



Modular Rails

The Complete Guide
to Modular Rails Applications

By Thibault Denizet

Modular Rails

The Complete Guide To Modular Applications

Thibault Denizet

Contents

Acknowledgements

Foreword

Chapter 1 Introduction

- 1.1** Who Am I ?
- 1.2** Who is this book for?
- 1.3** How is this book organized?
- 1.4** How to read this book?
- 1.5** Errors, bugs, problems?
 - 1.5.1** Typos & bugs
 - 1.5.2** About the code in this book
 - 1.5.3** Module & Engine
 - 1.5.4** Requirements

Chapter 2 The Modular Way

2.1 What is modularity

2.2 Modularity in programming

2.3 Advantages and Drawbacks

2.3.1 Advantages

2.3.2 Drawbacks

2.4 When should you build modular web applications?

2.5 There's no need to go All-In

Chapter 3 Modular Rails: The Basics

3.1 Ruby on Rails Magic

3.1.1 Interpreted Language

3.1.2 Bundler

3.1.3 Deface

3.1.4 Rails engines

3.2 Rails Engines

3.2.1 Quick overview of Rails engines

3.2.2 Creating an engine

3.2.3 Mounting an engine

3.2.4 Namespacing

3.2.5 Extending an engine

3.2.6 Engines in the wild!

3.3 Modular Architectures

3.3.1 What is a monolithic application?

3.3.2 What's a modular application?

3.3.3 Understand the difference between Core and Feature engines

3.3.4 Approach 1 – Three Tier Modules

3.3.5 Approach 2 – Hybrid Component Based

3.3.6 Approach 3 – Full Component Based

3.4 Build a modular CRM

3.4.1 The Idea

3.4.2 SamuraiCRM

Chapter 4 Building SamuraiCRM : the core

4.1 What's the core module ?

4.2 Build it

4.2.1 Part 1: Setup the Rails app

4.2.2 Part 2: The first controller

4.2.3 Part 3: Style our app

4.2.4 Part 4: Users & Authentication

4.2.5 Part 5: The admin panel

4.2.6 Part 6: Authorization

4.2.7 Part 7: Testing

4.3 Round Up

4.3.1 What did we learn?

4.3.2 Main points to remember

4.3.3 Next Step

Chapter 5 SamuraiCRM: The Contacts module

5.1 What's the Contacts module

5.2 Build it

5.2.1 Part 1: Generate the Contacts engine

5.2.2 Part 2: Contact Model

5.2.3 Part 3: Contacts Controller

5.2.4 Part 4: Extending the Core views

5.2.5 Part 5: Extending the Core models

5.2.6 Part 6: Extending the Dashboard

5.2.7 Part 7: Extending the Core controllers

5.2.8 Part 8: Extending the abilities

5.2.9 Part 9: Do It Yourself! Extend the admin Panel

5.3 Round Up

5.3.1 What did we learn?

5.3.2 Main points to remember

5.3.3 Next Step

Chapter 6 SamuraiCRM: The Tasks module

6.1 What's the Tasks module

6.2 Build it

6.2.1 Part 1: Generate the Tasks engine

6.2.2 Part 2: Task Model

6.2.3 Part 3: Tasks Controller & Views

6.2.4 Part 4: Adding tasks to the navigation menu

6.2.5 Part 5: Adding a list of tasks to Contacts

6.2.6 Part 6: Extend the dashboard

6.2.7 Part 7: Extending the abilities

6.2.8 Part 8: Do It Yourself! Extend the admin panel

6.3 Round Up

6.3.1 What did we learn?

6.3.2 Main points to remember

6.3.3 Next Step

Chapter 7 From development to production

- 7.1 Packaging your engines**
 - 7.1.1** What's an engine?
 - 7.1.2** Why package your engines as gems
 - 7.1.3** Make your first module gem
 - 7.1.4** Adding the source
- 7.2 Managing your engines**
 - 7.2.1** Development vs Production application
 - 7.2.2** Working with Git
 - 7.2.3** Where should you push your source code
 - 7.2.4** How to use branches
 - 7.2.5** Versioning your modules
 - 7.2.6** Eating your own dog food
- 7.3 Publishing your engines**
 - 7.3.1** Public engines vs Private engines
 - 7.3.2** Public? Rubygems!
 - 7.3.3** Private? Pick one
- 7.4 Build a private home for your engines**
 - 7.4.1** Setting up Geminabox
 - 7.4.2** Pushing gems to your private gem server
 - 7.4.3** Bundle install your gems
 - 7.4.4** Pushing all our modules
- 7.5 Push your application**
- 7.6 The modular workflow**
- 7.7 Round Up**
 - 7.7.1** What did we learn?
 - 7.7.2** Main points to remember
 - 7.7.3** Next Step

Chapter 8 The End

Acknowledgements

I'd like to thank Philippe D'Acquet who indoctrinated me to modular applications. He probably had no idea I would write a book about it!

I also want to thank Jean Pierre Dumas and Mike Darnell for allowing me to learn so much while building my first modular application.

Finally, thank you Pin and Tar, my modularity bros. You guys are awesome!

And of course, thanks Hongyok for your support and your understanding!

Here are other people who helped me a lot!

- Médéric Petit: For checking that everything made at least some sense.
- Nicholas Baker: For fixing my French style.
- Simon Bonnard: For cheering me up when I wanted to give up.

