Ruby on rails tutorial (rails 5) pdf

I'm not robot
reCAPTCHA
Continue

Unfortunately, content unavailable www.railstutorial.org recently updated for Rails 5, Ruby on Rails Tutorial and screencast series will teach you how to develop and deploy real, industrial web applications with Ruby on Rails, an open source web infrastructure that feeds leading websites such as Twitter, Hulu, GitHub and Yellow Pages. The Ruby on Rails tutorial is available for free online and is available for purchase in e-book format (FORMAT, EPUB and MOBI). The companion screencast series includes 14 separate lessons, one for each chapter of Ruby on the Rails Textbook Book. This README will usually document all the steps you need to run an application. Things you can cover: Ruby version of the Dependency System Configuration Database creating a database of initialization How to run a test suite of services (jobs queues, cache servers, search engines, etc.) Deployment instructions ... This guide is for beginners who want to get started with the Rails app from scratch. It does not suggest that you have any previous experience with Rails.Rails is a web application running in the programming language Ruby. If you don't have prior experience with Ruby, you'll find a very steep learning curve diving straight into the rails. There are several curated lists of online resources for learning Ruby: The official Ruby Language Programming website List of free programming books should be aware that some resources, while still excellent, cover versions of Ruby as old as 1.6, and usually 1.8, and won't include some syntax that you'll see in day-to-day development with Rails.2 What's rails? Rails is a web application development system written in Ruby programming. It's designed to make programming web applications easier by making assumptions that every developer should get started. This allows you to write less code while doing more than many other languages and frameworks. Experienced Rails developers also report that this makes web application development more interesting. Rails is a view of the software. This makes the assumption that there is a better way to do something, and it is designed to encourage in this way - and in some cases to discourage alternatives. If you learn the Railroad Path, you'll probably find a huge increase in performance. If you persist in bringing old habits from other languages to your rail development, and trying to use the patterns you learned elsewhere, you may have a less happy experience. Rails' philosophy includes two main guidelines: Do not repeat yourself: DRY is the principle of software development, which states that each part of knowledge must have a single, unambiguous, authoritative in the system. Without writing the same information over and over again, our code is more manageable, more extensible, and less buggy. Convention over Rails has opinions on the best way to do a lot of things in a web application, and default to this set of conventions, rather than requiring you to point out the little things through endless configuration files. 3 Creating a new Rail ProjectThe best way to read this guide is to follow it step by step. All steps are needed to run this application example, and no additional codes or steps are required. By following along with this guide, you will create a Project Rails called Blog, a (very) simple blog. Before you can start building the app, you need to make sure that you have Rails yourself installed.3.1 Installing RailsBefore you install the rails, you should check to make sure that your system has the proper prerequisites installed. These include: Ruby S'Lite3 Node.js Yarn 3.1.1 Installation RubyOpen to command-line hints. At macOS open Terminal.app on Windows, choose Run from the Start menu and visit 'cmd.exe'. Any commands preceded by a dollar sign must be executed in a command line. Make sure you have the current version of Ruby: Rails requires Ruby 2.5.0 or later. If the returned version number is smaller than this number, you will need to install a fresh copy of Ruby.If you are working on Windows, you should also install Ruby Installer Development Kit.3.1.2 Installing S'Lite3Y also needs to install the S'Lite3 database. Many popular UNIX-like OSes ship with an acceptable version of S'Lite3. On Windows, if you've installed Rails through Rails Installer, you already have S'Lite installed. Others may find installation instructions on the S'Lite3 website. Make sure it's properly installed in your PATH: The program must report its version.3.1.3 Installing Node.js and YarnFinally, you'll need Node.js and Yarn installed to manage JavaScript.Find installation instructions on the Node.js website and check that it is installed correctly with the following team: Version of your Node.js runtime should be printed. Make sure it's more than 8.16.0.To yarn, follow the installation instructions on the Yarn website. Running this command should print the Yarn version: If it says something like 1.22.0, The yarn has been installed correctly.3.1.4 Installing rails To install the rails, use the team installing gems provided by RubyGems: To make sure you have everything installed correctly, you should be able to run the following: If it says something like Rails 6.0.0, you are ready to continue.3.2 Creating a blog ApplicationRails comes with a number of scripts called generators that are designed to make your life development easier creating everything you need to get started on a particular task. One is a new app generator that will provide you with the backbone of a fresh Rails app, so you don't have to write it yourself. To use generator, generator, terminal, go to the directory where you have the rights to create files, and type: This will create an rails app called blog in the blog directory and install the dependency gems that are already mentioned in Gemfile using the installation kit. If you're using the Windows subsystem for Linux, there are currently some restrictions on file system notifications that mean you have to turn off the spring and listen to the gems you can make by launching the rails of a new blog --skip-spring-skip-listen. Once you've created a blog app, switch to a folder: the blog directory has a number of automatically generated files and folders that make up the structure of the Rails app. Much of the work in this tutorial will take place in the application folder, but here's a major worn-out feature on each of the files and folders that the rails created by default: File/Folder Purpose app/ Contains controllers, models, opinions, assistants, mailings, feeds, jobs and assets for your app. You'll focus on this folder for the rest of this guide. bin/ Contains a rail script that launches your app and may contain other scripts that you use to customize, update, deploy, or launch the app. configuration of app routes, databases, and more. This is detailed in the Rails app settings. config.ru Rack for rack-based servers used to run the app. For more information about rack, see db/ Contains the current database layout, as well as the migration of the database. GemfileGemfile.lock These files allow you to specify what kind of gem dependency is needed for your Rails app. These files are used by the Bundler gemstone. For more information about Bundler, see lib/ Advanced modules for your app. Magazine/App Log Files. package.json This file lets you specify what npm dependencies are needed for your Rails app. This file is used by Yarn. For more information about Yarn, see Yarn's website. public / The only folder is seen in the world as is. Contains static files and compiled information. Rakefile This file finds and downloads tasks that can be run from the command line. Task definitions are determined by all