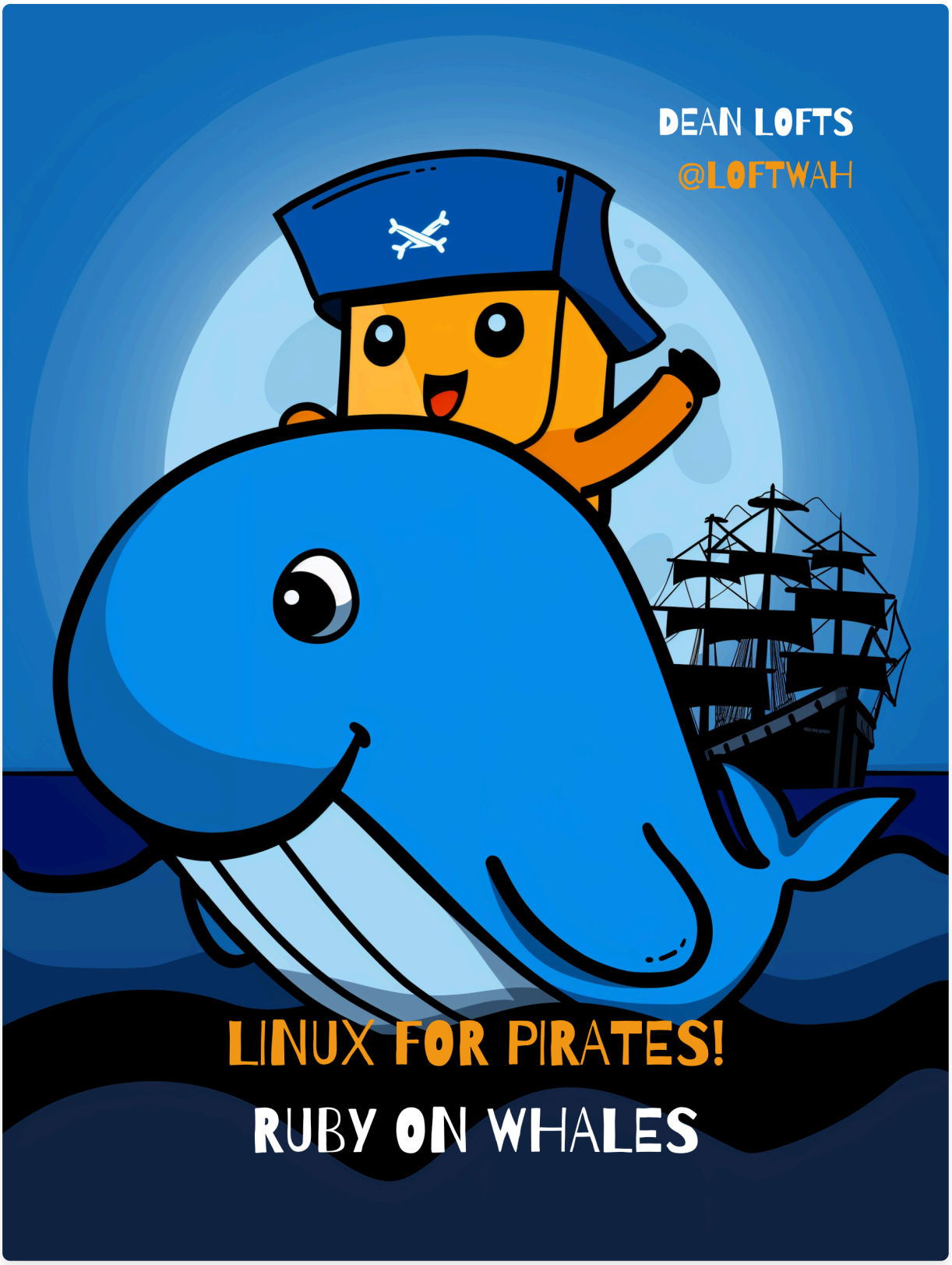

By Dean Lofts | GitHub

Linux for Pirates! Ruby on Whales

Table of Contents

Introduction

Ahoy, mateys! Welcome aboard the grand adventure that be “Linux for Pirates! Ruby on Whales” This here tome be yer trusty map through the treacherous seas of Ruby on Rails, Docker, and the mystical arts of DevOps. Whether ye be a scallywag just learning the ropes or a seasoned buccaneer, this book be the compass to guide ye through the stormy waters of modern application deployment.

What Ye Will Learn:

Setting Sail with Ruby on Rails:

- **Introduction to Rails**: Discover the lore and legend of Rails in web development.
- **Creating Your Rails Application**: Hoist the Jolly Roger and set up a new Rails project, configure PostgreSQL, and rig the sails with Tailwind CSS.

Treasure Mapping Your Project Structure:

- **Project Structure Overview**: Chart the directory layout of a Rails application.
- **Version Control with Git**: Best practices for organizing yer Git repository, managing sensitive data, and handling private gems and npm packages.

Docker – Yer Ship for the Journey:

- **Introduction to Docker**: Learn why Docker be the vessel of choice and master key concepts like images, containers, and Dockerfiles.
- **Creating a Dockerfile for Rails**: Write a Dockerfile from scratch, optimize with multi-stage builds, and understand workflows for existing Rails projects.
- **Running Yer Rails App with Docker**: Build and run Docker containers locally.

Docker Compose – Coordinatin' Yer Fleet:

- **Introduction to Docker Compose**: Explore the advantages of Docker Compose for multi-service applications.
- **Setting Up Docker Compose**: Write a docker-compose.yml file, configure services for Rails, PostgreSQL, and Redis, and manage environment variables like a true pirate captain.

Guardin' Yer Treasure – Private Dependencies:

- **Managing Private Ruby Gems**: Create and use private gems in yer Rails application, and authenticate with GitHub for secure access.
- **Handling Private npm Packages**: Create and use private npm packages, integrating them seamlessly with yer Rails application.

Secure Authentication with SSH and HTTPS:

- **SSH Key Management**: Set up and use SSH keys in Docker, understand the security implications, and explore better alternatives like PATs.
- **HTTPS Authentication**: Use GitHub Personal Access Tokens (PATs) for secure access, managing them securely in yer environment.

Automatin' Yer Deployment – CI/CD:

- **Introduction to CI/CD**: Understand the importance of continuous integration and deployment.
- **Setting Up GitHub Actions**: Write workflows for automated testing and deployment, and follow best practices for secure and efficient CI/CD pipelines.
- **Deploying with GitHub Actions and Cloud**: Step-by-step guide to deploying Rails applications with GitHub Actions and a cloud provider.

Advanced Docker Techniques:

- **Multi-Stage Builds**: Implement multi-stage builds for optimized Docker images.
- **Docker Volumes and Networks**: Manage data persistence with Docker volumes, purge data safely, and configure Docker networks for inter-container communication.

Monitoring and Scaling Yer Application:

- **Application Monitoring**: Tools and techniques for monitoring a Rails application, including Raygun, Honeybadger, Axiom, Datadog, and Vector.
- **Scaling with Docker and the cloud**: Strategies for scaling yer application, auto-scaling, and load balancing considerations.

Security Best Practices:

- **Securing Yer Docker Containers**: Identify and mitigate common security vulnerabilities.
- **Managing Secrets and Sensitive Data**: Best practices for handling sensitive information in Docker and CI/CD pipelines.

Troubleshootin' and Debuggin':

- **Common Issues and Solutions**: Troubleshoot common problems in Dockerized Rails applications.
- **Debuggin' Techniques**: Tools and methods for effective debugging in a Docker environment.

Who This Book Be For:

"Rails and DevOps with Docker for Pirates!" be perfect for:

- **Developers and DevOps Engineers**: Lookin' to deepen their understanding of Rails, Docker, and CI/CD practices.
- **Intermediate to Advanced Practitioners**: With a basic knowledge of Ruby, Rails, Docker, and DevOps, aim to level up their skills.
- **Tech Enthusiasts and Innovators**: Passionate about modern deployment practices and efficient application management.

Prerequisites

Before ye embark on this grand adventure, make sure ye have the following knowledge and tools at yer disposal:

Knowledge:

1. **Basic Programming Skills**: Ye should be comfortable with basic programming concepts and syntax.
2. **Ruby and Rails**: Familiarity with the Ruby programming language and basic Rails framework. Ye should know how to create a simple Rails application and understand MVC (Model-View-Controller) architecture.
3. **Command Line Proficiency**: Experience with using the command line interface (CLI) for navigating directories, running scripts, and managing files.
4. **Version Control with Git**: Basic understanding of Git, including creating repositories, committing changes, and pushing to remote repositories.
5. **Docker Fundamentals**: Knowledge of Docker concepts like containers, images, and Dockerfiles. Ye should be able to build and run a basic Docker container.
6. **DevOps Concepts**: Familiarity with DevOps practices, including continuous integration and deployment (CI/CD), and infrastructure as code (IaC).

Tools:

1. **Development Environment**: A computer running macOS, Linux, or Windows with a Unix-like shell (e.g., Git Bash, WSL for Windows).
2. **Ruby and Rails**: Ensure Ruby (version 3.3.0 or later) and Rails are installed on yer machine. Use version managers like RVM or rbenv for managing Ruby versions.
3. **Docker**: Install Docker Desktop for container management. Make sure Docker Compose is also installed.
4. **Git**: Have Git installed and configured with yer GitHub account for version control and accessing repositories.
5. **Code Editor**: A code editor or integrated development environment (IDE) like Visual Studio Code, Sublime Text, NeoVim or RubyMine.
6. **Package Managers**: Ensure you have package managers like npm (Node Package Manager) and yarn for managing JavaScript dependencies.

By having these prerequisites, ye'll be well-prepared to navigate the treacherous seas of modern application deployment with "Rails and DevOps with Docker for Pirates!" Ready yerself, and let the adventure begin!

Why This Book?

In this fast-paced world of tech, mastering the integration of development and operations be crucial. Docker be revolutionizin' application deployment, makin' it easier to build, ship, and run applications consistently across different environments. This book provides a hands-on, practical approach to implementin' these technologies, ensuring ye can deliver reliable, scalable, and secure applications.

Join the adventure with "Rails and DevOps with Docker for Pirates!" and embark on a journey to become a master of modern web development and deployment. Whether ye be settin' sail for the first time or seekin' to enhance yer skills, this book be yer ultimate guide to navigatin' the digital seas.

About the Author

Ahoy, me hearties! Gather 'round and let me spin ye a yarn about Dean Lofts, the legendary Loftwah of the tech and music seas. Dean be a seasoned DevOps Engineer, currently plundering the digital waves at Operoo. With over a decade of swashbuckling experience, Dean has amassed a treasure trove of skills in infrastructure management,

network security, and cloud computing, with a particular prowess in AWS. His journey through various industries and roles has forged him into a versatile and formidable force, capable of tackling the most complex technical challenges.

Professional Background

Dean's voyage began in the Royal Australian Navy, where he served as a Communications and Information Systems Sailor. For six years, he braved the high seas of satellite communication, radio operation, and LAN administration. His valor and skill earned him prestigious medals, such as the Australian Active Service Medal, the Iraq Medal, and the Australian Defence Medal. This military stint laid a strong keel for his technical expertise and disciplined approach to problem-solving.

After his naval adventures, Dean set sail into civilian waters, taking on roles like System Engineer at Unisys and Solutions Consultant at UXC Connect. At UXC Connect, he navigated high-profile projects for clients like Chevron and the Department of Defence, ensuring their systems stayed afloat and their infrastructure was shipshape.

Dean's prowess grew with roles such as Senior Test & Integration Technician at Thales and System Administrator at PVS Australia Pty Ltd. He maintained IT infrastructure, implemented network security measures, and optimized system performance, honing his skills in troubleshooting, infrastructure management, and cloud computing.

In his recent voyages, including Site Reliability Engineer at PlayHQ Sports and DevOps Engineer at CorpCloud Pty Ltd, Dean bridged the gap between development and IT operations. He wielded automation tools to create scalable and reliable software systems, championing the importance of continuous integration and continuous deployment (CI/CD) practices.

Skills and Endorsements

Dean's technical arsenal be extensive and highly endorsed. Key skills include:

- **Infrastructure and Networking**: Proven capabilities in designing and maintaining robust infrastructure and networks, with endorsements in network security and DNS.
- **DevOps and CI/CD**: Expertise in deploying and automating processes with tools like Docker, Jenkins, and GitHub Actions, ensuring efficient and secure software delivery.

- **Cloud Computing**: Skilled in leveraging AWS services for scalable and efficient solutions, demonstrated by his AWS Certified Solutions Architect – Professional certification.
- **Programming and Scripting**: Proficient in languages such as Ruby, Python, Bash, and more, with a strong focus on creating maintainable and efficient code.
- **Security**: In-depth knowledge of network security, ensuring secure and reliable operations, with certifications like CompTIA Security+.

Dean's skills have been recognized and endorsed by shipmates and landlubbers alike, highlighting his expertise and reliability.

