# DOWNLOAD PDF RUBY BEST PRACTICES

## Chapter 1 : Ruby logging best practices and tips - Coralogix - 3rd generation log analyitcs

*Ruby Best Practices (RBP) by Gregory Brown is unlike any previous book on Ruby written yet. This is not a book of commandments, recipes, design patterns, or style guides. Rather this is a book that is designed to help intermediate Ruby programmers learn how to think about writing and analyzing software.*

The language was designed to be object-oriented, intuitive, and easy to learn to optimize for developer time. We are the masters. They are the slaves. We have one variable, major, and we want to check its value and respond. We may be tempted to use an if statement with multiple elsifs. Avoid this temptation; there is a more concise way! Avoid for loops and use the each method with a block While for loops are common and accepted in most other programming languages, in Ruby, we should almost universally avoid them. For loops are not as space efficient as using the each method and passing it as a block because they store the variable that represents each element. See the following example: Instead, we use the more efficient each method with a block. It is much better practice to write the above code as follows: Suzie is feeling excited today. Suzie is feeling nervous today. People can feel many different things or they can just feel one thing. Because we have a variable number of inputs, we use the splat operator, which groups all of the inputs after the defined inputs into an array. The splat operator also has some other fancy use cases. Whenever we see a splat, we can just think of it as a way of wrangling a collection of an unknown number of items into an organized array so we can work with it more easily. Our board is represented as an array of arrays, where each inner array is a row on the board. If we wanted to access a piece, we would get it like so: We are going to be accessing pieces many times, and it would be easier if we could just call it like this instead: It is also easier to see that [] is just a method if we rewrite array[i] as array. Now, we can read what any piece is by just calling board[row, column]. But what if we also want to be able to move a piece by changing a value of board like this: Keep the developer happy! Use double bang to determine if a value exists The bang symbol! Suppose we were trying to determine if a person has a middle name. Nil is not equal to false, but it is a falsey value. Therefore, we use one bang to turn nil into true and then another bang to turn true into false. Use symbols instead of strings in hashes Symbols are a Ruby specific data-type, denoted like so,: Symbols are similar to strings, except they are immutable and are used to name variables. They should be used whenever you are storing the name of something that does not have to be mutated. They are more efficient than strings because they are stored in one spot in memory. However, when we perform the same operation with a string, this is not the case. Thus, we should use symbols for improved space efficiency when we do not need to mutate its value. We are more efficient and happier if we are not bogged down with syntax and mundane specifics. When we are happier, we write better programs! Thank you Matz for keeping us happy! As a UC Berkeley Engineering graduate and early startup employee, she has navigated many complex challenges with her technical know-how and perseverance. While preparing for her next adventure to become a full time software engineer, she writes tutorials to give back to the developer community. Get in touch at hannahsquier gmail.

## Chapter 2 : 10 Ruby on Rails Best Practices â€” SitePoint

*In this tutorial, we'll explore Ruby best practices to ensure our code is in line with Matz's principles of being legible.*