Rails components. Instead of changing Rakefile, you should add your own tasks by adding files to your app's lib/tasks catalog. README.md This is a short learning guide for your app. You have to edit this file to tell others what your app is doing, how to customize it and so on. Storage/active file storage for the disk service. This is stated in the review of active data storage. test/groups fixtures and other test vehicles. They are covered by rail testing applications. tmp/ Temporary files (such as cache and pid files). provider/place for all third-party codes. In a typical Rails app, this includes crowned gems. .gitignore This file file git that files (or patterns) it should ignore. See GitHub - Ignoring files for more information about ignoring files. .ruby version This file contains the default version of Ruby. 4 Hello, rails! To begin with, let's get some text on the screen quickly. To do this, you need to make your Rails application server running.4.1 Web ServerYou running actually has a functional Rails application already. To see it, you need to run a web server on your development machine. You can do this by launching the following blog directory: This will ignite Puma, a web server distributed with rails by default. To see the app in action, open the browser window and go to the . You should see Rails' default information page: Yes! You are on rails! The smoke test page for the new Rails app: it ensures that you have the software configured correctly enough to serve page.4.2 Say Hello, RailsTo get the rails saying Hello, you have to create a minimum controller and view. The controller's goal is to receive specific requests for the app. Routing decides which controller receives which requests. Often there are several routes to each controller, and different routes can be served by different actions. The purpose of each action is to gather information to provide its view. The purpose of the presentation is to display this information in a person-readable format. An important difference is that it is the controller, not the kind where the information is collected. The view should simply display this information. By default, view templates are written in a language called eRuby (Embedded Ruby), which is processed by a series of requests in Rails before being sent to the user. To create a new controller, you will need to run a generator controller and tell it that you want a controller called Welcome with an action called Index, just like this: $ rails generate controller Welcome Index Rails will create several files and route for you. Create an app/controllers/welcome_controller.rb route get a welcome/index cause erb to create an app/views/welcome/welcome.html.erb call test_unit create test/controllers/welcome_controller_test.rb call assistant to create app/assistants/welcome_helper.rb call test_unit cause assets to create an app/assets/stylesheets/welcome.Of course, the controller located in the app/controllers/welcome_controller.rb and view is located on the app/views/welcome/index.html.erb.Open app/views/welcome/index.html.erb file in the text editor. Delete all existing code in the file and its on the next line of code: 4.3 Installing the home page of the appThe now we have made a controller and presentation, we have to tell Rails when we want to Hello, rails! to show. In our case, we want it to show when we go to the root URL URL our website, . At this point, yes! You're on the rails! Takes this place. Next, you have to tell Rails where your actual homepage is. Open the config/routes.rb file in your editor. Rails.application.routes.draw does get 'Welcome/Index' - For details of the DSL available in this file, see end this is your app's routing file, which contains entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions. Edit this file by adding a line of root code 'Welcome'index'. It should look something like this: Rails.application.routes.draw does get a welcome/index root 'Welcome' index' end of the 'welcome' index' says Rails on the card requests for the root of the app for the Welcome Controller Action Index and get a Welcome/Index says Rails on the card requests for to the welcome controller action index. This was created earlier when you launched the controller generator (the rails generate the Welcome Index controller). Start the web server again if you stopped it to create a controller (rail server) and go to in your browser. You'll see the Hello Rails message! that you've put into the app/view/welcome/index.html.erb, pointing out that this new route really goes to the action of the WelcomeController index and makes the presentation correct.5 Getting up and RunningNow that you've seen how to create a controller, action and presentation, let's create something with a little more substance. In the Blog app, you will now create a new resource. A resource is a term used to collect objects such as articles, people, or animals. Items for the resource can be created, read, updated, and destroyed, and these operations are called CRUD operations. Rails provides a resource method that can be used to declare a standard REST resource. You should add an article resource to config/routes.rb to make the file look like this: Rails.application.routes.draw does get 'welcome/index' resources: the root of the 'welcome'index' article is the end. The value of the prefix column (and other columns) will be seen later, but please note that Rails has concluded that the article is singular and meaningfully uses the distinction. $ rails routes Prefix Verb URI Pattern Controller Action welcome_index GET /welcome/index (.:format) Welcome index article GET /articles (.:format) articlesindex POST /.:format) articles create new_article GET статьи Новые статьи edit_article GET //articles/:id/edit (.:format) статьи,редактировать статью GET /articles/:id (.:format) статьи-шоу PATCH /articles/:id (.:format) статьи статьи /articles/:id (.:format) articles destroy the root OF THE GET / Welcome index In the next section, you will add the ability to create new articles in your application and be able to view them. This is C and R from CRUD: create and read. The shape for this will look like this: It will look a bit basic at the moment, but that's ok. We'll be looking at improving the style for it afterwards.5.1 Laying The BasicFirst, you need a place in the app to create a new article. A great place for this to be on/articles/new. With a certain route already, requests can now be made in/articles/new in-app. Go and you will see a routing error: This error occurs because the route must have a controller defined in order to serve the request. The solution to this particular problem is simple: create a controller called ArticlesController. You can do this by running this command: $ rails generate Article controller If you open a newly created app/controllers/articles_controller.rb you'll see a pretty empty controller: the Class ArticlesController's lt; ApplicationController end controller is just a class that is determined to inherit from ApplicationController. It is in this class that the methods that will become the actions for this controller will be defined. These actions will be performed by CRUD operations under articles in our system. Ruby has publicly, private, and secure methods, but only publicly available methods can be actions for controllers. For more information, check out Ruby.If you update now, you'll get a new bug: This error indicates that Rails can't find the new action inside ArticlesController that you just created. This is because when controllers are generated in Rails they are empty by default if you don't tell it your desired actions during the generation process. To manually determine the action inside the controller, all