Contributions and Achievements

Dean be a passionate advocate for open source, making significant contributions to numerous projects, including Appwrite, EddieHub, BioDrop, NASA, and WordPress. He served as an ambassador for these communities, driving technological advancement and fostering a collaborative environment. His open-source contributions reflect his commitment to sharing knowledge and improving the tech ecosystem.

One of Dean's notable feats be his integration of the yjit compiler, which delivered a 15% performance boost and substantial cost savings. This innovation showcases his ability to identify and implement improvements that have a meaningful impact on performance and efficiency.

Creative Pursuits

Beyond the tech realm, Dean be an accomplished music producer known as Loftwah The Beatsmiff. He has produced albums and collaborated with artists like Grind Mode Cypher and OptiMystic MC. His musical projects, including "Salty Waterz" and "Day of the Guiding Light," demonstrate his creative prowess and ability to blend technical precision with artistic expression.

Published Works

Dean be the author of "Linux for Pirates!", a unique and engaging book that demystifies Linux for enthusiasts and professionals alike. The book's innovative approach and accessible style have made it a valuable resource for those looking to deepen their

understanding of Linux. Dean's ability to break down complex concepts into understandable content be evident in his writing and teaching.

Personal Life

In his personal life, Dean be a dedicated father and mental health advocate. He balances his professional and creative pursuits with a commitment to his family and personal well-being. Dean enjoys beat making, reading, and curating tech-focused content on his YouTube channel, where he shares insights and tutorials with a broad audience.

Dean's journey be a testament to his relentless pursuit of excellence and his commitment to sharing knowledge with the community. His work ethic and innovative approach have earned him accolades and recognition in both the tech and music industries.

Discover More

To learn more about Dean's multifaceted journey, check out his contributions on platforms like GitHub, listen to his music on Spotify, and read his insightful tech articles. Welcome to Dean's world—a blend of rhythm, code, and a relentless drive for excellence.

Chapter 1: Setting Sail with Ruby on Rails

Introduction to Ruby on Rails

Ahoy, mateys! Before we hoist the sails and embark on our grand adventure with Ruby on Rails, let's take a moment to understand the history and significance of this mighty framework.

Ruby on Rails, often simply called Rails, was forged by the brilliant mind of David Heinemeier Hansson (DHH) back in 2004. Rails be a powerful web application framework written in the Ruby programming language, designed to make the development of web applications smoother than a calm sea. It follows the principles of Convention over Configuration (CoC) and Don't Repeat Yourself (DRY), allowing buccaneers like ye to write less code and achieve more.

Rails has charted the course for many popular web applications, including Basecamp, GitHub, and Shopify. Its treasure lies in its simplicity and productivity, enabling developers to build complex applications swiftly and efficiently.

Creating Your Rails Application

Now, let's set sail and create our Rails application. Follow these steps to ensure your ship is well-equipped for the journey ahead.

Step 1: Installing Ruby and Rails

To begin our voyage, we need to install Ruby. There be several ways to manage Ruby versions on your system. We'll cover the most popular methods: RVM, rbenv, ruby-install, and chruby. Choose the one that best suits your needs.

Using RVM (Ruby Version Manager)

1. **Install RVM**:

   Head over to the RVM website for the latest installation instructions. As of now, you can use this command to install RVM:

   ```
   curl -sSL https://get.rvm.io | bash -s stable
   ```

2. **Load RVM into your shell session**:

   ```
   source ~/.rvm/scripts/rvm
   ```

3. **Add RVM to your shell configuration file (e.g., .zshrc)**:

   ```
   echo 'source ~/.rvm/scripts/rvm' >> ~/.zshrc
   ```

4. **Install Ruby using RVM**:

```
rvm install 3.3.0
rvm use 3.3.0 --default
```

5. **Verify the installation**:

```
ruby -v
```

Other Ruby Version Managers

While RVM be a popular choice, there be other ways to manage Ruby versions. Here be a quick overview:

- **rbenv**: Lightweight and simpler to set up. Visit the rbenv GitHub page for installation details.
- **ruby-install and chruby**: For those who prefer more control and flexibility. Visit the ruby-install GitHub page and chruby GitHub page for instructions.

Each of these tools has its own way of configuring your shell (e.g., .zshrc). Here's what to do:

- **rbenv**:

```
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.zshrc
echo 'eval "$(rbenv init - zsh)"' >> ~/.zshrc
```

- **chruby**:

```
echo 'source /usr/local/share/chruby/chruby.sh' >> ~/.zshrc
echo 'source /usr/local/share/chruby/auto.sh' >> ~/.zshrc
echo 'chruby ruby-3.3.0' >> ~/.zshrc
```

Once ye've configured yer shell, restart it or run `source ~/.zshrc` to apply the changes.

Checking Your Shell Configuration

To ensure yer shell be properly set up, ye can check if the Ruby version manager is working correctly:

1. **Open a new terminal session** or run:

```
source ~/.zshrc
```

2. **Verify the Ruby version**:

```
ruby -v
```

This should return `ruby 3.3.0` if everything be set up correctly.

Step 2: Install Rails

With Ruby ready to set sail, it's time to install Rails.

1. **Install Rails**:

```
gem install rails
```

2. **Verify the installation**:

```
rails -v
```

Step 3: Creating Your Rails Application

Now, it's time to create our Rails application. We'll set it up to use PostgreSQL as the database and Tailwind CSS for styling.

1. **Create a new Rails application with PostgreSQL and Tailwind CSS**:

```
rails new pirate_app -d postgresql --css tailwind
cd pirate_app
```

2. **Configure PostgreSQL**: Open the `config/database.yml` file and configure it for PostgreSQL:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  username: postgres
  password: postgres
  host: localhost

development:
  <<: *default
  database: pirate_app_development

test:
  <<: *default
  database: pirate_app_test

production:
  <<: *default
  database: pirate_app_production
  username: pirate_app
  password: <%= ENV['PIRATE_APP_DATABASE_PASSWORD'] %>
```

> Note: your configuration for username and password may be different so if you have issues here that is where you'll need to look

3. **Create the database**:

```
rails db:create
```

4. **Start the Rails server**:

```
rails server
```

Step 4: Generating a Home Controller and View

1. **Generate a Home Controller**:

   ```
   rails generate controller Home index
   ```

2. **Set the root route**: Open the `config/routes.rb` file and set the root route to the `index` action of the `Home` controller:

   ```
   Rails.application.routes.draw do
     root 'home#index'
   end
   ```

3. **Create the Landing Page Content**: Open the `app/views/home/index.html.erb` file and add the following HTML code with Tailwind CSS classes to create a pirate-themed landing page:

```
<div class="min-h-screen bg-blue-900 flex items-center
justify-center">
  <div class="max-w-2xl bg-gray-800 text-white p-10 rounded-
lg shadow-lg">
    <h1 class="text-5xl font-bold mb-4 text-center">Ahoy,
Matey!</h1>
    <p class="text-xl mb-6 text-center">
      Welcome to Pirate App, the ultimate pirate-themed
adventure on the high
      seas!
    </p>

    <div class="flex justify-center mb-6">
      <img
        src="https://example.com/pirate-ship.jpg"
        alt="Pirate Ship"
        class="w-full h-auto rounded-lg shadow-md"
      />
    </div>

    <div class="text-center">
      <a
        href="#get-started"
        class="bg-yellow-500 text-black font-bold py-2 px-4
rounded hover:bg-yellow-600"
        >Get Started</a
      >
    </div>
  </div>
</div>
```

> Note: Replace the image with one of your own.

Step 5: Adding Navigation and Footer

1. **Add Navigation Bar**: Update `app/views/layouts/application.html.erb` to include a simple navigation bar and footer:

```
<!DOCTYPE html>
<html>
  <head>
    <title>PirateApp</title>
    <%= csrf_meta_tags %> <%= csp_meta_tag %> <%=
stylesheet_link_tag
    "application", "data-turbo-track": "reload" %> <%=
javascript_include_tag
    "application", "data-turbo-track": "reload", defer: true
%>
  </head>

  <body class="bg-blue-900 text-white">
    <header class="bg-gray-800 p-4">
      <div class="container mx-auto flex justify-between
items-center">
        <h1 class="text-2xl font-bold">PirateApp</h1>
        <nav>
          <ul class="flex space-x-4">
            <li><a href="/" class="hover:text-yellow-
500">Home</a></li>
          </ul>
        </nav>
      </div>
    </header>
    <%= yield %>
    <footer class="bg-gray-800 p-4 mt-10">
      <div class="container mx-auto text-center">
        <p>&copy; 2024 PirateApp. All rights reserved.</p>
      </div>
    </footer>
  </body>
</html>
```

With these steps, you will have a simple pirate-themed landing page for your Rails application using Tailwind CSS, complete with navigation and footer.

Avast! Ye now have a Rails application with PostgreSQL and Tailwind CSS ready to set sail!

Configuring PostgreSQL in Rails

Configuring PostgreSQL for a Rails application involves setting up the `config/database.yml` file. This file tells Rails how to connect to your PostgreSQL database in different environments (development, test, and production). Here's a detailed explanation of the configuration process and how it works in the context of a Rails application.

Step-by-Step Explanation of `database.yml`

The `config/database.yml` file contains the database configuration for different environments. Each environment (development, test, and production) can have its own settings. Here's the structure and explanation:

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  username: postgres
  password: postgres
  host: localhost

development:
  <<: *default
  database: pirate_app_development

test:
  <<: *default
  database: pirate_app_test

production:
  <<: *default
  database: pirate_app_production
  username: pirate_app
  password: <%= ENV['PIRATE_APP_DATABASE_PASSWORD'] %>
```

Default Configuration

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  username: postgres
  password: postgres
  host: localhost
```

- `default: &default` : Defines a set of default settings that can be reused across different environments. The `&default` is a YAML anchor that allows these settings to be referenced elsewhere in the file.
- `adapter: postgresql` : Specifies that PostgreSQL is the database adapter to be used.
- `encoding: unicode` : Sets the character encoding for the database to Unicode, which supports international characters.
- `pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>` : Sets the database connection pool size. It fetches the value from the `RAILS_MAX_THREADS` environment variable, defaulting to 5 if not set. The connection pool is the number of connections Rails maintains to the database.
- `username: postgres` : The database username for connecting to PostgreSQL.
- `password: postgres` : The database password for the given username.
- `host: localhost` : Specifies that the database is hosted on the local machine.

Development Configuration

```
development:
  <<: *default
  database: pirate_app_development
```

- `<<: *default` : Inherits all the settings from the `default` section using the YAML alias `*default` .
- `database: pirate_app_development` : Specifies the name of the development database. Rails will connect to this database when running in development mode.

Test Configuration

```
test:
  <<: *default
  database: pirate_app_test
```

- `<<: *default`: Inherits all the settings from the `default` section.
- `database: pirate_app_test`: Specifies the name of the test database. Rails uses this database when running tests.

Production Configuration

```
production:
  <<: *default
  database: pirate_app_production
  username: pirate_app
  password: <%= ENV['PIRATE_APP_DATABASE_PASSWORD'] %>
```

- `<<: *default`: Inherits all the settings from the `default` section.
- `database: pirate_app_production`: Specifies the name of the production database.
- `username: pirate_app`: The database username for production. This should be different from the development and test usernames for security reasons.
- `password: <%= ENV['PIRATE_APP_DATABASE_PASSWORD'] %>`: Fetches the database password from an environment variable. This ensures sensitive information is not hardcoded in the file.

Environment Variables

Using environment variables for sensitive information (like database passwords) is a best practice. It keeps these details secure and allows for easier changes across different environments (e.g., staging, production).

- **Setting Environment Variables**:

- On Unix-like systems (macOS, Linux), you can set environment variables in your shell configuration file (`.bashrc` , `.zshrc` , etc.):

```
export PIRATE_APP_DATABASE_PASSWORD=your_secure_password
```

- On Windows, you can set environment variables using the `set` command:

```
set PIRATE_APP_DATABASE_PASSWORD=your_secure_password
```

Creating and Managing the Database

Once the `config/database.yml` file is properly configured, you can create the database and start the Rails server.

1. **Create the database**:

```
rails db:create
```

   This command will create the databases specified in the `database.yml` file for the current environment.

2. **Start the Rails server**:

```
rails server
```

   This command starts the Rails application. The server will use the database configuration for the current environment (development by default).

Summary

The `config/database.yml` file is a critical part of configuring your Rails application to work with PostgreSQL. It specifies how Rails connects to the database in different environments, using settings inherited from a default configuration. By leveraging

environment variables, you can securely manage sensitive information like database passwords. Once configured, you can create the database and start the Rails server to begin development.

Rails Project Troubleshooting Steps

Even the most seasoned pirates encounter rough seas. If ye run into trouble while setting up yer Rails project, here be some common issues and troubleshooting steps to help ye navigate through the storm.

Troubleshooting Installation Issues

1. Ruby and Rails Installation Problems:

- **Issue:** Ruby or Rails not found after installation.
  - **Solution:** Verify the installation with `ruby -v` and `rails -v`. Ensure the version manager (RVM, rbenv, etc.) is properly configured in your shell profile (`.zshrc` or `.bashrc`). Restart your terminal or run `source ~/.zshrc` or `source ~/.bashrc`.

2. Bundler Errors:

- **Issue:** Bundler fails to install gems.
  - **Solution:** Ensure you have the correct version of Bundler installed. Run `gem install bundler` and then `bundle install`. Check for any specific gem installation errors and ensure all dependencies are met.

Database Configuration Issues

1. PostgreSQL Connection Problems:

- **Issue:** Rails cannot connect to the PostgreSQL database.
  - **Solution:** Verify that PostgreSQL is running. Check the `config/database.yml` file for correct database credentials. Test the connection with `psql -U postgres -h localhost`. Ensure the `pg` gem is installed by running `gem install pg`.

2. Database Creation Failures:

- **Issue:** Error running `rails db:create`.
    - **Solution:** Ensure you have the necessary privileges to create databases. Check PostgreSQL user permissions and roles. Verify that `username` and `password` in `config/database.yml` match your PostgreSQL configuration.

Rails Server Issues

1. Server Not Starting:

- **Issue:** Rails server fails to start.
    - **Solution:** Check the server logs for error messages (`log/development.log`). Ensure all necessary gems are installed and the database is properly configured. Run `rails db:migrate` to apply any pending migrations.

2. Port Conflicts:

- **Issue:** Address already in use.
    - **Solution:** Ensure no other process is using port 3000. Stop any running Rails servers or other services using the same port. Use `lsof -i :3000` to identify and kill conflicting processes.

Docker-Related Issues

1. Docker Build Failures:

- **Issue:** Docker image build fails.
    - **Solution:** Check the Dockerfile for syntax errors and correct paths. Ensure all dependencies are available and properly referenced. Use `docker build --no-cache .` to rebuild the image without using the cache.

2. Container Connection Problems:

- **Issue:** Rails application cannot connect to PostgreSQL in Docker.
    - **Solution:** Verify the `docker-compose.yml` configuration. Ensure the services are correctly defined and linked. Use `docker-compose logs` to inspect service logs for errors. Check environment variables for proper database connection details.

Common Errors and Solutions

1. Missing Gem Errors:

- **Issue:** `GemNotFound` or `LoadError`.
  - **Solution:** Run `bundle install` to install missing gems. Ensure the gem is listed in the `Gemfile`. If using private gems, verify authentication settings.

2. Asset Compilation Failures:

- **Issue:** Errors during asset precompilation.
  - **Solution:** Check for missing dependencies or incorrect paths in asset files. Ensure all necessary JavaScript and CSS libraries are installed. Run `RAILS_ENV=production rails assets:precompile` for a more detailed error output.

Finding Help

1. Rails Guides and Documentation:

- Visit the Rails Guides for comprehensive documentation and tutorials.

2. Stack Overflow:

- Search for similar issues on Stack Overflow. Use tags like `ruby-on-rails`, `postgresql`, `docker`, and `tailwind-css` to find relevant solutions.

3. GitHub Issues:

- Check the GitHub repositories of gems and libraries for reported issues and solutions.

4. Community Forums and Chat:

- Join Rails, Docker, and DevOps communities on platforms like Reddit, Discord, Twitter/X and Slack. Engaging with these communities can provide quick help and insights from experienced developers.

By following these troubleshooting steps and utilizing available resources, ye'll be better equipped to handle any issues that arise during yer Rails and Docker adventure. Happy sailing!

Summary

Ye've successfully set up Ruby and Rails on yer ship, configured PostgreSQL for yer database needs, and hoisted the Tailwind CSS sails. With yer environment shipshape, ye're ready to embark on the next leg of yer Rails adventure. Now, it's time to navigate the seas of development, deployment, and beyond!

Installing NodeJS

Installing NodeJS with NVM

To manage multiple versions of Node.js efficiently, you can use Node Version Manager (NVM). Here's how to install NVM and set it up with the LTS version of Node.js (v20.15.1):

1. **Install & Update Script**:

   You can install or update NVM using the install script. Use either of the following commands to download and run the script:

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
   ```

   ```
   wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
   ```

   Running either of these commands will clone the NVM repository to `~/.nvm` and attempt to add the following lines to your profile file (`~/.bash_profile`, `~/.zshrc`, `~/.profile`, or `~/.bashrc`):

   ```
   export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${XDG_CONFIG_HOME}/nvm")"
   [ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
   ```

2. **Load NVM**:

   After installation, you might need to restart your terminal or source your profile file to load NVM:

```
# For bash
source ~/.bashrc

# For zsh
source ~/.zshrc

# For ksh
. ~/.profile
```

3. **Verify Installation**:

   To verify that NVM has been installed, run:

```
command -v nvm
```

   This should print `nvm` if the installation was successful.

4. **Install Node.js LTS Version (v20.15.1)**:

   Once NVM is installed, you can install the LTS version of Node.js (v20.15.1):

```
nvm install 20.15.1
```

5. **Use the LTS Node.js Version**:

   To use the LTS version of Node.js, run:

```
nvm use 20.15.1
```

6. **Set the LTS Node.js Version as Default**:

To set the LTS version of Node.js as the default, run:

```
nvm alias default 20.15.1
```

Troubleshooting

If you encounter issues where `nvm` is not found after installation:

- Ensure that the profile file is sourced properly.
- Restart your terminal.
- Manually add the following lines to your profile file and source it:

```
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
```

Notes for macOS Users

For macOS users, especially those with Apple Silicon chips:

- Ensure you have Xcode command line tools installed:

```
xcode-select --install
```

- If you face issues, ensure you create the necessary profile files (`~/.bash_profile` or `~/.zshrc`) if they don't exist, and then rerun the install script.

Summary

By installing and configuring NVM, you can easily manage multiple versions of Node.js, making your development workflow more flexible and efficient. For this setup, we have used the LTS version v20.15.1 to ensure stability and long-term support.

Next Steps

In the next chapter, we'll delve deeper into organizing yer Rails project structure, setting up version control with Git, and managing private gems and npm packages. Stay the course, and ye'll be a Rails and DevOps pirate in no time!

Chapter 2: Treasure Mapping Your Project Structure

Project Structure Overview

Ahoy, me hearties! Now that we've set sail with our Ruby on Rails application, it's time to chart the course of our project structure. Understanding the directory layout of a Rails application is crucial for navigating and managing your code effectively.

When ye create a new Rails application, Rails generates a standard directory structure. Here's a brief overview of the key directories and their purposes:

- **app/**: Contains the core application code. This is where ye'll find your models, views, controllers, helpers, mailers, and jobs.
    - **assets/**: Manages your application's front-end assets like images, JavaScript, and stylesheets.
    - **controllers/**: Holds the controller classes responsible for handling requests and responses.
    - **models/**: Contains the model classes that interact with the database.
    - **views/**: Holds the view templates that render HTML.
    - **helpers/**: Contains helper modules used to clean up your views.
    - **mailers/**: Manages email delivery logic.
    - **jobs/**: Contains background job classes.
- **bin/**: Contains executable files like rails and rake.
- **config/**: Houses configuration files for the application, including database, routes, initializers, and environments.

    - **environments/**: Contains environment-specific settings (development, test, production).
    - **initializers/**: Holds files that run during application initialization.
    - **locales/**: Manages translation files for internationalization.

- **db/**: Manages database-related files.
    - **migrate/**: Contains migration files for altering the database schema.
    - **seeds.rb**: Holds seed data for populating the database.

- **lib/**: Used for extended modules and libraries.
    - **tasks/**: Contains custom rake tasks.

- **log/**: Stores application log files.

- **public/**: Holds static files like error pages and assets.

- **test/** or **spec/**: Manages test or spec files for testing the application.

- **tmp/**: Contains temporary files like cache and pid files.

- **vendor/**: Manages third-party code like plugins and gems.

Understanding this structure will help ye navigate and organize yer code effectively.

Setting Up Version Control with Git

Now, let's set up version control to keep our codebase shipshape. Git be the tool of choice for version control, and GitHub be our treasure chest for storing code.

Best Practices for Organizing Your Git Repository

Creating a GitHub Repository

If ye haven't already, head over to GitHub and sign up for an account.

Create a New Repository:

- Navigate to GitHub.
- Click on the "+" icon in the top-right corner and select "New repository".

- Fill in the repository name, description (optional), and choose whether to make it public or private.
- Click “Create repository”.

Once ye have a repository, follow these steps to get yer code into the cloud.

1. **Initialize Git**:

```
git init --initial-branch=main
```

2. **Commit Early and Often**: It be best practice to commit yer changes frequently with clear and concise commit messages. This makes it easier to track changes and roll back if needed.

```
git add .
git commit -m "Yargh Mateys"
```

3. **Add Remote Repository**: Add the GitHub repository as a remote to yer local repository. Replace `your-username` and `your-repo` with yer GitHub username and repository name.

```
git remote add origin git@github.com:loftwah-
demo/pirate_app.git
```

4. **Push to GitHub**: Push yer local commits to the remote repository on GitHub.

```
git branch -M main
git push -u origin main
```

5. **Use Branches**: Create branches for new features or bug fixes. This keeps the main branch clean and stable.

```
git checkout -b feature/new-feature
```

6. **Merge with Pull Requests**: Use pull requests to merge changes into the main branch. This allows for code review and discussion before changes are integrated.

Importance of .gitignore and Managing Sensitive Data

The .gitignore file be crucial for managing what files Git tracks. It helps keep unnecessary files out of your repository, reducing clutter and protecting sensitive information.

Never commit sensitive data such as API keys, passwords, or configuration files containing secrets. Use environment variables or a secrets management tool instead.

For Rails applications, the `.env` file is commonly used to manage environment-specific configurations. However, remember to add `.env` to your `.gitignore` to prevent it from being tracked by Git.

```
# .gitignore
.env
```

Creating and Using Private Gems and npm Packages

In our adventure, we'll create our own private gems and npm packages that we can use in our Rails application. This be an essential skill for any Rails developer, as it allows for modular and reusable code across multiple projects.

Creating a Private Ruby Gem

1. **Set Up the Gem Structure**: Create a new directory for your gem:

```
mkdir my_private_gem
cd my_private_gem
bundle gem .
```

2. **Edit the Gem Specification**: Open the `my_private_gem.gemspec` file and fill in the necessary details:

```
Gem::Specification.new do |spec|
  spec.name          = "my_private_gem"
  spec.version       = "0.1.0"
  spec.authors       = ["Your Name"]
  spec.email         = ["your.email@example.com"]
  spec.summary       = "A private gem for demonstration
purposes"
  spec.description   = "This gem is used to demonstrate using
private gems in a Rails application"
  spec.homepage      = "https://github.com/loftwah-
demo/my_private_gem"
  spec.license       = "MIT"
  spec.files         = Dir["lib/**/*.rb"]
  spec.require_paths = ["lib"]
end
```

3. **Add Functionality**: Create a simple functionality in `lib/my_private_gem.rb`:

```
module MyPrivateGem
  class Error < StandardError; end

  def self.hello
    "Hello from MyPrivateGem!"
  end
end
```

4. **Commit and Push to GitHub**: Initialize a new Git repository and push the gem to a private repository on GitHub:

```
git init --initial-branch=main
git add .
git commit -m "Initial commit"
git remote add origin git@github.com:loftwah-
demo/my_private_gem.git
git branch -M main
git push -u origin main
```

> Note: the following changes are made in the Rails application you started earlier.

Using the Private Gem in Rails

To use the private gem in your Rails application, ye'll need to create a GitHub Personal Access Token (PAT) for authentication.

1. **Create a GitHub PAT**:
    - Go to GitHub and navigate to **Settings** > **Developer settings** > **Personal access tokens**.
    - Click **Generate new token**.
    - Select the scopes `repo` and `read:packages`, then generate the token.
    - Copy the token and store it securely.
2. **Add the Private Gem to Your Rails Application's Gemfile**:

```
# Gemfile
gem 'my_private_gem', git: 'https://github.com/loftwah-
demo/my_private_gem.git'
```

3. **Configure Bundler to Use the PAT**: Create or update your `.bundle/config` file to include the GitHub PAT:

```
bundle config https://github.com/loftwah-demo
<your_github_token>
```

Alternatively, set the environment variable in your shell:

```
export BUNDLE_GITHUB__COM=<your_github_token>
```

4. **Install the Gem**: Run `bundle install` to install the gem:

```
bundle install
```

> Note: We will cover using this in your Rails application in chapter x

Creating a Private npm Package

1. **Set Up the npm Package**: Create a new directory for your npm package:

```
mkdir my-private-npm-package
cd my-private-npm-package
npm init -y
```

2. **Update `package.json`**: Edit `package.json` to include the necessary details:

```
{
  "name": "my-private-npm-package",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": ["loftwah", "linux", "for", "pirates"],
  "author": "Dean Lofts",
  "license": "ISC",
  "description": "A private package for Linux for Pirates!"
}
```

3. **Add Functionality**: Create a simple functionality in `index.js`:

```
function helloFromNpm() {
  return "Hello from my private npm package!";
}

module.exports = { helloFromNpm };
```

4. **Commit and Push to GitHub**: Initialize a new Git repository and push the package to a private repository on GitHub:

```
git init --initial-branch=main
git add .
git commit -m "Initial commit"
git remote add origin git@github.com:loftwah-demo/my-private-npm-package.git
git push -u origin main
```

Installing the Private Package in Your Rails Application

1. **Navigate to Your Rails Application Directory**:

```
cd ../pirate_app
```

2. **Create a `package.json` File**:

```
npm init -y
```

3. **Update `package.json` to Include the Private Package**:

   Open the generated `package.json` file in a text editor and modify it to include your private package:

```
{
  "name": "pirate_app",
  "version": "1.0.0",
  "description": "This README would normally document
whatever steps are necessary to get the application up and
running.",
  "main": "index.js",
  "directories": {
    "lib": "lib",
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "my-private-npm-package":
"git+https://github.com/loftwah-demo/my-private-npm-
package.git"
  }
}
```

4. **Configure npm for Authentication**:

   Create an `.npmrc` file in your Rails application's root directory to authenticate with GitHub Packages:

```
//npm.pkg.github.com/:_authToken=${GITHUB_PAT}
```

5. **Install the Dependencies**:

   Run `npm install` to install the packages defined in `package.json`:

```
npm install
```

> Note: You will be able to see the package in your `node_modules` directory and we will set this up further in chapter 5.

