN = 'https://cran.rstudio.com/'), download.file.method = 'libcurl', libcurl.options = 'ssl')"
```

```
RUN R -e 'install.packages("remotes")'
RUN Rscript -e 'remotes::install_version("pkgload", upgrade="never", version = "1.0.4")'
RUN Rscript -e 'remotes::install_version("shiny", upgrade="never", version = "1.7.1")'
RUN Rscript -e 'remotes::install_version("config", upgrade="never", version = "0.3.1")'
RUN Rscript -e 'remotes::install_version("golem", upgrade="never", version = "0.3.0")'
RUN mkdir /build_zone
ADD . /build_zone
WORKDIR /build_zone
RUN R -e 'remotes::install_local(upgrade="never")'
RUN rm -rf /build_zone
EXPOSE 80
CMD R -e "options('shiny.port'=80, shiny.host='0.0.0.0');exemplorgolem::run_app(
```

# Resumo do deploy no GCP

1. Menu Lateral
2. *Google Cloud Run*
3. *Create service*
4. *Continuously deploy new revisions from a source repository*
5. Escolher repositório e configurar ajustes
6. *Maximum number of instances* = 1 (para exemplo)
7. *Allow unauthenticated invocations*
8. *Container port* = 80
9. *Create*

# 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 (*build* feito, 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.

Fim!