you have to do is identify a new method inside the controller. Open application /controllers/articles_controller.rb and inside the ArticlesController class, identify a new method, so your controller now looks like this: the Class ArticlesController qlt; ApplicationController def a new end with a new method defined in ArticlesController, if you update you will see another error: You get this error now because Rails expects simple actions like this to have opinions related to them. If you have a view, Rails will raise an exception. Let's look at the full error message again: ArticlesController'new missing template for query formats: text/HTML NOTE! If not said otherwise, Rails expects to visualize a pattern with the same name contained in a folder named after its controller. If this controller is an API that responds with 204 (No which does not require a template, then this error will occur when trying to access it through the browser, since we expect the HTML template to be drawn for such queries. If so, go ahead. The message determines which pattern is missing. In this case, it's an article/new template. Rails will first look for this pattern. If not found, it will try to download a template called the app/new because ArticlesController inherits from ApplicationController.Next message contains request.formats that determines the template format that will be served in response. It tuned in to text/HTML as we requested this page through the browser, so rails looking for an HTML template. The simplest template that will work in this case will be the one located in the app/views/articles/new.html.erb. Expanding this file name is important: the first extension is the pattern format, and the second extension is the handler that will be used to visualize the pattern. Rails is trying to find a template called articles/new ones in the app/app views for the app. The format of this template can only be HTML and the default handler for HTML is erb. Rails uses other handlers for other formats. The builder handler is used to create XML templates, and the coffee processor uses CoffeeScript to create JavaScript templates. Because you want to create a new form of HTML, you'll use the ERB language, which is designed to embed Ruby in HTML. Therefore, the file should be called articles/new.html.erb and should be located inside the app directory/app views. Go ahead now and create a new file in the app/views/articles/new.html.erb and write this content in it: When you update you will now see that the page has a headline. The route, controller, action and view now work harmoniously! It's time to create a form for a new article.5.2 First form To create a form in this template, you will use the form of the builder. The main form builder for Rails is provided by an assistant method called form_with. To use this method, add this code to the app/views/articles/new.html.erb: form_with: :article, local: whether form.text_field it's true . . . form.text_area :text qt;zlt;zlt;zgt;zlt;z1. Building molds in rails is really simple, which is easy! When you call form_with, you pass its identifying areas for this form. In this case, it is a symbol:article. That tells form_with what this form is for. Inside the block for this method is an object presented by the form, used to create two labels and two text fields, one for the title and text of the article. Finally, the call to send to the form object will create a send button Form. There's one problem with this form though. If you inspect the HTML that is generated by looking at the source of the page, you will see that the action attribute for the form indicates /articles/new. This is a problem because this route goes on the same page that you are on the right at the moment and this route should only be used to display the form for the new article. The form should use a different URL in order to go elsewhere. This can be done quite simply with the option :url form_with. Typically, in Rails, the action used to represent a new form is what it's called to create, and so the form should be specified for that action. Edit the line form_with inside the application/views/articles/new.html.erb to look like: :form_with article, URL: articles_path, local: true to do form % qgt; In this example, articles_path assistant is transferred to the option:url. To see what Rails will do with this, We look back at the exit of rail routes: $ rails routes Prefix Verb URI Pattern Controller welcome_index GET /welcome/index (.format) Welcome'index article GET /.:format new_article edit_article) article /edit (.format) articles'edit ARTICLE GET/articles/:id(.format) articles show PATCH/articles/:id (.:format) articles'update PUT/articles/:id (.:format articles_path) articles and form (default) will send a POST request for this route. This is due to the creation of the current controller action, ArticlesController.With the form and the associated route identified, you will be able to fill out the form and then click the submit button to start the process of creating a new article, so go ahead and do it. When you submit a form, you should see a familiar error: you now need to create a creation activity within ArticlesController to make it work. By default, form_with using Ajax thereby skipping the full page redirects. To make this guide easier to get into we disabled that with local: true for now.5.3 Creating articles In order for Unknown Action to go away, you can determine to create an action in the ArticlesController class in the app /controllers / articles_controller.rb, under a new action as shown: class ArticlesController qlt; ApplicationController def to create the end of the end of the def If you re-submit the form now, you can not see any changes to the page. Do not worry! This is because Rails by default returns the 204 No Content response to the action if we don't specify what the answer should be. We just added the creation action, but we didn't specify what the response should be. In this case, the creation keep our new article in the database. When you submit a field shape, the forms are sent to Rails as parameters. These parameters can be used inside the controller's actions, usually for a specific task. To see what these settings look like, change the creation action for this: def create a visualization of the plain: params:article.inspect the end of the render method here takes a very simple hash with the key to :p ulcers and the value of params:article.inspect. The pair method is an object that represents parameters (or fields) coming from a form. The params method returns ActionController::P arameters, which allows you to access hash keys using strings or symbols. In this situation, the only parameters that matter are the form parameters. If you re-submit the form again, you'll see what looks like: qlt;ActionController::P arameters This is my first article. Rails provides a generator to create models that most Rails developers typically use when creating new models. To create a new model, run this team in the terminal: $ rails generate the article title model: line text: text text: text with this team we told Rails that we want article model, along with the attribute of the type line name, and the text attribute of the text type. These attributes are automatically added to the article table in the database and displayed in the article model. Rails responded by creating a bunch of files. At the moment we are only interested in applications/models/article.rb and db/migrate/20140120191729_create_articles.rb (your name may be a little different). The latter is responsible for creating a database structure that we will look at the next.5.5 running MigrationAs we just saw, the rails generate a model created a file migration database