Summary

1. **Set Up a Private npm Package**: Created a directory, added functionality, and pushed to a private GitHub repository. Also, updated `package.json` with the necessary details.
2. **Create a `package.json` File in Rails App**: Initialized a `package.json` file and added the private package as a dependency.
3. **Configure Authentication**: Created an `.npmrc` file for GitHub package authentication.
4. **Install Dependencies**: Ran `npm install` to install the private package.

By following these steps, you've set up a private npm package, pushed it to GitHub, and configured your Rails application to use this package. This ensures your codebase is well-organized and secure, allowing you to smoothly develop your application.

Next Steps

In the next chapter, we'll dive into Docker, the mighty ship that'll carry our Rails application across the seas of deployment. We'll learn how to write Dockerfiles, build Docker images, and run our Rails app within Docker containers. Stay the course, and ye'll be a Rails and DevOps pirate in no time!

Chapter 3: Docker – Your Ship for the Journey

Introduction to Docker

Ahoy, me hearties! Docker be the sturdy ship that'll carry ye across the treacherous seas of deployment and development. In this chapter, we'll set sail with Docker, learnin' its key concepts and why it's the vessel of choice for modern developers.

What is Docker and Why Use It?

Docker be an open platform for developing, shipping, and running applications. It enables ye to separate yer applications from yer infrastructure, makin' it easier to deploy and manage them across different environments. Here be why Docker be invaluable:

- **Consistency**: Docker ensures yer application runs the same, regardless of where it be deployed.
- **Isolation**: Each application runs in its own container, avoidin' conflicts with other applications.
- **Portability**: Docker containers can be easily moved between different environments and cloud providers.
- **Efficiency**: Docker images be lightweight and start quickly, improvin' resource utilization.

Key Docker Concepts: Images, Containers, Dockerfiles

- **Images**: Immutable snapshots of yer application, includin' all dependencies and configurations.
- **Containers**: Running instances of Docker images, providin' isolated environments for yer application.
- **Dockerfiles**: Scripts that define how to build Docker images, includin' all necessary steps and configurations.

Using and Modifying the Provided Dockerfile for Rails

When ye generate a new Rails project, it comes with a Dockerfile ready to use. However, we had to make some modifications to this default Dockerfile. These changes were necessary to install a private package from GitHub using a Personal Access Token (PAT). Let's explore these modifications.

Commands Used

1. To show Rails credentials using the nano editor:

```
EDITOR=nano bin/rails credentials:show
```

> Note: you need this one to get your SECRET_KEY_BASE

2. To run the Docker container:

```
docker run -e SECRET_KEY_BASE=<my-key> -p 3000:3000
pirate_app
```

3. To build the Docker image using a secret token:

```
GITHUB_TOKEN=<my-token> docker buildx build --secret
id=GITHUB_TOKEN -t pirate_app .
```

Modifications to the Dockerfile

Here's the modified Dockerfile with the necessary changes to handle secrets and build the application:

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

# Throw-away build stage to reduce size of final image
FROM base as build

# Install packages needed to build gems
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config

# Install application gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token to install private packages
RUN --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=${GITHUB_TOKEN} && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
```

```
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Copy application code
COPY . .

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails
```

```
# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]
```

Explanation of Changes

The changes we made are here:

```
# Use secret to access GitHub token to install private packages
RUN --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=${GITHUB_TOKEN} && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile
```

We made these changes to the Dockerfile to handle the installation of private packages from GitHub using a Personal Access Token (PAT). This approach ensures that our application can securely access and install these private packages during the Docker build process.

Running Yer Rails App with Docker

Now that we've modified our Dockerfile, let's build and run our Docker containers locally.

1. **Build the Docker Image**:

```
GITHUB_TOKEN=<my-token> docker buildx build --secret
id=GITHUB_TOKEN -t pirate_app .
```

2. **Run the Docker Container**:

```
docker run -e SECRET_KEY_BASE=<my-key> -p 3000:3000
pirate_app
```

Yer Rails application should now be built into a Docker container. However, since our application relies on PostgreSQL, it won't be runnin' properly just yet. In the next chapter, we'll dive into Docker Compose to orchestrate multiple services, including our Rails app and PostgreSQL database.

Dockerfile Walkthrough

Let's walk through the Dockerfile step-by-step to understand each part and how it helps in building and running your Rails application.

Base Stage

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"
```

- **Base Image**: We start with a base image for Ruby. The version is specified using the `ARG RUBY_VERSION` argument, allowing flexibility if you need to change the Ruby version.
- **Working Directory**: The `WORKDIR /rails` command sets the working directory inside the container to `/rails`, where our Rails application will reside.
- **Environment Variables**:
    - `RAILS_ENV="production"`: Sets the Rails environment to production.
    - `BUNDLE_DEPLOYMENT="1"` and `BUNDLE_PATH="/usr/local/bundle"`: Configure Bundler to install gems in deployment mode and to a specific path.
    - `BUNDLE_WITHOUT="development"`: Excludes the development group from the bundle install.

Build Stage

```
# Throw-away build stage to reduce size of final image
FROM base as build

# Install packages needed to build gems
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config

# Install application gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token to install private packages
RUN --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=${GITHUB_TOKEN} && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Copy application code
COPY . .

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
```

```
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile
```

- **Build Stage**: This stage is used to build the application and its dependencies.
- **Install Packages**: The `RUN apt-get install` command installs necessary packages for building Ruby gems, such as `build-essential`, `git`, `libpq-dev`, `libvips`, and `pkg-config`.
- **Install Gems**:
  - `COPY Gemfile Gemfile.lock ./`: Copies the Gemfile and Gemfile.lock into the container.
  - `RUN --mount=type=secret,id=GITHUB_TOKEN ... bundle install`: Uses a GitHub token to install private gems, ensuring the token remains secret during the build.
  - `rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache "${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git`: Cleans up unnecessary files to reduce the image size.
  - `bundle exec bootsnap precompile --gemfile`: Precompiles bootsnap code for faster boot times.
- **Copy Application Code**: `COPY . .` copies the entire application code into the container.
- **Precompile Assets**: Precompiles Rails assets for production without requiring the master key by using a dummy key.

Final Stage

```
# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails

# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]
```

- **Final Stage**: This stage creates the final image that will be used to run the application.
- **Install Deployment Packages**: Installs necessary packages for running the Rails application, such as `curl`, `libvips`, and `postgresql-client`.

- **Copy Artifacts**: Copies the built artifacts (gems and application code) from the build stage to the final image.
- **Set Up User and Permissions**:
  - `useradd rails --create-home --shell /bin/bash`: Adds a non-root user named `rails` for security.
  - `chown -R rails:rails db log storage tmp`: Changes ownership of important directories to the `rails` user.
  - `USER rails:rails`: Switches to the `rails` user for running the application.
- **Entrypoint and CMD**:
  - `ENTRYPOINT ["/rails/bin/docker-entrypoint"]`: Sets the entrypoint script to prepare the database.
  - `EXPOSE 3000`: Exposes port 3000 for the application.
  - `CMD ["./bin/rails", "server"]`: Starts the Rails server by default.

Running Your Rails App with Docker

Now that we've modified our Dockerfile, let's build and run our Docker containers locally.

1. **Build the Docker Image**:

```
GITHUB_TOKEN=<my-token> docker buildx build --secret
id=GITHUB_TOKEN -t pirate_app .
```

2. **Run the Docker Container**:

```
docker run -e SECRET_KEY_BASE=<my-key> -p 3000:3000
pirate_app
```

Yer Rails application should now be built into a Docker container. However, since our application relies on PostgreSQL, it won't be runnin' properly just yet. In the next chapter, we'll dive into Docker Compose to orchestrate multiple services, including our Rails app and PostgreSQL database.

Summary

Ye've successfully explored the provided Dockerfile for yer Rails application, made necessary modifications to handle private package installations, and run yer application within a Docker container. With Docker as yer ship, ye be ready to navigate the seas of deployment and beyond!

Next Steps

In the next chapter, we'll dive into Docker Compose, coordinatin' multiple services like a true pirate captain. We'll set up a `docker-compose.yml` file, configure services for Rails, PostgreSQL, and Redis, and manage environment variables. Stay the course, and ye'll be a Rails and DevOps pirate in no time!

Chapter 4: Docker Compose – Coordinating Your Fleet

Introduction to Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. This simplifies orchestration, streamlines the development process, and ensures consistency across different stages of development and deployment.

Setting Up Docker Compose

Docker Compose uses a `docker-compose.yml` file to define and configure multiple services. We'll set up a multi-service application for Rails and PostgreSQL. Additionally, we'll handle environment variables securely using `.env` files.

Writing a docker-compose.yml File for a Multi-Service Application

To manage your development environment effectively, we'll create a `docker-compose.yml` file. This file will configure services for Rails, PostgreSQL, and handle secrets securely.

Why the Original Dockerfile is Suitable for Production

The original Dockerfile is optimized for production environments:

- **Multi-stage builds**: This reduces the size of the final image by only including necessary artifacts.
- **Environment-specific configurations**: Ensures the environment variables are set for production.
- **Security considerations**: Runs the application as a non-root user.

Here's the production Dockerfile:

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

# Throw-away build stage to reduce size of final image
FROM base AS build

# Install packages needed to build gems
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config

# Install application gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token
RUN --mount=type=bind,target=. \
    --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=$(cat /run/secrets/GITHUB_TOKEN) && \
    git config --global
```

```
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Copy application code
COPY . .

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
```

```
USER rails:rails

# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]
```

Why We Need a Separate Dockerfile for Development

Using the same Dockerfile for both production and development can lead to inefficiencies and complexities. The development Dockerfile (`Dockerfile.dev`) is optimized for:

- **Rapid Iteration**: Allows for quick rebuilding and live code reloading.
- **Development Tools**: Includes tools and dependencies needed for development, not required in production.

Here's the development Dockerfile:

```
# Dockerfile.dev
# syntax=docker/dockerfile:1

ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

WORKDIR /rails

# Set development environment
ENV RAILS_ENV="development" \
    BUNDLE_PATH="/usr/local/bundle" \

# Install packages needed to build gems and development tools
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config nodejs yarn

# Copy only the Gemfiles for installing gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token for private repositories and
install gems
RUN --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=$(cat /run/secrets/GITHUB_TOKEN) && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install

# Copy the rest of the application code
COPY . .
```

```
# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Start the server by default
EXPOSE 3000
CMD ["./bin/rails", "server", "-b", "0.0.0.0"]
```

Writing the docker-compose.yml File

The `docker-compose.yml` file coordinates multiple services needed for the application. Here's the complete file, which includes configurations for the Rails app and PostgreSQL:

```
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile.dev
      secrets:
        - GITHUB_TOKEN
    environment:
      DATABASE_URL:
postgres://postgres:postgres@db:5432/pirate_app_development
    ports:
      - "3000:3000"
    volumes:
      - .:/rails
    depends_on:
      - db

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: pirate_app_development
    volumes:
      - db_data:/var/lib/postgresql/data

secrets:
  GITHUB_TOKEN:
    environment: GITHUB_TOKEN

volumes:
  db_data:
```

Managing Environment Variables

Using `.env` files for environment-specific configurations helps in maintaining different settings for development and production environments. Docker Compose automatically

reads the `.env` file located in the same directory as the `docker-compose.yml` file and uses the variables defined within it.

Ensure your `.env` file contains the necessary environment variables:

```
# .env
GITHUB_TOKEN=<your-github-token>
SECRET_KEY_BASE=<your-secret-key-base>
```

Running the Development Environment with Docker Compose

With the `docker-compose.yml` and `.env` files set up, you can build and run the development environment using Docker Compose.

To build the services:

```
docker compose build
```

To start the services:

```
docker compose up
```

> Note: Use `docker compose up -d` to run the application in the background.

Summary

- **Production Dockerfile**: Optimized for production with multi-stage builds, security considerations, and environment-specific configurations.
- **Development Dockerfile (`Dockerfile.dev`)**: Tailored for development with necessary development tools and rapid iteration support.
- **Docker Compose**: Simplifies the orchestration of the development environment, ensuring consistency and ease of setup.

- **Environment Variables**: Managed using an `.env` file for secure and environment-specific configurations.

By following these instructions, you can effectively use Docker Compose to manage your development environment, leveraging the power of Docker to ensure consistency and ease of setup.

Chapter 5: Guarding Your Treasure – Using Private Dependencies

Ahoy, me hearties! Now that we've set up our Rails application, it's time to put our private packages to use. This chapter covers integrating private Ruby gems and npm packages into our Rails application.

Using Private Ruby Gems

Let's use the private Ruby gem we created earlier in our Rails application (part of this will already be done if you were following along).

1. **Add the Gem to Your Gemfile**:

```
gem 'my_private_gem', git: 'https://github.com/loftwah-demo/my_private_gem.git'
```

2. **Configure Bundler to Use the PAT**:

```
bundle config https://github.com <your_github_token>
```

3. **Install the Gem**:

```
bundle install
```

4. **Use the Gem in Your Rails Application**: In `app/controllers/application_controller.rb`, add:

```
class ApplicationController < ActionController::Base
  def test_my_private_gem
    render plain: MyPrivateGem.hello
  end
end
```

5. **Add a Route to Test the Gem**: In `config/routes.rb`, add:

```
Rails.application.routes.draw do
  get 'test_my_private_gem', to:
'application#test_my_private_gem'
end
```

6. **Verify the Gem is Working**: Start your Rails server and navigate to `http://localhost:3000/test_my_private_gem`. You should see:

```
Hello from MyPrivateGem!
```

Using Private npm Packages

Let's integrate the private npm package into our Rails application.

1. **Navigate to Your Rails Application Directory**:

```
cd ../pirate_app
```

2. **Create a `package.json` File** (we may have already done this if you're following along):

```
npm init -y
```

3. **Update `package.json` to Include the Private Package**:

```
{
  "name": "pirate_app",
  "version": "1.0.0",
  "dependencies": {
    "my-private-npm-package":
"git+https://github.com/loftwah-demo/my-private-npm-
package.git"
  }
}
```

4. **Configure npm for Authentication**: Create an `.npmrc` file in your Rails application's root directory:

```
//npm.pkg.github.com/:_authToken=${GITHUB_TOKEN}
```

5. **Install the Dependencies**:

```
npm install
```

Setting Up Vite for JavaScript Bundling

1. **Install the `vite_rails` Gem**:

   Add the `vite_rails` gem to your Gemfile:

```
gem 'vite_rails'
```

Then run `bundle install`:

```
bundle install
```

2. **Install Vite with Rails**:

   Initialize Vite in your Rails application:

   ```
   bundle exec vite install
   ```

   Install npm dependencies and build the Vite project:

   ```
   npm ci
   npx vite build
   ```

3. **Run Rails and Check**:

   Start the Rails server and Vite development server:

   ```
   rails s
   bin/vite dev
   ```

4. **Shut Down Rails and Vite**:

   Before proceeding, shut down the Rails server and Vite development server by pressing `Ctrl+C` in the terminal where they are running.

5. **Install Foreman (Development Only)**:

   Foreman is a tool to manage Procfile-based applications. Add it to your Gemfile for the development group:

```
group :development do
  gem 'foreman'
end
```

Then run `bundle install`:

```
bundle install
```

6. **Set Up `Procfile.dev`**:

   If not already present, create a `Procfile.dev` in the root of your project directory with the following content:

```
web: bin/rails server
css: bin/rails tailwindcss:watch
vite: bin/vite dev
```

7. **Run Foreman**:

   Use Foreman to start all processes defined in `Procfile.dev`:

```
bin/bundle exec foreman start -f Procfile.dev
```

This will start your Rails server, Tailwind CSS watcher, and Vite development server all at once, making your development workflow more efficient.

5. **Shut Down Foreman**:

   Before proceeding, shut down Foreman and any running processes by pressing `Ctrl+C` in the terminal where they are running.

6. **Update `docker-compose.yml`**:

   To run the necessary processes using Docker, update your `docker-compose.yml` file as follows:

```
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile.dev
      secrets:
        - GITHUB_TOKEN
    environment:
      SECRET_KEY_BASE: ${SECRET_KEY_BASE}
      DATABASE_URL:
postgres://postgres:postgres@db:5432/pirate_app_development
    ports:
      - "3000:3000"
    volumes:
      - .:/rails
    depends_on:
      - db
    command: bin/rails server -b 0.0.0.0

  vite:
    build:
      context: .
      dockerfile: Dockerfile.dev
      secrets:
        - GITHUB_TOKEN
    volumes:
      - .:/rails
    depends_on:
      - web
    command: bin/vite dev

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: pirate_app_development
    volumes:
      - db_data:/var/lib/postgresql/data
```

```
secrets:
  GITHUB_TOKEN:
    environment: GITHUB_TOKEN

volumes:
  db_data:
```

> Note: I use `foreman` to develop locally and this is my preference, with `docker compose` being a secondary option.

Summary

You've successfully set up Vite with Rails using the `vite_rails` gem and integrated your development environment to use Docker for managing multiple processes. This setup ensures that your JavaScript is bundled correctly and can take advantage of Vite's features.

Next Steps

In the next chapter, we'll dive into secure authentication with SSH and HTTPS, managing secrets, and using GitHub Personal Access Tokens (PATs) for secure access. Stay the course, and you'll be a Rails and DevOps expert in no time!

Chapter 6: Secure Authentication with SSH and HTTPS

Ahoy, Mateys! Welcome to the Treasure of Secure Authentication

SSH Key Management

Setting up and Using SSH Keys in Docker

Arrr, ye landlubbers, we be needin' to set sail with secure authentication using SSH keys in Docker. Here be the steps to hoist the Jolly Roger and fetch private treasures (repositories):

1. **Generate an SSH Key**: We be generatin' a new key so we don't mess with yer existing `id_rsa`.

   ```
   ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_pirate_key -C "your_email@example.com"
   ```

   This command creates a new key pair (`id_pirate_key` and `id_pirate_key.pub`).

2. **Add the SSH Key to Your SSH Agent**: The SSH agent be a daemon that holds yer private keys in memory. We be usin' it so yer private keys don't be needin' to be entered every time.

   ```
   eval $(ssh-agent -s)
   ssh-add ~/.ssh/id_pirate_key
   ```

3. **Add Your SSH Key to GitHub**: Copy the public key to yer clipboard and add it to yer GitHub account.

```
cat ~/.ssh/id_pirate_key.pub
```

Configuring SSH for Accessing Private Repositories

Swab the decks and prepare to clone private repositories using SSH mounts in Docker. This be the Dockerfile we be usin′ to fetch our private treasures.

Dockerfile for SSH Authentication

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

# Throw-away build stage to reduce size of final image
FROM base AS build

# Install packages needed to build gems
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config openssh-client

# Add SSH key to the build environment
ADD id_pirate_key /root/.ssh/id_pirate_key
RUN chmod 600 /root/.ssh/id_pirate_key && \
    ssh-keyscan github.com >> /root/.ssh/known_hosts

# Install application gems
```

```
COPY Gemfile Gemfile.lock ./

# Use SSH to access private GitHub repo
RUN --mount=type=ssh \
    GIT_SSH_COMMAND='ssh -i /root/.ssh/id_pirate_key' bundle install
&& \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Copy application code
COPY . .

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails
```

```
# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails

# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]
```

Explanation

1. **Generate and Add SSH Key**: We create a new SSH key named `id_pirate_key` to avoid conflicts with existing keys. This key be added to yer SSH agent for secure use.
2. **SSH Agent**: The SSH agent keeps yer keys in memory, makin' it easy to use 'em without repeatedly enterin' the passphrase.
3. **Dockerfile**: We be usin' an `ADD` command to include the SSH key in the Docker build environment and `RUN --mount=type=ssh` to securely clone private repositories.

Why Use SSH?

- **Alternative to PAT**: While we typically use PATs for secure access, SSH keys offer another method for those who prefer it.
- **Automation**: Docker builds with SSH mounts allow seamless access to private repositories without manual intervention.

With this knowledge, ye be ready to set sail with secure SSH authentication in Docker. Remember, a true pirate guards his keys well and uses them wisely to fetch the richest treasures from the seas of code!

> Note: I tend to avoid using SSH in my Docker images and containers.

Chapter 7: CI/CD – Automating Your Deployment

Introduction to CI/CD

Ahoy, mateys! In this chapter, we'll dive into the importance of Continuous Integration (CI) and Continuous Deployment (CD). These practices ensure that your code is always in a deployable state, making it easier to catch bugs early and deploy new features quickly and reliably.

Why CI/CD?

- **Consistency**: Ensures code changes are integrated and tested frequently.
- **Speed**: Automates the deployment process, reducing manual effort and errors.
- **Quality**: Enhances code quality through automated testing and validation.

Setting Up RSpec Locally

1. **Add RSpec to your `Gemfile`**:

```
group :development, :test do
  gem "rspec-rails"
end

group :test do
  gem "capybara"
  gem "selenium-webdriver"
  gem "factory_bot_rails"
  gem "faker"
end
```

2. **Install the gems**:

```
bundle install
```

3. **Generate RSpec configuration**:

```
rails generate rspec:install
```

4. **Run RSpec to ensure it's set up correctly**:

```
bundle exec rspec
```

Setting Up GitHub Actions for CI/CD

1. **Create a GitHub Actions Workflow**:
    - Create a new file in the `.github/workflows` directory named `ci.yml`.
2. **Define the Workflow for Testing**:

```yaml
name: CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:16
        ports:
          - 5432:5432
        env:
          POSTGRES_DB: pirate_app_test
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres

    steps:
      - uses: actions/checkout@v4

      - name: Set up Ruby
        uses: ruby/setup-ruby@v1
        with:
          ruby-version: 3.3.0

      - name: Install dependencies
        env:
          BUNDLE_GITHUB__COM: ${{ secrets.GH_PAT }}
        run: |
          gem install bundler
          bundle config set --local github.com "x-access-
token:${{ secrets.GH_PAT }}"
          bundle install
```

```
      - name: Set up database
        run: |
          bundle exec rails db:create db:migrate
        env:
          RAILS_ENV: test
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres

      - name: Run tests
        run: bundle exec rspec
        env:
          RAILS_ENV: test
```

3. **Add `GH_PAT` as a Repository Secret**:
    - Navigate to your GitHub repository.
    - Go to `Settings` > `Secrets and variables` > `Actions`.
    - Click `New repository secret`.
    - Name it `GH_PAT` and paste your personal access token.
    - Ensure the token has `repo` and `package:read` scopes.

Explanation

1. **Create a GitHub Actions Workflow**:
    - Create a new file named `ci.yml` in the `.github/workflows` directory.
2. **Define the Workflow for Testing**:
    - Use `actions/checkout@v4` to ensure compatibility with the latest Node.js versions.
    - Use `ruby/setup-ruby@v1` to set up Ruby 3.3.0.
    - Use `BUNDLE_GITHUB__COM` environment variable to authenticate with the personal access token (`GH_PAT`).
    - Install dependencies with `bundle install`.
    - Set up the database with `bundle exec rails db:create db:migrate`.
    - Run tests with `bundle exec rspec`.
3. **Add `GH_PAT` as a Repository Secret**:

- Ensure the personal access token has the necessary permissions and add it as a secret to your repository.

By following these steps, you'll ensure that your CI/CD pipeline is set up correctly to test your Rails application with RSpec, and that your private gem repository is accessible using the provided personal access token.

Best Practices for Secure and Efficient CI/CD Pipelines

- **Secrets Management**: Store sensitive information like API keys and database passwords securely using GitHub Secrets.
- **Environment Variables**: Use environment variables for configuration settings.
- **Caching Dependencies**: Cache dependencies to speed up the build process.

Deploying with GitHub Actions and DigitalOcean using `doctl`

To automate the deployment of your Rails application, we'll use `doctl` to manage DigitalOcean and GitHub Actions to handle the deployment process.

Setting Up `doctl` for DigitalOcean

1. **Install `doctl`:**
   - On macOS:

     ```
     brew install doctl
     ```

   - On Ubuntu:

     ```
     snap install doctl
     ```

   - Download and install from GitHub for other systems.
2. **Create a DigitalOcean API Token:**

- Go to the DigitalOcean API page and create a new token with read and write access.
- Save the token securely as it will be needed to authenticate `doctl`.

3. **Authenticate `doctl`:**

```
doctl auth init
```

- Follow the prompts to enter your API token.

- **Generate SSH Keys:**

  If you don't already have SSH keys, you can generate them using the following command:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

- Follow the prompts to save the keys in the default location (`~/.ssh/id_rsa`).