If you are on Vim, using the right set of plugins is a requisite to be productive. There is the popular https: Even Matz uses emacs; search and you shall find. I am hardpressed to remember occasions where I had to use them instead of the Enumerable methods each, map, select, inject, reject and detect. Learn these methods, chew on them, and use it everywhere! The first two did not have a hash, associative array or dictionary - whichever you prefer to call it. Ah, the failed promises of drag and drop programming! So Hash was a revelation and I started using it anywhere and everywhere. Do you want to build a CRUD app to manage customer info? There were even more crimes committed using Hash that I dare not mention here. You would have gone through enough exercises that uses Hash when working through RubyMonk or Ruby Koans. Understand Immutability and how Ruby passes object references around This has slightly got to do with the above point - all the Enumerable methods are immutable, and it is a good introduction to how functional Ruby veer towards immutability. Immutability is more of a good programming practice than a Ruby specific idea - it helps you write clean predictable code, leave aside concurrent programming and race conditions. If you come from a C programming background, building new objects willy-nilly would be a little hard to digest. So much memory put to waste! I remember reading somewhere that programmers who use high level languages leave a higher carbon footprint because their code is inefficient. I leave you to ponder over it. Understand the difference between a shallow clone and a deep clone. BasicObject true All classes are instances of the class Class. Class true Class is a subclass of BasicObject. Class true Okay, I lost it. It is pretty crazy: For now it is safe to understand that BasicObject is usually the root object of all objects in Ruby. And everything in Ruby is an object. This has a very useful side-effect try this in IRB: Whenever you need a general purpose method and wonder whether Ruby comes with it, just try some plausible syntax in IRB. You might be surprised at what you find. Ruby treats even methods as objects: Always think in terms of objects - not classes. Thinking in terms of Classes can subtly make you evolve your design upfront. Let your objects guide you in how your class definition should look. As a rough analogy, when building a home, the blueprint is valuable only as a reference for building the actual home. You imagine what your home should look like and draw a blueprint accordingly, not the other way round. Start with sparse classes, add methods and attributes as your objects demand it. Srushti puts it better: But even if you are a die-hard procedural ninja, trust me, thinking in terms of objects will help you write better programs, tackle complexity and be a more capable programmer. So, what are the things that are specific to Ruby that you need to be aware of? There is a lot more to OO, some less specific to Ruby. As you go deep into the rabbit hole, ponder over these blanket statements: Use that to your advantage. Interpreted programs are almost always slower than native code which includes JIT. But this gives you a great advantage: Learn it, use it, change the world! It is too dangerous to be almost ever used. It is unsafe and unscoped, but there are better things to achieve similar and useful results. Since Ruby is interpreted, there is no limitation on what can be done during runtime. This can be used to great good as we will see in Metaprogramming. Base end and magically, the User class gives you methods like user. Depending on the fields in the database, Rails defines methods for you to use. This uses Metaprogramming where Rails defines the methods at runtime after consulting the table schema. They say that someone who knows metaprogramming well enough, but not enough to know where not to use it, is a danger to himself and society. These are the methods you would want to look up to get a decent overview of metaprogramming in Ruby: Come back and take a look again later. It is an acquired taste, give it time! Closures Blocks, Lambdas et al. Blocks are my favourite. Rather, they let you write beautiful DSLs when coupled with the right dose of metaprogramming.

## Chapter 3 : Ruby Best Practices : Books

*Ruby best practices, tips and methods to write a better Ruby code This is a scope of best practices, tips and methods that will help you write a better Ruby code, doesn't matter whether you're.*

Use unless Instead of! If you find yourself using an if statement with a negative condition i. However, if you need to involve an else to your conditional, never use unless-else. The preferred way is: I highly recommend you take a detailed look at the style guides here Ruby and here Rails. Write Tests If you are familiar with Rails, then you know how much emphasis the Rails community puts on testing. Also, some say it makes sense to get on with the basics of Rails or in some cases general web development first. Plus, it also covers so many of the edge cases that it drives out a much better design of our objects. A good Ruby developer is instinctively good at testing. Tests acts as detailed specifications of a feature or application. Tests acts as a documentation for other devs, which helps them understand your intent in an implementation. Tests helps in catching and fixing bugs beforehand. Tests gives you confidence when refactoring code or making performance enhancements that nothing is broken as a result. Consider you have these two classes: You can only inherit other classes from it. Use Modules Modules, on the other hand, are a flexible way to share behavior across classes. The reasons one would use modules composition over inheritance are beyond the scope of this post. Use a module, like this: You find yourself doing something like this: Define the status column to be integer, ideally not null null: Now, define enums in your model like this: Not only it gives you these predicate methods with names, it also gives you the methods to switch between defined statuses. These methods will as switch the status matching the method. What an elegant tool to have in your arsenal. Fat Models, Skinny Controllers and Concerns Another best practice is to keep non-response related logic out of the controllers. For example, someone might have their controller like: There are lots of other cases where you have to be smart to find the right balance and know what should go where. You have to figure out where would it fit the best. Controllers should only make simple queries to the model. Complex queries should be moved out to models and broken out in reusable scopes. Controllers should mostly contain request handling and response related logic. Any code that is not request and response related and is directly related to a model should be moved out to that model. Use modules if you have to extract out common functionality from otherwise unrelated functionality. Imagine you need to have a mechanism to send SMS or Email notifications to some subscribers when a book is published, or a push notification to their devices. The bigger, the better. It is one of the best practices to internationalize along with development. You can read more about it here. It gives you following out of the box: Here is short example conversion of a non-internationalized HTML code to an internationalized one: This file is responsible for holding English translations. If you need to add translations for more languages you can just add the files matching the locale name with. Letrs refactor the above HTML to use internationalization: As it is explained above, if you need to create the application database on another machine then you should use db: Running all migrations from scratch is discouraged due its tendency to be get flawed over time. If you are adding new migrations, you should simply just let db: If you run db: For example, if you have a Post resource and a Comment resource, and have these model associations set up:

## Chapter 4 : Ruby Best Practices - pdf - Free IT eBooks Download

*Book Description: Ruby Best Practices is for programmers who want to use Ruby the way Rubyists do. Written by the developer of the Ruby project Prawn (racedaydvl.com), this concise book explains how to design beautiful APIs and domain-specific languages, work with functional programming ideas and techniques that can simplify your code and make you more productive, write code.*

Listed here today are ten of the most popular and useful best practices you can use as a Ruby developer. Namely, any non-response-related logic should go in the model, ideally in a nice, testable method. Say you have code like this: For example, what if we wanted to fetch a single published post? Fortunately, Rails provides a better wayâ€”scopes in older versions of Rails they were called named scopes. Put simply, a scope is a set of constraints on database interactions such as a condition, limit, or offset that are chainable and reusable. As a result, I can call MyModel. So, taking our previous example again, we could rewrite it as follows in Rails 3: Even better, as of Rails 3, Rails now supports relationsâ€”essentially, arbitrary scopes that can be used anywhere. This leads to better constructed code and more reusable queriesâ€—you start to think in terms of what a scope or combination of scopes can achieveâ€—in other words, the big pictureâ€—rather than just what your one query is. As a bonus, it makes it much nicer to compose very complex queries such as searches by simply adding the relations and scopes you need. Along the same lines, spend some time looking at open source gems that already solve your problems. As an example, have a look at the following string interpolation: Alongside this, you can also implement the Enumerable module for any of your classes that you want to provide with useful iteration features. All you need to write in your class are the each and methods. These two simple additions give you a whole heap of extra functionality for free: Manage Attribute Access By default, when using mass assignment in Railsâ€”that is, code similar toUser. Using non-database-backed models can help to organize logic which might otherwise become muddy. For example, there are libraries that give you anActiveRecord-like interface for contact form emails. Adding virtual models also makes it easier to adhere to RESTful controller design, as you can represent data other than database entries as resources. When it comes time to interact with these models in your controller code, your code will be that much cleaner, as you can use the exact same approach as with database-backed models. Using virtual attributes, you can use alternative representations of data in forms with relatively little effort. Use Translations As of Rails 2. Essentially, the Rails i18n framework makes it easy to declare the map of an abstract context to a string. When it comes time to rename something in your interface, having a single place to look for the string in question in order to replace it across the whole application is much quicker than scouring your code for every occurrence. Do you follow these practices already? Let me know in the comments. Comments on this article are closed. Have a question about Ruby on Rails? Why not ask it on our forums?

## Chapter 5 : Ruby Best Practices - O'Reilly Media

*Ruby Best Practices is for programmers who want to use Ruby as experienced Rubyists do. Written by the developer of the Ruby project Prawn, this concise book explains how to design beautiful APIs and domain-specific languages with Ruby, as well as how to work with functional programming ideas and techniques that can simplify your code and make.*