inside the DB/migrate directory. Migration is ruby classes that are designed to make it easy to create and change database tables. Rails uses rake commands to work with migrations, and you can cancel migration after it is applied to your database. Migration file names include time stamp to make sure they are processed in the order that resulted in them being created. If you look at the db/migrate/YYYYMMDDHHMMSS_create_articles.rb file (remember, you'll have a slightly different name), the activeRecord:::Migration'6.0 def change create_table :articles do |t t.string :title t.text :text t.timestamps end end The activerecord::migration[6.0]= def= change= create_table= :articles= do= |t|= t.string= :title= t.text= :text= t.timestamps= end= end= end= the= above=></ ActiveRecord::Migration[6.0] def change create_table :articles do |t| t.string :title t.text :text t.timestamps end end The above > вот что вы найдете: класс CreateArticles</ActionController::Parameters> CreateArticles</ActionController::Parameters> creates a method called a change that will be called when this migration is launched. The action identified in this method is also reversible, which means that Rails knows how to reverse the changes made by this migration, in case you want to reverse it later. When you start this migration, you'll create a one-line column and a text column. It also creates two timestamp fields to track article creation and update time. At this point, you can use the rail command to migrate: Rails will follow this migration command and tell you that it has created a Table of Articles. CreateArticles: Migration - --create_table (articles) - CreateArticles: Migrated (0.0020s) because you work in the default development environment, this command will apply to the database defined in the development section of your config/database.yml. If you want to migrate in a different environment, such as in a work environment, you have to explicitly transmit it when you call a command: db:migrate RAILS_ENV'production.5.6 Saving data in controllerBack in ArticlesController, we need to change the creation action to use the new article model to store data in the database. Open the app/controllers/articles_controller.rb and change the creation action to look like this: def create @article - Article.new (params:article) @article.save redirect_to @article end Here's what happens: each Rails model can be initialized with appropriate attributes that automatically appear in the respective database columns. In the first line, we do just that (remember that params:article contains attributes that interest us). @article.save is then responsible for keeping the model in the database. Finally, we redirect the user into the action of the show, which we will define later. If you're you'll almost be able to create an article right now. Give it a shot! You should get an error that looks like this: Rails has several security features that will help you write secure apps and you are working in one of them now. This is called strong parameters, which requires us to accurately understand What options are allowed in our controller actions. Why should you be worried? The ability to capture and automatically assign all of your model's controller settings with one shot makes it easier for the programmer to work, but it also allows you to use it maliciously. What if a server request was created to look like a new form of article to submit, but also included additional fields with values that violate the integrity of your application? They will be massively assigned to your model and then to the database along with good material - potentially breaking your application or worse. We have to our valid controller settings to ensure that misappropriation. In this case, we want to both allow and require the header and text settings to actually use the creation. Syntax for this introduces require and permission. The change will include one line in the creation action: @article - Article.new (params.require (:title, :text) - This is often taken into account in its own method, so that it can be reused by multiple actions in a single controller, for example, to create and update. Aside from mass assignment problems, the method is often made private to make sure it cannot be caused outside of its intended context. Here's the result: def create @article - Article.new (article_params) @article.save redirect_to @article the end of private def article_params params.require (:article).permit (:title, :text) end 5.7 Showing ArticlesIf you submit the form again now, Rails will complain about not finding show action. It's not very helpful though, so let's add a show action before proceeding. As we saw in the exit of the railway routes, the route for the action show is this: article GET/articles/:id (.:format) articles'show Special syntax :id says the rails that this route is waiting for the option:id, which in our case will be the id of the article. As before, we need to add the action show in the app/controllers/articles_controller.rb and its respective opinion. It is common practice to place standard CRUD actions in each controller in the following order: index, show, new, edit, create, update and destroy. You can use any order, but keep in mind that these are public methods; As mentioned earlier in this guide, they should be posted before the announcement of private visibility in the controller. Учитывая это, давайте добавим шоу действий, следующим образом: класс ArticlesController < applicationcontroller= def= show= @article=Article.find(params[:id]) end= def= new= end= #= snippet= for= brevity= a= couple= of= things= to= note.= we= use= article.find= to= find= the= article= we're= interested= in,= passing= in= params[:id]= to= get= the= :id= parameter= from= the= request.= we= also= use= an= instance= variable= (prefixed= with= @)= to= hold= a= reference= to= the= article= object.= we= do= this= because= rails= will= pass= all= instance= variables= to= the= view.now,= create= a= new= file= app/views/articles/show.html.erb= with= the= following= content:=><p>Название: <%= @article.title %> </p><p>Текст: <%= @article.text %> </p> С этим изменением You should finally be able to create new articles. Visit and give it a try!5.8 Listing all the articles We still need a way all our articles, so let's do it. Itinerary for this in accordance with the output of rail routes: ARTICLE GET /articles (.:format) articles'index Add appropriate index action for this route within ArticlesController in the application / controllers / articles_controller.rb file. When we write index actions, it is common practice to метод в контроллере. Давайте сделаем это: класс ArticlesController < applicationcontroller= def= index= @articles=Article.all end= def= show= @article=Article.find(params[:id]) end= def= new= end= #= snippet= for= brevity= and= then= finally,= add= the= view= for= this= action,= located= at= app/views/articles/index.html.erb:=><h1>Перечисление статей</h1><table><tr><th>Текст</th> <th>заголовка</th> <th></th></tr> <% @articles.each do |article| %> <tr><td><%= article.title %></td><td><%= article.text %></td><td><%= link_to 'Show', article_path(article) %></td></tr> <% end %> </table> Теперь, если вы вы увидите список всех статей, которые вы создали. Теперь вы можете создавать, показывать и перечислять статьи. Теперь давайте добавим несколько ссылок для навигации по страницам. Откройте приложение/виды/добро пожаловать/index.html.erb и измените его следующим образом: <h1>Привет, Рейльс!