- **Add SSH Key to DigitalOcean:**

```
doctl compute ssh-key import loftwah --public-key-file
~/.ssh/id_rsa.pub
➜  pirate_app git:(main) doctl compute ssh-key import loftwah
--public-key-file ~/.ssh/id_rsa.pub
ID          Name       FingerPrint
42754072    loftwah
76:a5:d8:77:d2:86:52:14:f7:ea:9b:b6:94:f2:b7:fc
```

> Note: The ID from here is what you need below as .

- Replace `<KEY-NAME>` with a name for your SSH key.

- **Create a Droplet:**

```
doctl compute droplet create <DROPLET-NAME> --region syd1 --
image ubuntu-22-04-x64 --size s-1vcpu-1gb --ssh-keys <SSH-
KEY-ID> --wait
```

- Replace `<DROPLET-NAME>` with your desired droplet name.
- Replace `<SSH-KEY-ID>` with your SSH key ID. You can find this by running `doctl compute ssh-key list`.

```
doctl compute droplet create pirateapp --region syd1 --image
ubuntu-22-04-x64 --size s-1vcpu-1gb --ssh-keys 42754072 --
wait
```

It will give you the following output or you can use the below command to retrieve the IP address if you need it.

```
  pirate_app git:(main) doctl compute droplet create
pirateapp --region syd1 --image ubuntu-22-04-x64 --size s-
1vcpu-1gb --ssh-keys 42754072 --wait
  ID           Name         Public IPv4       Private IPv4
Public IPv6    Memory    VCPUs    Disk    Region    Image
VPC UUID                                 Status    Tags
Features                           Volumes
  432279856    pirateapp    170.69.169.169    10.126.0.2
1024      1        25      syd1      Ubuntu 22.04 (LTS) x64
102fec11-b9ca-4d95-9870-9619086e2408    active
droplet_agent,private_networking
```

- **Retrieve Droplet IP:**

```
doctl compute droplet list
```

- Note down the IP address of your newly created droplet.

- **Delete your instance if you're not ready to use it**

```
doctl compute droplet delete pirateapp -f
```

- ** If you are ready connect to the instance with SSH**

```
ssh -i ~/.ssh/id_rsa root@<your-ip>
```

Configuring Your Droplet for Docker

1. Connect to Your Droplet

```
ssh -i ~/.ssh/id_rsa root@<your-ip>
```

2. Update and Upgrade System

```
sudo apt-get update && sudo apt-get upgrade -y
```

3. Install Docker Using the Script

```
# Download the Docker installation script
curl -fsSL https://get.docker.com -o install-docker.sh

# Verify the script's content (optional but recommended)
cat install-docker.sh

# Run the script with --dry-run to verify the steps it executes
(optional)
sh install-docker.sh --dry-run

# Run the script to install Docker
sudo sh install-docker.sh
```

4. Start Docker and Enable on Boot

```
sudo systemctl start docker
sudo systemctl enable docker
```

5. Add Your User to Docker Group

```
sudo usermod -aG docker ${USER}
su - ${USER}
```

6. Verify Installations

```
# Check Docker
docker --version
docker run hello-world
```

Vite in Docker

If you have been following along we have broken our `Dockerfile` for production. I will go through and figure this out but for now we will have this section that explains what we need for it to build properly for Vite. Please update your `Dockerfile` to the following.

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

# Throw-away build stage to reduce size of final image
FROM base AS build

# Install packages needed to build gems and Node.js
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config curl && \
    curl -fsSL https://deb.nodesource.com/setup_20.x | bash - && \
    apt-get install --no-install-recommends -y nodejs

# Install application gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token
RUN --mount=type=bind,target=. \
    --mount=type=secret,id=GITHUB_TOKEN \
```

```
    GITHUB_TOKEN=$(cat /run/secrets/GITHUB_TOKEN) && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Install Node.js packages
COPY package.json package-lock.json ./
RUN npm install

# Copy application code
COPY . .

# Run Vite build
RUN npx vite build

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
```

```
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails

# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]
```

Setting Up Docker Compose

Instead of running on the system directly we can run the application using Docker Compose. Create a `docker-compose.yml` file in your application directory:

> Note: This is on the droplet you will be deploying to.

```
services:
  db:
    image: postgres:16
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: pirate_app_production
      POSTGRES_USER: pirate_app
      POSTGRES_PASSWORD: ${PIRATE_APP_DATABASE_PASSWORD}

  redis:
    image: redis:7

  web:
    build: .
    command: bundle exec rails server -b 0.0.0.0
    volumes:
      - .:/rails
    ports:
      - "80:3000"
    environment:
      RAILS_ENV: production
      SECRET_KEY_BASE: ${SECRET_KEY_BASE}
      DATABASE_URL:
postgres://pirate_app:${PIRATE_APP_DATABASE_PASSWORD}@db:5432/pira
te_app_production
      REDIS_URL: redis://redis:6379/1
    depends_on:
      - db
      - redis

volumes:
  postgres_data:
```

Setting Up GitHub Actions

Create a GitHub Actions workflow file (e.g., `.github/workflows/deploy.yml`):

```
name: Deploy

on:
  workflow_run:
    workflows: ["CI"]
    types:
      - completed

jobs:
  deploy:
    runs-on: ubuntu-latest
    if: ${{ github.event.workflow_run.conclusion == 'success' }}

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to GitHub Container Registry
        uses: docker/login-action@v2
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GH_PAT }}

      - name: Build and push Docker image
        run: |
          echo "${{ secrets.GH_PAT }}" | docker login ghcr.io -u
${{ github.actor }} --password-stdin
          GITHUB_TOKEN=${{ secrets.GH_PAT }} docker buildx build -
-secret id=GITHUB_TOKEN -t ghcr.io/loftwah-
demo/pirate_app/pirateapp:latest --push .

      - name: Deploy to droplet
        uses: appleboy/ssh-action@master
        with:
          host: 170.64.189.40
          username: root
```

```
          key: ${{ secrets.DROPLET_SSH_PRIVATE_KEY }}
          script: |
            echo "${{ secrets.GH_PAT }}" | docker login ghcr.io -u
${{ github.actor }} --password-stdin
            docker compose -f /root/docker-compose.prod.yml --env-
file /root/.env down
            docker compose -f /root/docker-compose.prod.yml --env-
file /root/.env pull
            docker compose -f /root/docker-compose.prod.yml --env-
file /root/.env up -d
```

Setting Up Environment Variables in GitHub Secrets

Add the following secrets to your GitHub repository:

- `GH_PAT`
- `DROPLET_SSH_PRIVATE_KEY`
- `PIRATE_APP_DATABASE_PASSWORD`
- `SECRET_KEY_BASE`

How to Add Secrets:

- **GITHUB_TOKEN**:
    1. Go to your GitHub profile.
    2. Navigate to Settings > Developer settings > Personal access tokens.
    3. Generate a new token with `repo` and `write:packages` permissions.
    4. Copy the token and add it as a secret in your repository settings.
- **DIGITALOCEAN_API_TOKEN**:
    1. Go to the DigitalOcean API page and create a new token.
    2. Copy the token and add it as a secret in your repository settings.
- **DIGITALOCEAN_SSH_KEY_ID**:
    1. Add the public key to your DigitalOcean account under Security > SSH Keys.
    2. Retrieve the key ID using `doctl compute ssh-key list`.
    3. Add the key ID as a secret in your repository settings.

> Note: Your application should be deployed but because we deployed the production environment we require a secure connection. You can modify the Rails config in your application if you want to see this in `config/environments/production.rb` where you need to change `config.force_ssl = false`. For info: I ended up updating this in my repo because it is only a demo application and who cares right? We can still set up SSL with this set to `false`.

Summary

By following these steps, you can set up a CI/CD pipeline with GitHub Actions and automate the deployment of your Rails application on DigitalOcean using `doctl` and GitHub Container Registry. This approach simplifies the deployment process, making it easier to manage infrastructure and deploy updates efficiently. Happy sailing, and may your deployments be swift and smooth!

Chapter 8: Advanced Docker Techniques

Ahoy, Mateys! The Secrets of Multi-Stage Builds

Benefits of Multi-Stage Builds for Docker Images

Arrr, ye scallywags, gather ‘round and lend an ear! The seas be rough, and our ship must be lean and mean. Multi-stage builds be our secret weapon, a true marvel of Docker wizardry. By breakin’ down our build into stages, we be savin’ space and keepin’ our final image as trim as a corsair’s blade. No more bloatin’ with unnecessary artifacts, aye, just the essentials to sail smooth and fast.

Implementin’ Multi-Stage Builds in Your Dockerfile

To start, ye be needin’ a Dockerfile more cunning than a fox in a henhouse. First, we be usin’ one stage to build our treasures, then another to copy only the finest loot into our final image. Here be an example from the ship’s log:

```
# syntax=docker/dockerfile:1

# Make sure RUBY_VERSION matches the Ruby version in .ruby-version
and Gemfile
ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

# Rails app lives here
WORKDIR /rails

# Set production environment
ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

# Throw-away build stage to reduce size of final image
FROM base AS build

# Install packages needed to build gems and Node.js
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config curl && \
    curl -fsSL https://deb.nodesource.com/setup_20.x | bash - && \
    apt-get install --no-install-recommends -y nodejs

# Install application gems
COPY Gemfile Gemfile.lock ./

# Use secret to access GitHub token
RUN --mount=type=bind,target=. \
    --mount=type=secret,id=GITHUB_TOKEN \
```

```
    GITHUB_TOKEN=$(cat /run/secrets/GITHUB_TOKEN) && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

# Install Node.js packages
COPY package.json package-lock.json ./
RUN npm install

# Copy application code
COPY . .

# Run Vite build
RUN npx vite build

# Precompile bootsnap code for faster boot times
RUN bundle exec bootsnap precompile app/ lib/

# Precompiling assets for production without requiring secret
RAILS_MASTER_KEY
RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

# Final stage for app image
FROM base

# Install packages needed for deployment
RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
```

```
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

# Copy built artifacts: gems, application
COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

# Run and own only the runtime files as a non-root user for security
RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails

# Entrypoint prepares the database.
ENTRYPOINT ["/rails/bin/docker-entrypoint"]

# Start the server by default, this can be overwritten at runtime
EXPOSE 3000
CMD ["./bin/rails", "server"]

# Build command
GITHUB_TOKEN=<my-token> docker buildx build --secret id=GITHUB_TOKEN
-t pirate_app .
# Run command
docker run -e SECRET_KEY_BASE=<my-key> -p 3000:3000 pirate_app
```

With this strategy, our final image be as swift as a schooner, ready to plunder the seven seas!

Docker Volumes and Networks: Keepin' Our Booty Safe and Secure

Managing Data Persistence with Docker Volumes

Aye, the life of a pirate be full of treasures, and we must keep 'em safe! Docker volumes be our chests to store precious data. They be independent of the container lifecycle, so our treasures don't disappear into the briny deep when a container meets Davy Jones. Here's how ye be creatin' a volume:

```
docker volume create my_volume
```

Mount it in yer container like so:

```
docker run -v my_volume:/path/in/container my_image
```

Now, our loot be safe and sound, even if the seas get stormy!

Purging Data and Considerations

But beware, ye don't want to be caught with too much ballast! When the time comes to purge old data, make sure ye know what ye be doin'. Deleting a volume be permanent, like sendin' it to the depths of the ocean. To remove a volume:

```
docker volume rm my_volume
```

Think twice before ye hit that command, or ye might be losin' vital booty.

Configurin' Docker Networks for Inter-Container Communication

To navigate the vast waters of inter-container communication, Docker networks be our charts and compass. Ye can set up a bridge network for yer containers to talk like old sea dogs:

```
docker network create my_network
```

Run yer containers on this network:

```
docker run --network my_network --name container_one my_image
docker run --network my_network --name container_two my_image
```

Now, `container_one` and `container_two` be chattin' like old mates in a tavern, sharin' secrets and plans for their next big raid.

So there ye have it, me hearties! Master these advanced Docker techniques, and ye'll be the scourge of the seven seas, feared by all and master of your digital domain! Arrr!

Chapter 9: Monitoring and Scaling Your Application

Avast, Me Hearties! Application Monitoring on the High Seas

Tools and Techniques for Monitoring a Rails Application

Ahoy, ye salty dogs! Keepin' a sharp eye on yer Rails application be as crucial as watchin' the horizon for enemy ships. Here be a list of fine tools to help ye monitor the health and performance of yer app:

- **Raygun**: Catch errors and crashes faster than a cannonball flyin' across the deck. Raygun gives ye detailed reports on where yer app be takin' on water.
- **Honeybadger**: Another trusty tool for error trackin', Honeybadger also keeps an eye on uptime and alertin' ye when things go awry.
- **Axiom**: For log aggregation and analysis, Axiom be the tool to gather all yer logs into one treasure chest for easy searchin' and visualizin'.
- **Datadog**: A full fleet of monitorin' services, includin' metrics, traces, and logs. Datadog helps ye keep tabs on every corner of yer app.
- **Vector**: A log processor that sails smoothly, collectin', processin', and transportin' yer logs to where they need to be.

Settin' up these tools be a task for a seasoned quartermaster. Configure 'em properly to ensure they be reportin' accurate and useful information.

Logging at the Application and System Level

Loggin' be the ship's log of yer application. At the application level, log errors, warnings, and key events to understand the state of yer app. System-level logging captures events at the OS level, such as resource usage and network activity. Together, they paint a full picture of yer ship's health.