With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, tutorials, and more. My code is better because I write tests that document the expected behaviors of my software while verifying that my code is meeting its requirements. Because my tests are automated, I can hand my code off to others and mechanically assert my expectations, which does more for me than a handwritten specification ever could do. However, the important thing to take home from this is that automated testing is really no different than what we did before we discovered it. The only difference is that one-off examples do not adequately account for the problems that can arise during integration with other modules. This problem can become huge, and is one that unit testing frameworks handle quite well. You write tests because you see the long-term benefits, but you usually write your code first. It takes you a while to write your tests, because it seems like the code you wrote is difficult to pin down behavior-wise. In the end, testing becomes a necessary evil. Masterful Rubyists will tell you otherwise, and for good reason. Testing may be hard, but it truly does make your job of writing software easier. I am using the Test:: Unit API here because it is part of standard Ruby, and because it is fundamentally easy to hack on and extend. Many of the existing alternative testing frameworks are built on top of Test:: Unit, and you will almost certainly need to have a working knowledge of it as a Ruby developer. The ideas here should be mostly portable to your framework of choice. And now we can move on. Most people interpret this as the process of writing some failing tests, getting those tests to pass, and then cleaning up the code without causing the tests to fail again. This general assumption is exactly correct, but a common misconception is how much work needs to be done between each phase of this cycle. For example, if we try to solve our whole problem all in one big chunk, add tests to verify that it works, then clean up our code, we end up with implementations that are very difficult to test, and even more challenging to refactor. This is especially true if you manage to keep your iterations nice and tight. The very nature of test-driven development lends itself to breaking your code up into smaller, simpler chunks that are easy to work with. What follows is the process that I went through while developing a simple feature for the Prawn PDF generation library. But first, a small diversion is necessary. TestCase must "be empty" do Unit before, you might be a bit confused by the use of the must method here. This is actually a custom addition largely based on the test method in the activesupport gem. All this code does is automatically generate test methods for you, improving the clarity of our examples a bit. In practice, these strings look very similar to the most basic HTML markup: I started by considering the possibility of passing all the strings rendered in Prawn through style processing, so the initial case I thought of was actually to allow the method to return the string itself when it did not detect any style data. My early example looked something like this: However, working in small steps like this helps keep things simple and also allows you to sanity-check that things are working as expected. Seeing that this was the case, I was able to move forward with another set of examples. The modified test case ended up looking like this: For evidence, we can look at the first bit of code that made this example work: Luckily, a useful aspect of using automated behavior verification is that it is helpful during refactoring. Before I could do that though, I needed to add another example to clarify the intended behavior: What I do know is that tests are downright awesome for describing a problem to your fellow developers. Within minutes of posting my examples to ruby-talk, I had a much better implementation in hand: However, your code is only as correct as the examples you choose, and as it turns out, this code gave me more than I bargained for. The required change was simple, and caused everything to pass again: However, later I realized that it would be better to check to see whether a string had any style tags in it before attempting to parse it. Because the process of rendering the text is handled in two very different ways depending on whether there are inline styles present, I needed to handle only the case when there were tags to be extracted in my parser: As this example was no longer relevant, I simply removed it and was back under the green light. But I still needed a related feature, which was the ability to test whether a string needed to be parsed. I considered making this a private method on Prawn:: Document, but it led to some ugly code: However, once I found out that I needed an additional helper method, I began to rethink the problem. I realized things would look better if I wrapped the code up in a module. I updated my examples to reflect this change, and cleaned them up a bit by adding a setup method, which gets run before each individual test: Document, it made me happy to give them a home of their own, ready to be expanded later as needed. This is what justifies spending time writing tests that are often longer than your implementation. The resulting examples are mostly a helpful side effect; the power of this technique is in what insight you gain through writing them in the first place. Well-Focused Examples A common beginner habit in testing is to create a single example that covers all of the edge cases for a given method. An example of this might be something along these lines: In the former example, your failure report will include only the first assertion that was violated; the code that follows it will not even be executed. In the latter approach, every single example will run, testing all of the cases simultaneously. This means that if a change you make to your code affects three out of the four cases, your tests will report back three out of four cases rather than just the first failed assertion in the example. Although the code shown here is unlikely to have side effects, there is an additional benefit to splitting up examples: This means you can use setup and teardown methods to manage pre- and postprocessing, but the code will run largely independent of your other examples. Because of this, your tests will be more isolated and less likely to run into false positives or strange errors. Testing Exceptions Code is not merely specified by the way it acts under favorable conditions. Unit makes it easy for us to specify both when code should raise a certain error, and when we expect it to run without error. See if you can understand the tests just by reading through them: We can take a quick look at the implementation of LockBox to see what the code that satisfies these tests looks like: Testing this way will help you catch trivial mistakes up front, which is always a good thing. A key feature of automated testing is that it gives you a comprehensive sense of how your software is running as a system, not just on a component-by-component basis. To keep aware of any problems that might occur during refactoring or wiring in new features, it is beneficial to run your entire suite of examples on every change. Here is a sample Rakefile that uses some of the most common conventions: A typical directory layout that works with this sort of command looks like this: These work pretty much the same as they do on the command line, so you can just put one together that suits your file layout. However, in most cases, this problem can be worked around, and actually leads to better tests. Keep your test cases atomic. If you are testing a function with multiple interfaces, write multiple examples. Also, write an example for each edge case you want to test. Use a rake task to automate running your test suite, and run all of your examples on every change to ensure that integration issues are caught as soon as they are introduced. Running tests individually may save time by catching problems early, but before moving from feature to feature, it is crucial to run the whole suite. Advanced Testing Techniques The most basic testing techniques will get you far, but when things get complicated, you need to break out the big guns. What follows are a few tricks to try out when you run into a roadblock. Using Mocks and Stubs In a perfect world, all the resources that we needed would be self-contained in our application, and all interactions would take place in constant time. In our real work, life is nothing like this. Testing these things can be painful. We could read and write from temporary files, clearing out our leftovers after each example runs. For things like web services, we could build a fake service that acts the way we expect our live service to act and run it on a staging server. The question here is not whether it is possible to do this, but whether it is necessary. Sometimes, you really do need to deal with real-world data. This is especially true when you want to tune and optimize performance or test resource-dependent interactions. However, in most cases, our code is mainly interested only in the behavior of the things we interact with, not what they really are. This is where either a mock or a stub could come in handy. There are additional benefits to removing dependencies on external code and resources as well. By removing these extra layers, you are capable of isolating your examples so that they test only the code in question. This purposefully eliminates a lot of interdependencies within your tests and helps make sure that you find and fix problems in the right places, instead of everywhere their influence is felt. What follows is some basic code that asks a user a yes or no question, waits for input, and then returns true or false based on the answer. A basic implementation might look like this: However, how do we test it? However, what if we wanted to build code that relies on the results of the ask method? There are lots of options for this, but one I especially like is the flexmock gem by Jim Weirich. Though it might be overkill to pull in a third-party package just to stub out a method or two, you can see how this interface would be preferable if you needed to write tests that were a little more complicated, or at least more involved.