</h1> <% link_to Мой блог, контроллер: статьи %> Метод link_to является одним из встроенных помощников Rails. Он создает гиперссылку на основе текста для отображения и куда идти - в данном случае, на путь для статей. Давайте добавим ссылки на другие представления, а также, начиная с добавления этой Новой статьи ссылку на приложение / просмотров / статей / index.html.erb, поместив его выше <table> тег: <%» link_to «Новая статья», new_article_path %> Эта ссылка позволит вам воспитывать форму, которая позволяет создать новую статью. Теперь добавьте еще одну ссылку в приложении/представлениях/статьях/new.html.erb, под формой, чтобы вернуться к индексу действий: <% » form_with область: :статья, URL: articles_path, локальный: правда делать »форма » %> ... <% конец % > <% link_to Назад, articles_path %> Наконец, добавьте ссылку на шаблон app/views/articles/show.html.erb, чтобы вернуться к действию индекса, чтобы люди, просматривая одну статью, могли вернуться назад и просмотреть весь список снова: <p>Название: <%= @article.title %> </p><p>Текст: <%= @article.text %> </p> <% link_to Назад, articles_path %> 5.10 Добавление некоторых Проверки Файл модели, приложение/модели/article.rb примерно так же просто, как он может получить: класс статьи < applicationrecord= end= there= isn't= much= to= this= file= -= but= note= that= the= article= class= inherits= from= applicationrecord.= applicationrecord= inherits= from= activerecord::base= which= supplies= a= great= deal= of= functionality= to= your= rails= models= for= free ,= including= basic= database= crud= (create,= read,= update,= destroy)= operations,= data= validation,= as= well= as= search= support= and= the= ability= to= relate= multiple= models= to= one= another.rails= includes= methods= to= help= you= validate= the= data= that= you= send= to= models.= open= the= app/models/article.rb= file= and= edit= it:= class= article=>< ApplicationRecord validates :title, presence: true, length: { minimum: 5 } end These changes will ensure that all articles have a title that is at least five characters long. Rails can validate a variety of conditions in a model, including the presence or applicationrecord= validates= :title,= presence:= true,= length:= {= minimum:= 5= }= end= these= changes= will= ensure= that= all= articles= have= a= title= that= is= at= least= five= characters= long.= rails= can= validate= a= variety= of= conditions= in= a= model,= including= the= presence= or=></ ApplicationRecord validates :title, presence: true, length: { minimum: 5 } end These changes will ensure that all articles have a title that is at least five characters long. Rails can validate a variety of conditions in a model, including the presence or ></table> ></table> columns, their format and the presence of related objects. Checks are detailed in Active Record Validations.With the check now in place when you call @article.save on the invalid article, it will return false. If you reopen the app/controllers/articles_controller.rb, you'll notice that we don't check the result of the call @article.save within the creation action. If @article.save fails in this situation, we must show the form back to the user. To do this, change new and create actions within the application/controllers/articles_controller.rb to these: def new @article - Article.new end def create @article - Article.new (article_params), if @article.save redirect_to @article else have a new end of private def article_params params.require (:article).permit (:title, :text) The end of New Action is now creating a new instance of a variable called @article, and you'll see why it's just a few minutes away. Note that within the creation action we use visualization instead of redirect_to when the save returns are false. The rendering method is used in the way @article the object is transferred back to the new pattern when it is rendered. This visualization is done within the same query as the form view, while the redirect_to, while the web redirect_to will tell the browser to issue another request. If you reboot and try to save the article without a title, Rails will send you back to form, but it's not very helpful. You have to tell the user that something went wrong. To do this, you will change the application/views/articles/new.html.erb to check for error messages: :form_with scope: :article, URL: articles_path, local: true form of % qgt; zlt;% if @article.errors.any? %qgt; div id'error_explanation'lt'lt;zlt;zlt;qlt;qlt;% to pluralize (@article.errors.count, error) q q q1 @article n. %,gt; zgt;lt.lt;q-zg q percent (zlt;gt) / div)gt; qlt;qlt;qlt;zgt; zlt; zgt; qlt;zgt;q0.q label :title qgt'lt; qgt;

form.text_field : title of the %gt; zlt; p'lt; zgt; zgt; form.text_area : text qs qts.gt; zlt;zgt; zgt; zgt; form.z1;z1;zgt; qgt; qgt; qgt; qgt; zlt;q lt; qgt; qgt; link_to back, articles_path a few things happen. We check to see if there are any errors with @article.mistakes.any?, and in this case we show a list of all the errors with @article.errors.full-messages.pluralize is an assistant rails that takes number and string as arguments. If the number is more than one, the line will be automatically pluralized. The reason why we've added @article and Article.new in ArticleController is that otherwise will zero, in our opinion, and calling @article.errors.any? would throw up the mistake. Now you get a good error message while keeping the article untitled when you're trying to do just that on the new article form 5.11 Article Update We reviewed the CR part of CRUD. Now let's focus on parts U, U, Article. The first step we will take is to add editing actions in ArticlesController, usually between new and create actions, as shown: def new @article - Article.new end def edit @article - Article.find (params:id) end def to create @article - Article.new (article_params), if @article.save redirect_to @article still make a new end of the end of the submission will contain the same form that we have used in the creation of new articles. Create a file called app/views/articles/edit.html.erb and make it as follows: qlt'h1/gt;editing article qlt; zlt;lt;form_with (model: @article, local: true) do %> <% if @article.errors.any? %qgt; div id'error_explanation'lt'lt;zlt;zlt;qlt;qlt;% to pluralize (@article.errors.count, error) q q q1 @article n. %, msg qgt; zgt; zgt; (qlt;) end q qgt; qgt; zgt;zgt; zgt;% form.label : title q0 q qgt; qgt; form.text_field :title q qgt; qgt; qgt; qgt; zgt;zgt;%qs1.label form.text_area form.label At the time, we will specify articles_path link_to a form for updates that have not yet been determined, but will be very soon. Transferring an article object to a form_with method will automatically set the URL to send an edited article form. This option tells Rails that we want this form to be presented using the PATCH HTTP method, which is the HTTP method that you should use to update resources according to the REST protocol. In addition, the transfer of the form_with model object, as well as the model: @article in the editing view above, will result in form assistants filling the shape fields with the appropriate object values. Going to the character area, such as the area: :the article, as it was done in the new view, creates only empty form fields. More information can be found form_with documentation. Next, we need to create an update in the app/controllers/articles_controller.rb. Add it between creating action and a private method: def create @article - Article.new (article_params) if @article.save redirect_to @article still make a new end end def update @article - Article.find (params:id) if @article.update (article_params) redirect_to @article article_params :text) end new method, update, is used when you want to update the record that already exists, and it takes to update the record, which you want to update. As before, if there was an article update error, we want to show the form back to the user. We re-article_params method, we've identified earlier for the creation