```
# Rails example for logging
Rails.logger.info "Shiver me timbers! This be a log message."
```

Scaling with Docker and Cloud: Unfurl the Sails!

Strategies for Scaling Your Application Using Docker and Cloud Services

Scalin' yer application be like addin' more sails to catch the wind. Docker and cloud services be the wind in those sails, helpin' ye handle more traffic without capsizin'. Here be some strategies:

- **Container Orchestration**: Tools like Kubernetes and Docker Swarm manage yer fleet of containers, deployin' and scalin' 'em as needed.
- **Load Balancin'**: Distribute incoming requests across multiple containers or instances to ensure smooth sailin'.
- **Microservices**: Break yer monolithic application into smaller services that can be scaled independently.

Here be an example of a `docker-compose.yml` file that scales yer Rails app with PostgreSQL and Redis:

```
services:
  db:
    image: postgres:16
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: pirate_app_production
      POSTGRES_USER: pirate_app
      POSTGRES_PASSWORD: ${PIRATE_APP_DATABASE_PASSWORD}
    ports:
      - "5432:5432"

  redis:
    image: redis:7
    ports:
      - "6379:6379"

  web:
    build: .
    command: bundle exec rails server -b 0.0.0.0
    volumes:
      - .:/rails
    ports:
      - "3000:3000"
    environment:
      RAILS_ENV: production
      SECRET_KEY_BASE: ${SECRET_KEY_BASE}
      DATABASE_URL:
postgres://pirate_app:${PIRATE_APP_DATABASE_PASSWORD}@db:5432/pira
te_app_production
      REDIS_URL: redis://redis:6379/1
    depends_on:
      - db
      - redis

volumes:
  postgres_data:
```

Auto-Scaling and Load Balancin’ Considerations

Auto-scaling keeps yer ship ready for any storm, addin' or removin' resources based on demand. Cloud providers like AWS, Azure, and Google Cloud offer auto-scaling services. Configure yer auto-scaling policies to match yer needs, considerin' metrics like CPU usage, memory, and request count.

Load balancers be the helmsmen, directin' traffic to ensure no single container be overwhelmed. Configure health checks to ensure only healthy containers receive traffic.

Configurin' Instance Sizes, CPU Specs, Storage, etc.

Choose the right instance sizes for yer needs. Consider CPU, memory, and storage requirements:

- **CPU and Memory**: Choose instances with enough CPU and memory to handle yer application's load. Use tools like AWS EC2, Azure VM, or Google Compute Engine.
- **Storage**: Ensure ye have enough storage for yer data. Consider using cloud storage solutions like AWS S3, Azure Blob Storage, or Google Cloud Storage.
- **Networking**: Configure virtual private clouds (VPCs) and subnets for secure and efficient communication between yer instances.

With these techniques, ye'll be ready to scale yer application like a true captain of the digital seas, conquerin' new lands and servin' more users without fear! Arrr!

Chapter 10: Security Best Practices

Avast, Me Hearties! Securing Your Docker Containers

Common Security Vulnerabilities and How to Mitigate Them

In the treacherous waters ye must guard yer Docker containers against scallywags and marauders. Here be the most common vulnerabilities and how ye can fortify yer defenses:

- **Outdated Images**: Keep yer base images up to date. Use trusted sources and scan for vulnerabilities with tools like **Anchore** or **Trivy**.

```
docker scan my_image
```

- **Unnecessary Privileges**: Run containers with the least privilege necessary. Avoid root user; create a dedicated user instead.

```
FROM registry.docker.com/library/ruby:3.3.0-slim
RUN useradd -m appuser
USER appuser
```

- **Unexposed Ports**: Only expose the ports ye need. Unexposed ports be like open portholes for enemies.

```
ports:
  - "3000:3000"
```

- **Secrets in Images**: Never hardcode secrets in yer images. Use environment variables or secrets management tools.

Managing Secrets and Sensitive Data

Best Practices for Handling Sensitive Information in Docker and CI/CD Pipelines

In the digital age of 2024, handling secrets be akin to guarding a treasure map. Here be the best practices to ensure yer secrets stay secret:

- **Environment Variables**: Store sensitive data in environment variables. Use `.env` files and Docker secrets for better security.

```
docker run -e SECRET_KEY_BASE=${SECRET_KEY_BASE} my_app
```

- **Secrets Management Tools**: Use tools like **HashiCorp Vault**, **AWS Secrets Manager**, or **Azure Key Vault** to manage secrets securely.

```
secrets:
  secret_key_base:
    file: ./secrets/secret_key_base
```

- **CI/CD Pipelines**: Ensure that yer CI/CD pipelines are secure. Use built-in secrets management in GitHub Actions, GitLab CI, or Jenkins.

```
jobs:
  deploy:
    secrets:
      SECRET_KEY_BASE: ${{ secrets.SECRET_KEY_BASE }}
```

Context-Specific Security Considerations

Security be not one-size-fits-all. Consider the context in which ye operate:

- **Public vs. Private Repositories**: For public repositories, never store sensitive data in code. For private repositories, still follow best practices to avoid leaks.
- **Compliance Requirements**: Ensure ye meet industry-specific compliance, such as GDPR for personal data or PCI DSS for payment information.
- **Network Security**: Configure yer Docker networks to limit exposure. Use firewalls and VPNs for additional layers of security.

```
networks:
  default:
    external:
      name: my_private_network
```

- **Monitoring and Auditing**: Regularly monitor and audit yer containers and systems. Use tools like **Falco** for runtime security monitoring.

With these best practices, yer Docker containers and sensitive data will be guarded like the finest treasures in the pirate's hold, safe from the hands of ne'er-do-wells and rival captains. Arrr!

Chapter 11: Troubleshooting and Debugging

Avast, Me Hearties! Common Issues and Solutions

Troubleshooting Common Problems in Dockerized Rails Applications

Even the finest ships encounter rough seas. Here be some common issues ye might face with Dockerized Rails applications, along with solutions to keep yer vessel on course:

- **Container Fails to Start**: Check yer logs first and foremost. Use `docker logs <container_id>` to see what be ailin' yer container.

```
docker logs pirate_app
```

- **Database Connection Errors**: Ensure that yer environment variables be correctly set. Verify that the database service be up and accessible.

```
environment:
  DATABASE_URL:
postgres://pirate_app:${PIRATE_APP_DATABASE_PASSWORD}@db:5432
/pirate_app_production
```

- **Missing Gems or Node Modules**: Double-check yer `Dockerfile` and ensure that dependencies be properly installed.

```
COPY Gemfile Gemfile.lock ./
RUN bundle install
COPY package.json package-lock.json ./
RUN npm install
```

- **Slow Performance**: Examine resource usage. Ensure containers have enough CPU and memory. Use tools like **Docker stats** to monitor performance.

```
docker stats
```

Debugging Techniques

Tools and Methods for Effective Debugging in a Docker Environment

Debugging be a critical skill for any pirate captain. Here be tools and techniques to help ye navigate troubled waters:

- **Interactive Shell**: Enter yer running container with an interactive shell to inspect and debug.

```
docker exec -it pirate_app /bin/bash
```

- **Rails Console**: Use the Rails console within the container to test and debug application code.

```
docker exec -it pirate_app bundle exec rails console
```

- **Binding.pry**: Use the `pry` gem for interactive debugging. Insert `binding.pry` in yer code and restart the container.

```
gem 'pry'
# In your code
binding.pry
```

- **Logs and Monitoring**: Continuously monitor logs using tools like **Logspout** or integrated logging services in AWS, Azure, or GCP.

```
docker logs -f pirate_app
```

- **Debugger Tools**: Leverage debuggers like **Byebug** or **Pry-Byebug** for more advanced debugging capabilities.

```
gem 'byebug'
# In your code
debugger
```

- **Health Checks**: Implement health checks in yer Docker Compose file to automatically restart unhealthy containers.

```
services:
  web:
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000"]
      interval: 1m30s
      timeout: 10s
      retries: 3
```

- **Network Debugging**: Use tools like **Wireshark** or **tcpdump** to capture and analyze network traffic within yer Docker network.

```
docker run --rm -it --network container:pirate_app
nicolaka/netshoot
```

With these techniques and tools, ye'll be well-equipped to troubleshoot and debug yer Dockerized Rails applications, ensuring smooth sailin' on the high seas of development. Arrr!

Conclusion

Recap and Future Directions

Congratulations, mateys! Ye've sailed through the vast and turbulent seas of Docker and Ruby on Rails, learnin' the ins and outs of modern application deployment and development. Here be a summary of the key takeaways from this adventure:

1. **Setting Sail with Ruby on Rails**: Ye set up a Rails application, configured it with PostgreSQL and Tailwind CSS, and learned the project structure.
2. **Docker – Yer Ship for the Journey**: We explored Docker, wrote Dockerfiles, and ran our Rails app in containers, understandin' the importance of containerization.
3. **Docker Compose – Coordinatin' Yer Fleet**: Ye configured Docker Compose to manage multiple services, makin' development and deployment more efficient.
4. **Guardin' Yer Treasure – Private Dependencies**: Ye integrated private Ruby gems and npm packages into yer Rails app, ensuring secure and modular code.
5. **Secure Authentication with SSH and HTTPS**: Ye set up SSH keys and GitHub Personal Access Tokens (PATs) for secure access to private repositories.
6. **Automatin' Yer Deployment – CI/CD**: Ye established a CI/CD pipeline with GitHub Actions, deployin' yer Rails application to DigitalOcean using `doctl`.
7. **Advanced Docker Techniques**: Ye learned about multi-stage builds, Docker volumes, and networks, ensuring yer containers are efficient and secure.
8. **Monitoring and Scaling Yer Application**: Ye set up monitoring tools and strategies for scaling yer application using Docker and cloud services.
9. **Security Best Practices**: Ye fortified yer Docker containers against vulnerabilities and managed secrets securely in yer CI/CD pipelines.
10. **Troubleshootin' and Debuggin'**: Ye tackled common issues in Dockerized Rails applications and mastered debugging techniques to keep yer app shipshape.

Next Steps for Continuous Learning

The journey doesn't end here, mateys! The seas of technology are always changin', and it's important to keep learnin' and adaptin'. Here be some resources to help ye stay updated and further yer knowledge:

1. **Official Documentation**:
    - Ruby on Rails Guides
    - Docker Documentation
    - GitHub Actions Documentation
2. **Online Courses and Tutorials**:
    - Udemy and Coursera offer courses on Docker, Rails, and DevOps.
    - Codecademy and freeCodeCamp have interactive tutorials.
3. **Community and Forums**:
    - Engage with the Rails, Docker, and DevOps communities on Stack Overflow, Reddit, and GitHub Discussions.
4. **Books and Publications**:
    - "Docker Deep Dive" by Nigel Poulton
    - "Ruby on Rails Tutorial" by Michael Hartl
    - "The DevOps Handbook" by Gene Kim, Jez Humble, Patrick Debois, and John Willis
5. **Conferences and Meetups**:
    - Attend conferences like DockerCon, RailsConf, and DevOpsDays to network with experts and learn about the latest trends.
6. **Practice and Projects**:
    - Keep buildin' projects, contributin' to open-source, and experimentin' with new tools and techniques to refine yer skills.

Final Words

With this knowledge, ye’re equipped to conquer the digital seas, buildin’ robust and scalable applications with Docker and Ruby on Rails. Keep explorin’, innovatin’, and sailin’ towards new horizons. May yer code be clean, yer deployments swift, and yer adventures grand!

Fair winds and following seas, mateys! Arrr!

Appendices

Appendix A: Glossary of Terms

- **Docker**: An open platform for developin’, shippin’, and runnin’ applications using containerization.
- **Rails**: A web application framework written in Ruby, designed for productivity and simplicity.
- **CI/CD**: Continuous Integration and Continuous Deployment, practices for automatin’ the build, test, and deployment processes.
- **SSH**: Secure Shell, a protocol for secure remote login and command execution.
- **PAT**: Personal Access Token, a token used for authentication with GitHub and other services.
- **Volume**: A persistent storage mechanism in Docker, used to store data outside the container lifecycle.

Appendix B: Reference Configurations

Production Dockerfile

```
# syntax=docker/dockerfile:1

ARG RUBY_VERSION=3.3.0
FROM registry.docker.com/library/ruby:$RUBY_VERSION-slim AS base

WORKDIR /rails

ENV RAILS_ENV="production" \
    BUNDLE_DEPLOYMENT="1" \
    BUNDLE_PATH="/usr/local/bundle" \
    BUNDLE_WITHOUT="development"

FROM base AS build

RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y build-essential git
libpq-dev libvips pkg-config curl && \
    curl -fsSL https://deb.nodesource.com/setup_20.x | bash - && \
    apt-get install --no-install-recommends -y nodejs

COPY Gemfile Gemfile.lock ./

RUN --mount=type=bind,target=. \
    --mount=type=secret,id=GITHUB_TOKEN \
    GITHUB_TOKEN=$(cat /run/secrets/GITHUB_TOKEN) && \
    git config --global
url."https://${GITHUB_TOKEN}@github.com/".insteadOf
"https://github.com/" && \
```

```
    bundle install && \
    rm -rf ~/.bundle/ "${BUNDLE_PATH}"/ruby/*/cache
"${BUNDLE_PATH}"/ruby/*/bundler/gems/*/.git && \
    bundle exec bootsnap precompile --gemfile

COPY package.json package-lock.json ./
RUN npm install

COPY . .