## Chapter 6 : Driving Code Through Tests - Ruby Best Practices [Book]

*Our best tips and practices are trying to fill the gap. Check out the Toptal resource pages for additional information on Ruby. There is a Ruby hiring guide, Ruby job description, and Ruby interview questions.*

You are free to: Click here to read the full license. Excerpts from the Preface: Some programming languages excel at turning coders into clockwork oranges. By enforcing rigid rules about how software must be structured and implemented, it is possible to prevent a developer from doing anything dangerous. However, this comes at a high cost, stifling the essential creativity and passion that separates the masterful coder from the mediocre. Thankfully, Ruby is about as far from this bleak reality as you can possibly imagine. As a language, Ruby is designed to allow developers to express themselves freely. However, Ruby is highly malleable, and is nothing more than putty in the hands of the developer. With a rigid mindset that tends to overcomplicate things, you will produce complex Ruby code. With a light and unencumbered outlook, you will produce simple and beautiful programs. A dynamic, expressive, and open language does not fit well into strict patterns of proper and improper use. In fact, there is a great degree of commonality in the way that professional Ruby developers approach a wide range of challenges. My goal in this book has been to curate a collection of these techniques and practices while preserving their original context. Much of the code discussed in this book is either directly pulled from or inspired by popular open source Ruby projects, which is an ideal way to keep in touch with the practical world while still studying what it means to write better code. This book is much more about how to go about solving problems in Ruby than it is about the exact solution you should use. Whenever someone asks the question "What is the right way to do this in Ruby? At this point, Ruby stops being scary and starts being beautiful, which is where all the fun begins. Instead, I assume a decent technical grasp of the Ruby language and at least some practical experience in developing soft- ware with it. The most important thing is that you actually care about improving the way you write Ruby code.

## Chapter 7 : Ruby Best Practices - Best2study

*The world's largest collection of lessons for experienced Ruby developers.*

## Chapter 8 : 6 Ruby Best Practices Beginners Should Know | Codementor

*Ruby on Rails can be tricky for developers coming from other platformsâ€"not only do you need to learn a new language, but a whole new set of best practices as well. Darcy runs through the top*

## Chapter 9 : Ruby Best Practices

*This Ruby style guide recommends best practices so that real-world Ruby programmers can write code that can be maintained by other real-world Ruby programmers. A style guide that reflects real-world usage gets used, while a style guide that holds to an ideal that has been rejected by the people it is supposed to help risks not getting used at.*