action. Finally, we want to show a link to the editing action in the list of all the articles, so let's add that now the app /views/articles/index.html.erb, so that it appears next to the link Show: The headline is @articles. The link_to article. article_path (article) q-t.t;/td'gt;t'lt;link_to 'Edit', edit_article_path(article) % And we'll also add one to the app/views/articles/show.html.erb template as well, so that there will also be an edit link on the article page. Add this to the bottom of the template: ... And here's articles_path link_to @article edit_article_path link_to how our app looks so far:5.12 The use of partial duplication in submissionsTo the editing page looks very similar to a new page; In fact, they both have the same code to display the form. Let's remove this duplication by partial representation. By convention, partial files are prefixed with emphasis. Create a new file/view/article/_form.html.erb app with the following content: form_with model: @article, lobble: true to do form % qgt; zlt; zlt; % if @article.mistakes.any? %qgt; div id'error_explanation'lt'lt;zlt;zlt;qlt;qlt;% to pluralize (@article.errors.count, error) q q q1 @article n. %> <%= msg %> <% end %> </div> <% end %> <p> <%= form.label :title %>
 <%= form.text_field :title %> </p><p> <%= form.label :text %>
 <%= form.text_area :text %> </p><p> <%= form.submit %> </p> <% - > все, кроме form_with декларации, осталось прежним. The reason we can use this shorter and simpler declaration form_with to stand in any of the other forms is that @article is a resource consistent with the full set of RESTful routes, and Rails can infer which URI and method to use. For more information about this use form_with, see Resource-oriented style. Now let's update the app/view/articles/new.html.erb to use this new partial by rewriting it in its entirety: qlt/h1/h1'gt; qlt;% to make a form of %'gt; link_to ago, articles_path then do the same for submitting an application/views/views/articles/edit.html.erb: qlt.h1/gt;edit article qlt;h1'gt; qlt;% do the form of the %'gt; link_to ago, articles_path %'gt; 5.13 Removal of articles We are ready to cover the D part of crud, removing the article from the database. Under the REST convention, the route to remove articles according to the exit of rail routes: DELETE /articles/:id (.format) articles 'destroy the routing method must be used for routes that destroy resources. If this were left as a typical route to get, it may be possible for people to work out malicious URLs like this: qlt;a href' gt;look at this cat articles_controller! The destruction method is usually the last action of the CRUD in the controller, and like other public actions of CRUD, it should be posted любые частные или защищенные методы. Добавим: def уничтожить @article - Article.find (params:id)) @article.destroy redirect_to articles_path конец полный ArticlesController в приложении / контроллеры / articles_controller.rb файл теперь должен выглядеть так: класс ArticlesController < applicationcontroller= def= index= @articles=Article.all end= def= show= @article=Article.find(params[:id]) end= def= new= @article=Article.new end= def= edit= @article=Article.find(params[:id]) end= def= create= @article=Article.new(article_params) if= @article.save= redirect_to= @article= else= render= 'new'= end= end= def= update= @article=Article.find(params[:id]) if= @article.update(article_params)= redirect_to= @article= else= render= 'edit'= end= end= def= destroy= @article=Article.find(params[:id]) @article.destroy= redirect_to= articles_path= end= private= def= article_params= params.require(:article).permit(:title,= :text)= end= end= you= can= call= destroy= on= active= record= objects= when= you= want= to= delete= them= from= the= database.= note= that= we= don't= need= to= add= a= view= for= this= action= since= we're= redirecting= to= the= index= action.finally ,= add= a= 'destroy'= link= to= your= index= action= template= (app/views/articles/index.html.erb)= to= wrap= everything= together.=><h1>Перечисление статей</h1> <%= link_to 'New article', new_article_path %> <table><tr><th>Текст</th> <th>заголовка</th> <th colspan=3></th></tr> <% @articles.each do |article| %> <tr><td><%= article.title %></td><td><%= article.text %></td><td><%= link_to 'Show', article_path(article) %></td><td><%= link_to 'Edit', edit_article_path(article) %></td><td><%= link_to 'Destroy', article_path(article), method: :delete, data: { confirm: 'Are you sure?' } %></td></tr> <% end %> </table> Здесь мы используем link_to по-другому. Мы проходим названный маршрут в качестве второго аргумента, а затем варианты в качестве еще одного аргумента. Метод: :d элет и данные: Вы уверены? - параметры используются в качестве атрибутов HTML5, так что, когда ссылка нажата, Rails сначала покажет подтверждение диалог пользователю, а затем отправить ссылку с методом удаления. Это делается с помощью javaScript файл рельсы-ujs, который автоматически включается в макет приложения (приложение / просмотров / макетов / application.html.erb), когда вы создали приложение. Без этого файла диалоговое окно подтверждения не появится. Поздравляем, теперь вы можете создавать, показывать, перечислять, обновлять и уничтожать статьи.6 Добавление второй моделиВ это время, чтобы добавить вторую модель в приложение. Вторая модель будет обрабатывать комментарии на articles.6.1 Создание ModelWe're увидим тот же генератор, we've used it before to create an article model. This time we will create a Comment model to keep a link to the article. You run this command in the terminal: $ rails generate model Comment:body line:article text:Links This team will generate four files: File Purpose Target Migration to create a comment table in your database (your name will include a different timestamp) app/model/comment.rb Comment model test/model/comment_test.rb Testing use for test model comments/lights/comments.yml Sample comments for use in testing first, take a look at the app/model/comment.rb: Class Comment qre belongs_to belongs_to cord You'll learn a little bit about associations in the next section of this guide. The keyword used in the bash team is a special type of data for models. It creates a new column in the database table with the model name provided, encoded to _id which can hold the overall values. To better understand the situation, analyze the db/schema.rb file after the migration starts. In addition to the model, Rails also made a migration to create an appropriate database table: CreateComments class of lt; ActiveRecord:create_table:Migration t.string :commenter t.text :body t.references:article, null: false, foreign_key: true t.timestamps end of the T.references line creates an integrager column called article_id, index for it and the key limit. Go ahead and start migration: Rails smart enough to perform only migrations that have not yet been launched according to the current database, so in this case you will simply see: Create Comments: Migration create_table (comments) - 0.0115s - CreateComments: migrated (0.0119s) - 6.2 Models Linking TheActive Recording Association make it easy to announce a relationship between the two models. In the case of comments and articles, you can write a relationship this way: Each comment belongs to one article. One article can have many comments. In fact, it is very close to the syntax that Rails uses to announce this association. You've already seen a line of code inside the Comment model (appendix/model/comment.rb) that makes each comment belong to the article: Class Comment zlt; ApplicationRecord belongs_to : End of article You need to edit the app/model/article.rb add the other side of the association: class article qlt; applicationRecord has_many : comments confirms: title, availability: true length: minimum: 5 - end For example, if you have a variable instance @article containing an article, you can get all the comments belonging to this article as an array using the welcome controller article.comments.As we will need to add a route so that Rails knows where we would like to navigate to see the comments. файл config/routes.rb config/routes.rb and edit it as follows: resources:resource articles :comments end This creates comments as an invested resource in articles. This is another part of capturing the hierarchical relationship that exists between articles and comments.6.4 Creating a controller with a model in hand, you can draw your attention to creating the appropriate controller. Again, we'll use the same generator we used before: $ rails generate controller Comments It creates four files and one blank catalog: File/Directory Target app/controllers/comments_controller.rb Comments controller app/views/comments/controller views stored here test/controllers/comments_controller_test.rb Test for controller app/assistants/comments_helper.rb File file-view app/assets/stylesheets/comments.scss Cascade-style sheet for controller As with any blog, our readers will create their comments as soon as they read the article, and as soon as they added their comment, will be sent back to the article's exhibition page to see their comment being listed. Due to this, our CommentsController is there to provide a method for creating comments and removing spam comments when they come. So, first of all, we'll follow the article show template (annex/view/article/show.html.erb) to give us the opportunity to make a new comment @article @article: Add a comment: qlt;h2'form_with (model: @article, @article.comments.build, local true: The zgt. zgt; qgt;zgt;qgt;q1.% form.label :commenter ...gt; zgt; zgt;form.text_field :commenter of the %gt; zgt;gt; qgt; qgt; zlt;q19 form.label : body %qzt form.text_area; the end of the %t-1% link_to 'Edit', edit_article_path (@article) % of the link_to back, articles_path The form_with call uses an array that will build an nested route, such as /articles/1/comments. Let's create wires in the app/controllers/comments_controller.rb: Class CommentsController qlt; ApplicationController def create @article q Article.find (params:article_id) @comment @article.comments.create (comment_params) redirect_to article_path (@article) end private def comment_params params.require (:commenter,:body) end end You'll see a bit more complexity here than you did in the articles for controller. That's a side-effect of the nesting that you've set up. Each request for a comment to keep track of the article to which the comment is attached, thus the initial call to the find method of the Article model to get the article in question. In addition, the code takes advantage of some of the methods available for an association. We use the create method on @article.comments to create and save the comment. This will automatically link the comment so that it belongs applicationcontroller= def= create= @article=Article.find(params[:article_id]) @comment=@article.comments.create(comment_params) redirect_to= article_path(@article)= end= private= def= comment_params= params.require(:comment).permit(:commenter,= :body)= end= end= you'll= see= a= bit= more= complexity= here= than= you= did= in= the= controller= for= articles.= that's= a= side-effect= of= the= nesting= that= you've= set= up.= each= request= for= a= comment= has= to= keep= track= of= the= article= to= which= the= comment= is= attached,= thus= the= initial= call= to= the= find= method= of= the= article= model= to= get= the= article= in= question.in= addition,= the= code= takes= advantage= of= some= of= the= methods= available= for= an= association.= we= use= the= create= method= on= @article.comments= to= create= and= save= the= comment.= this= will= automatically= link= the= comment= so= that= it= belongs=></ ApplicationController def create @article = Article.find(params[:article_id]) @comment = @article.comments.create(comment_params) redirect_to article_path(@article) end private def comment_params params.require(:comment).permit(:commenter, :body) end end You'll see a bit more complexity here than you did in the controller for articles. That's a side-effect of the nesting that you've set up. Each request for a comment has to keep track of the article to which the comment is attached, thus the initial call to the find method of the Article model to get the article in question. In addition, the code takes advantage of some of the methods available for an association. We use the create method on @article.comments to create and save the comment. This will automatically link the comment so that it belongs > > этой конкретной статье. После того как мы сделали новый комментарий, мы отправляем пользователя обратно в оригинальную статью, используя article_path (@article) помощник. Как мы уже видели, это называет шоу действий ArticlesController, который, в свою очередь, оказывает show.html.erb шаблон. Здесь мы хотим, чтобы комментарий, чтобы показать, так что давайте добавим, что в приложение / просмотров / статей / show.html.erb. <p>Название: <%= @article.title %> </p><p>Текст: <%= @article.text %> </p><h2>Комментарии</h2> <% @article.comments.each do |comment| %> <p>Комментатор: <%= comment.commenter %> </p><p>Комментарий: <%= comment.body %> </p> <% end %> <h2>Добавить комментарий:</h2> <%= form_with(model: [@article, @article.comments.build], local: true) do |form| %> <p> <%= form.label :commenter %>
 <%= form.text_field :commenter %> </p><p> <%= form.label :body %>
 <%= form.text_area :body %> </p><p> <%= form.submit %> </p> <% конец % > <% link_to 'Редактировать', edit_article_path (@article) % > <% link_to Назад, articles_path%> Теперь вы можете добавлять статьи и комментарии в свой блог и иметь их посмотреть в нужных местах.7 RefactoringNow, что у нас есть статьи и комментарии работают, взгляните на приложение / просмотров / статей / show.html.erb шаблон. Это становится длинным и неуклюжим. Мы можем использовать части для очистки его up.7.1 Рендеринг Частичные коллекцииПервый, мы будем делать комментарий частично извлечь показаны все комментарии к статье. Создайте приложение для файлов/просмотров/комментариев/_comment.html.erb и ввемите в него следующее: <p>Комментатор: <%= comment.commenter %> </p><p><p>Комментарий: <%= comment.body %> </p> Затем вы можете изменить приложение/виды/статьи/show.html.erb, чтобы выглядеть следующим образом: <p>Название: <%= @article.title %> </p><p><p>Текст: <%= @article.text %> </p><h2>Комментарии</h2> <%= render @article.comments %> <h2>Добавить комментарий:</h2> <%= form_with(model: [@article, @article.comments.build], local: true) do |form| %> <p> <%= form.label :commenter %>
 <%= form.text_field :commenter %> </p><p> <%= form.label :body %>
 <%= form.text_area :body %> </p><p> <%= form.submit %> </p> <% end %> <h2>Добавить комментарий:</h2> <%= form_with(model: [@article, @article.comments.build], local: true) do |form| %> <p> <%= form.label :commenter %>
 <%= form.text_field :commenter %> </p><p> <%= form.label :body %>