RUN npx vite build

RUN bundle exec bootsnap precompile app/ lib/

RUN SECRET_KEY_BASE_DUMMY=1 ./bin/rails assets:precompile

FROM base

RUN apt-get update -qq && \
    apt-get install --no-install-recommends -y curl libvips
postgresql-client && \
    rm -rf /var/lib/apt/lists /var/cache/apt/archives

COPY --from=build /usr/local/bundle /usr/local/bundle
COPY --from=build /rails /rails

RUN useradd rails --create-home --shell /bin/bash && \
    chown -R rails:rails db log storage tmp
USER rails:rails

ENTRYPOINT ["/rails/bin/docker-entrypoint"]
```

```
EXPOSE 3000

CMD ["./bin/rails", "server"]
```

Production docker-compose.yml

```
services:
  db:
    image: postgres:16
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: pirate_app_production
      POSTGRES_USER: pirate_app
      POSTGRES_PASSWORD: ${PIRATE_APP_DATABASE_PASSWORD}
    ports:
      - "5432:5432"

  redis:
    image: redis:7
    ports:
      - "6379:6379"

  web:
    build: .
    command: bundle exec rails server -b 0.0.0.0
    volumes:
      - .:/rails
    ports:
      - "3000:3000"
    environment:
      RAILS_ENV: production
      SECRET_KEY_BASE: ${SECRET_KEY_BASE}
      DATABASE_URL:
postgres://pirate_app:${PIRATE_APP_DATABASE_PASSWORD}@db:5432/pira
te_app_production
      REDIS_URL: redis://redis:6379/1
    depends_on:
      - db
      - redis

volumes:
  postgres_data:
```

Appendix C: Additional Resources

- **doctl**: DigitalOcean CLI tool - Install Guide
- **Rails Guides**: Comprehensive documentation for Ruby on Rails - Rails Guides
- **Docker Documentation**: Official Docker documentation - Docker Docs
- **GitHub Actions Documentation**: Learn how to set up CI/CD pipelines with GitHub Actions - GitHub Actions Docs

Misc

I haven't thought of the following or where to put them yet.

- sidekiq and redis
- setting up a local development environment (extend the initial part to cover postgres and redis etc)
- error handling and troubleshooting for each section and steps
- ruby version, ruby versioning differences and things to watch out for think of
- system dependencies
- application dependencies
- rails configuration
- database creation
- database initialization
- how to set up and configure the test suite
- how to run the test suite
- services (job queues, cache servers, search engines, etc.)
- deployment instructions
- pointing a domain at the application with cloudflare for secure connection
- vite-rails using deprecated CJS here

Load Testing

- Stress (Linux)

  A simple command-line tool to impose load on the system's CPU, memory, and disk I/O.

Application: Useful for stress testing the server infrastructure to observe system behavior under high load.

- Loadster (Service)

  A cloud-based load testing service that simulates user traffic from multiple locations.
  Application: Effective for testing the performance and scalability of web applications under real-world usage patterns.

- k6 (Grafana Labs)

  A modern load testing tool using JavaScript to write test scripts, integrated with Grafana for monitoring.
  Application: Ideal for continuous performance testing and monitoring, with the ability to integrate into CI/CD pipelines for automated testing.

Upgrading the Landing Page for “Linux for Pirates! 2 - Ruby on Whales” Book Promotion

I had to change the application JavaScript to import the stylesheet.

```
// Configure your import map in config/importmap.rb. Read more:
https://github.com/rails/importmap-rails
import "../stylesheets/application.css";
// To see this message, add the following to the `<head>` section
in your
// views/layouts/application.html.erb
//
//    <%= vite_client_tag %>
//    <%= vite_javascript_tag 'application' %>
console.log("Vite ⚡ Rails");

// If using a TypeScript entrypoint file:
//     <%= vite_typescript_tag 'application' %>
//
// If you want to use .jsx or .tsx, add the extension:
//     <%= vite_javascript_tag 'application.jsx' %>

console.log(
  "Visit the guide for more information: ",
  "https://vite-ruby.netlify.app/guide/rails"
);

// Example: Load Rails libraries in Vite.
//
// import * as Turbo from '@hotwired/turbo'
// Turbo.start()
//
// import ActiveStorage from '@rails/activestorage'
// ActiveStorage.start()
//
// // Import all channels.
// const channels = import.meta.globEager('./**/*_channel.js')

// Example: Import a stylesheet in app/frontend/index.css
// import '~/index.css'
```

I commented out everything in `app/javascript/application.js`.

```
// Configure your import map in config/importmap.rb. Read more:
https://github.com/rails/importmap-rails
// import "@hotwired/turbo-rails"
// import "controllers"
// import './stylesheets/application.css'
```

Add PostCSS configuration:

```
// postcss.config.js
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
};
```

Add Vite configuration with PostCSS plugins:

```
// vite.config.ts
import { defineConfig } from "vite";
import RubyPlugin from "vite-plugin-ruby";
import tailwindcss from "tailwindcss";
import autoprefixer from "autoprefixer";

export default defineConfig({
  plugins: [RubyPlugin()],
  css: {
    postcss: {
      plugins: [tailwindcss, autoprefixer],
    },
  },
  assetsInclude: ["**/*.png", "**/*.jpg", "**/*.jpeg", "**/*.gif",
"**/*.svg"],
});
```

In development, I set the following configurations:

```
# config/environments/development.rb
config.assets.quiet = false
config.assets.debug = true
```

The following stylesheet was missing and added:

```
/* pirate_app/app/javascript/stylesheets/application.css */
@import "tailwindcss/base";
@import "tailwindcss/components";
@import "tailwindcss/utilities";
```

Step 1: Organise Your Project Structure

Ensure your project structure under `app/javascript` looks like this:

```
app/javascript
├── images
│   ├── logo.png
│   ├── favicon.png
│   └── social-banner.png
├── entrypoints
│   ├── application.js
│   ├── application.css
│   └── main.js
├── styles
│   └── main.css
└── index.html
```

Step 2: Add Images

Place your images (e.g., logo, favicon, social banner) in the `app/javascript/images` directory.

Step 3: Update Vite Configuration

Ensure your `vite.config.ts` is set up correctly:

```
// vite.config.ts
import { defineConfig } from "vite";
import RubyPlugin from "vite-plugin-ruby";

export default defineConfig({
  plugins: [RubyPlugin()],
  assetsInclude: ["**/*.png", "**/*.jpg", "**/*.jpeg", "**/*.gif",
"**/*.svg"],
});
```

Step 4: Update Rails Layout

Update your application layout to include Vite tags and Open Graph meta tags.

```
<!DOCTYPE html>
<html>
<head>
  <title>PirateApp</title>
  <%= csrf_meta_tags %>
  <%= csp_meta_tag %>

  <%= stylesheet_link_tag "application", "data-turbo-track":
"reload" %>
  <%= javascript_include_tag "application", "data-turbo-track":
"reload", defer: true %>
  <%= vite_client_tag %>
  <%= vite_javascript_tag 'application' %>

  <!-- Open Graph meta tags -->
  <meta property="og:title" content="Linux for Pirates! 2 - Ruby
on Whales" />
  <meta property="og:description" content="Embark on a Ruby
adventure with Linux for Pirates! 2 - Ruby on Whales. Learn Ruby
and conquer the seas!" />
  <meta property="og:image" content="<%= vite_asset_url
'images/pirate-adventure.png' %>" />
  <meta property="og:url" content="<%= request.original_url %>" />
  <meta name="twitter:card" content="summary_large_image" />
</head>

<body class="bg-blue-900 text-white">
  <header class="bg-gray-800 p-4">
    <div class="container mx-auto flex justify-between items-
center">
      <h1 class="text-2xl font-bold">PirateApp</h1>
      <nav>
        <ul class="flex space-x-4">
          <li><a href="/" class="hover:text-yellow-500">Home</a>
</li>
        </ul>
      </nav>
    </div>
  </header>
  <%= yield %>
```

```
  <footer class="bg-gray-800 p-4 mt-10">
    <div class="container mx-auto text-center">
      <p>&copy; 2024 PirateApp. All rights reserved.</p>
    </div>
  </footer>
</body>
</html>
```

Step 5: Update Your View

Update your view to include image tags and debug information.

```
<!DOCTYPE html>
<html>
<head>
  <title>PirateApp</title>
</head>
<body>
  <h1>Home</h1>
  <p>Linux for Pirates! 2 - Ruby on Whales</p>
  <p>Embark on a Ruby adventure with Linux for Pirates! 2 - Ruby
on Whales. Learn Ruby and conquer the seas!</p>

  <!-- Print the asset path for debugging -->
  <p>Image Path: <%= vite_asset_path('images/book-cover.jpg') %>
</p>
  <p>Image Path: <%= vite_asset_path('images/pirate-
adventure.png') %></p>

  <!-- Display the images -->
  <%= image_tag vite_asset_path('images/book-cover.jpg') %>
  <%= image_tag vite_asset_path('images/pirate-adventure.png') %>

  <footer>
    <p>Pirate Adventure</p>
    <a href="#">Get Started</a>
    <p>© 2024 PirateApp. All rights reserved.</p>
  </footer>
</body>
</html>
```

Step 6: Testing and Deployment

Run your Rails server and test the page locally to ensure everything is working as expected. When you're ready, deploy your application.

```
rails server
```

This guide provides a comprehensive overview of upgrading your landing page to promote “Linux for Pirates! 2 - Ruby on Whales” using Vite in a Rails application.

> Note: There is still a warning in Tailwind but I can’t figure it out yet.

Updating package.json to Fix Deprecation Warning

The Rails plugin for Vite generates a `vite.config.ts` that has been deprecated. Instead of translating the configuration file, updating the `package.json` can resolve the deprecation warning.

Original package.json

```
{
  "name": "pirate_app",
  "version": "1.0.0",
  "description": "This README would normally document whatever
steps are necessary to get the application up and running.",
  "main": "index.js",
  "directories": {
    "lib": "lib",
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "autoprefixer": "^10.4.19",
    "my-private-npm-package": "git+https://github.com/loftwah-
demo/my-private-npm-package.git",
    "tailwindcss": "^3.4.4"
  },
  "devDependencies": {
    "vite": "^5.3.3",
    "vite-plugin-ruby": "^5.0.0"
  }
}
```

Updated package.json

To resolve the deprecation warning, add `"type": "module"` to your `package.json`.

```
{
  "name": "pirate_app",
  "version": "1.0.0",
  "description": "This README would normally document whatever
steps are necessary to get the application up and running.",
  "main": "index.js",
  "type": "module",
  "directories": {
    "lib": "lib",
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "autoprefixer": "^10.4.19",
    "my-private-npm-package": "git+https://github.com/loftwah-
demo/my-private-npm-package.git",
    "tailwindcss": "^3.4.4"
  },
  "devDependencies": {
    "vite": "^5.3.3",
    "vite-plugin-ruby": "^5.0.0"
  }
}
```

Enabling `"type": "module"` in `package.json` resolves the CJS deprecation warning.

Troubleshooting Errno::EACCES

The error `Errno::EACCES` indicates that there is a permissions issue when trying to access a file or directory. In this case, it is related to the cache directory used by Sprockets. The specific file it is trying to access is

`/home/loftwah/gits/dockerrails/pirate_app/tmp/cache/assets/sprockets/v4.0.0/2K/2Kqqs5CDZKeqQ8Zi3OcRThACYBsE09CB_YbQaG96080.cache`.

To resolve this issue, you need to ensure that the user running the Rails server has the appropriate permissions to read and write to this directory. Here are the steps you can take to fix this:

1. **Check and Update Permissions**:

   Ensure that the `tmp` directory and its subdirectories have the correct permissions. You can use the `chmod` command to set the correct permissions.

   ```
   sudo chmod -R 755 /home/loftwah/gits/dockerrails/pirate_app/tmp
   ```

   This sets the permissions to read, write, and execute for the owner, and read and execute for group and others.

2. **Change Ownership**:

   Ensure that the correct user owns the directory and its contents. Replace `username` with the appropriate user (e.g., your current user or the user running the Rails server).

   ```
   sudo chown -R username:username /home/loftwah/gits/dockerrails/pirate_app/tmp
   ```

3. **Clear the Cache**:

   If the issue persists, you can try clearing the cache to ensure there are no corrupted files causing the issue.

   ```
   bundle exec rails tmp:cache:clear
   ```

After applying these changes, restart your Rails server and check if the issue is resolved.

Summary

This error is caused by a permissions issue with the cache directory used by Sprockets. By updating the permissions and ensuring the correct ownership, you can resolve the issue and allow your Rails server to access the necessary files.