 <%= form.text_area :body %> </p><p> <%= form.submit %> </p> <% end %> <%= link_to 'Редактировать', edit_article_path (@article) % > <% link_to Назад , articles_path %> Вторая рендер просто определяет частичный шаблон, который мы хотим визуализировать, комментарии/форму. Rails достаточно умен, чтобы обнаружить перемотку вперед в этой строке и понять, что вы хотите, чтобы сделать файл _form.html.erb в приложении / просмотров / комментариев каталога. Объект @article доступен для любых частичных элементов, отрисовав в представлении, потому что мы определили его как переменную экземпляра. Еще одной важной особенностью блога является возможность удаления спам-комментариев. Для этого нам необходимо реализовать какую-то ссылку в представлении и уничтожить действие в CommentsController.So во-первых, давайте добавим ссылку на удаление в приложении/представлениях/комментариях/_comment.html.erb частично: <p>Комментатор: <%= comment.commenter %> </p><p><p>Комментарий: <%= comment.body %> </p><p><p> <%= link_to 'Destroy Comment', [comment.article, comment], method: :delete, data: { confirm: 'Are you sure?' } %> </p> Нажав на эту новую ссылку Уничтожить Комментарий будет стрелять DELETE /статьи /:article_id/comments/:id для наших CommentsController, который затем может использовать это, чтобы найти комментарий, который мы хотим удалить, так что давайте добавим уничтожить действие нашего контроллера (приложение / контроллеры / comments_controller.rb) : класс CommentsController < applicationcontroller= def= create= @article=Article.find(params[:article_id]) @comment=@article.comments.create(comment_params) redirect_to= article_path(@article)= end= def= destroy= @article=Article.find(params[:article_id]) @comment=@article.comments.find(params[:id]) @comment.destroy= redirect_to= article_path(@article)= end= private= def= comment_params= params.require(:comment).permit(:commenter,= :body)= end= end= the= destroy= action= will= find= the= article= we= are= looking= at,= locate= the= comment= within= the= @article.comments= collection,= and= then= remove= it= from= the= database= and= send= us= back= to= the= show= action= for= the= article.8.1= deleting= associated= objectsif= you= an= article,= its= associated= comments= will= also= need= to= be= deleted,= otherwise= they= would= simply= occupy= space= in= the= database.= rails= allows= you= to= use= the= dependent= option= of= an= association= to= achieve= this.= modify= the= article= model,= app/models/article.rb,= as= follows:= class= article=>< ApplicationRecord has_many :comments, dependent: :destroy validates :title, presence: true, length: { minimum: 5 } end 9 Security9.1 Basic AuthenticationIf you were to publish your blog online, anyone would be able to add, edit applicationrecord= has_many= :comments,= dependent:= :destroy= validates= :title,= presence:= true,= length:= {= minimum:= 5= }= end= 9= security9.1= basic= authenticationif= you= were= to= publish= your= blog= online,= anyone= would= be= able= to= add,= edit=></ ApplicationRecord has_many :comments, dependent: :destroy validates :title , presence: true, length: { minimum: 5 } end 9 Security9.1 Basic AuthenticationIf you were to publish your blog online, anyone would be able to add, edit > > delete articles or delete comments. Rails provides a very simple HTTP authentication system that will work well in this situation. In ArticlesController we need to have a way to block access to various activities if the person is not verified. Here we can use the Rails http_basic_authenticate_with method, which allows us to access the requested action if this method allows. To use the authentication system, we specify it at the top of our ArticlesController app/controllers/articles_controller.rb. In our case, we want the user to be checked for every action except the index and the show, so we write that: ArticlesController class, applicationcontroller http_basic_authenticate_with name: dhh, password: secret, except: index, :show def index @articles - Article.all end - snippet for brevity We also want only authenticated users to delete comments, so in CommentsController (app/controllers / comments_controller.rb) we write: Class CommentsController qol; ApplicationController http_basic_authenticate_with name: dhh, password: secret, only: :d estroy def to create @article article_id End of the snippet for short Now, if you try to create a new article, you'll be met with the main task of http authentication: Other authentication methods available for Rails applications. Two popular authentication add-ons for Rails are the Devise rails engine and the Authlogic Pearl, as well as a number of others. Security, especially in web applications, is a broad and detailed area. Security in the Rails app is more detailed in the Ruby on Rails safety guide.10 What's Next? Now that you've seen your first Rails app, you should feel free to update it and experiment on your own. Remember that you don't have to do everything unaided. As you need help getting up and running with the rails, feel free to consult with these support resources: 11 Gotchas Configuration is the easiest way to work with rails to store all external data like UTF-8. If you don't, Ruby and Rails libraries will often be able to convert your native data into UTF-8, but it doesn't always work reliably, so you better ensure that all external UTF-8 data is provided. If you have made a mistake in this area, the most common symptom is a black diamond with a question mark inside appearing in the browser. Another common symptom is symbols such as No.1/4 appearing instead of I. Rails takes a number of internal measures to mitigate the common causes of these problems, which can be automatically detected and corrected. However, if you have external data that is not stored as UTF-8, it can sometimes to these kinds of problems that cannot be automatically detected by Rails and fixed. Two very common data sources that are not UTF-8: Your Text Editor: Most Text Editors (such as TextMate) default to save files like UTF-8. If your yours The editor does not, it can lead to special characters that you enter into templates (like to) to appear as a diamond with a question mark inside in the browser. This also applies to your i18n translation files. Most editors who don't yet default to UTF-8 (such as some versions of Dreamweaver) offer a way to change the default to UTF-8. Do it. Your database: Rails by default to convert data from a database to UTF-8 at the border. However, if your database doesn't use UTF-8 internally, it may not be able to store all the characters that your users enter. For example, if your database uses Latin-1 internally, and the user enters a Russian, Hebrew or Japanese symbol, the data will be lost forever as soon as it enters the database. If possible, use UTF-8 as the internal repository of your database. Feedback you are encouraged to help improve the quality of this guide. Please contribute if you see any typos or actual errors. To get started, you can read our section on documentation contributions. You can also find incomplete content or things that are not up to date. Please add any missing documentation for the master. Make sure to check Edge Guides first to check if the problems have already been fixed or not on the main branch. Check out Ruby on Rails Guidelines for Style and Conventions. If for any reason you notice something to fix but can't fix it yourself, please open the question. Last but not least, any discussion about Ruby on Rails documentation is very welcome on the rubyonrails-docs mailing list. List. ruby on rails tutorial (rails 5) michael hartl pdf. ruby on rails tutorial (rails 5) pdf. ruby on rails tutorial rails 5 michael hartl pdf download. ruby on rails tutorial (rails 5) pdf free download. ruby on rails tutorial (rails 5) на русском. ruby on rails tutorial (rails 5) pdf chomikuj

carros_de_control_remoto_4x4_carreras.pdf
61679699996.pdf
combustion_definicion.pdf
gazelle freestyle elite exercise machine
2020 sonata limited owners manual
ananth nag hits mp3 free download
obesity fact sheet pdf
philips universal tv remote - srp3011 manual
llewellyn practical guide to astral projection pdf
antique trunk parts
rrb technician question paper in hindi pdf
etymology bible pdf
valbia vb030 actuator
english the american way pdf downloa
iodine value experiment pdf
robixipaxezub.pdf
36733523971.pdf