Cheers!

Foreword

As a Ruby on Rails developer, have you ever encountered problems like slow tests, no re-usability, and some messes in codes that are not-so-easy to clean up? If you answer “Yes” to this question, this book is definitely for you. On the contrary, if your answer is “No”, you could also benefit from this book in plenty of ways such as learning a modular approach to Ruby on Rails application development, making your application more configurable, and applying techniques presented in the book to avoid the problems mentioned above altogether in the future.

The writer of this book, Thibault Denizet, is a colleague of mine at AppyHotel. We are the only two Ruby on Rails developers in the company, and responsible for AppyHotel’s backend applications. One day, the business team came and asked us to make the backend modular. Knowing absolutely nothing about modularity at the time, we went into several stressful meetings with the business team, did a lot of research on the Internet, and experimented with suggestions here and there. At last, we have got it, a modular AppyHotel backend. Developing a Ruby on Rails application with a modular approach surely has pros and cons like everything else in this world. It is a decision you have to make whether or not a modular application works for your task in hand.

I have known Thibault for almost 2 years. Keen, confident, and full of energy, he is not the kind of person you find every day. When something really gets his attention, he will keep doing it. The word “addicted” could be used here appropriately in my opinion. These days, Thibault has much interest in LEGO®. The way he builds them is like a Lego geek would do: building with his mind — building without any instruction provided in Lego boxes. Building Lego is not far from building modular applications. You start with the very first brick, then add another, and on and on and on. Do not stop and eventually you will have a giant robot, or a town, or even the whole world if you may.

In this book, Modular Rails, Thibault shows you how to build a Ruby on Rails application in a modular way using a mock-up of a CRM system as an example to guide you step-by-step. He also discusses about the term “modularity” and how Ruby and Rails are suitable for creating modular applications. With his beautifully creative mind, countless trials and errors, and hand-on experience in shaping AppyHotel’s modular applications, the knowledge and effort he puts into his first book here are exceptional. Entry level Ruby on

Rails developers alongside with those who master Ruby on Rails would surely enjoy the book content and find it valuable. I shall certainly be recommending the book to any Ruby on Rails developers but particularly to those who suffer from monolithic application nightmare.

Panitan P.

Chapter 1

Introduction

It's funny. I had to rewrite this introduction after receiving the foreword you just read. Initially, I spoke about LEGO® but I think my friend Pin covered it already, so here's a short introduction before we get started.

So you know how modular applications work now. Brick by brick, module by module, but you don't know yet how you can build a modular application with Ruby on Rails and that's why you are reading this book. Building modular Rails application means that you will build a set of reusable modules, a bit like libraries, some with very specific functionalities. Then you will just assemble them, just like bricks, and build your application.

Building modular applications is hard. You won't be building your web application like everyone else and finding help is not easy. There's not much documentation either, but with this book you will be able to learn a lot. I will also help you if you still have questions after reading it!

1.1 Who Am I ?



I'm Thibault Denizet. It's me in the picture. I'm a young and dynamic Ruby on Rails developer living in Bangkok. I'm actually French but I hate the cold weather. I also dislike cooking. So I came to a country where it's always (too) hot and where I don't have to cook. Thailand is a great country to live in. If you ever come by, let me know so we can grab a beer!

My story with modularity started over two years ago when I was charged with the rewriting of an existing web application into something more configurable and modular. I've been working on a few modular applications since then while still creating regular apps when modularity was overkill.

I wrote this book because I couldn't find any documentation when I created my first modular application. I was studying open source apps and even though reading code is awesome to learn something, you don't always understand what's going on. This is the book I wish I had at that time. It's not a bible, far from it, but it will teach you everything you need to know. I hope you will enjoy it! (If not, please tell me what's wrong so I can improve it.)

1.2 Who is this book for?

I think every Ruby on Rails developer should read this book or at least give it a try. It's not super long but comes with a lot of 'underground' information about engines. If you're about to write a modular application, then you should definitely read it.

The goal is to teach you how to build your own modules and at the end of the road, create complete modular applications.

This book will probably be more useful to web agencies and freelancers, or anyone who ends up building the same thing over and over again.

1.3 How is this book organized?

This book only has 7 chapters but don't let that fool you. They are huge chapters with a lot of sub-chapters!

- **Chapter 1 - Introduction**

You are reading it right now.

- **Chapter 2 – Theory: The Modular Way**

A quick and easy to follow overview of what is modularity.

- **Chapter 3 – Theory: Modular Rails: The Basics**

This chapter is about the features of Ruby on Rails that allows us to create modular applications.

- **Chapter 4 – Practice: Building SamuraiCRM : the Core**

First practice chapter. We'll create a Rails application (SamuraiCRM) and our first module: the Core! This chapter focuses on configuring a modular application and integrating modules.

- **Chapter 5 – Practice: SamuraiCRM: The Contacts module**

In this chapter, we'll create a second module (Contacts) to add a new feature to SamuraiCRM. This chapter will focus on extending models, controllers and views.

- **Chapter 6 – Practice: SamuraiCRM: The Tasks module**

Here we will add another feature to SamuraiCRM: Tasks. We'll see how we can extend our previous modules and how to make two modules work together while keeping them independent.

- **Chapter 7 – Theory & Practice: From development to production**

After building a complete modular application, we will see how to maintain it and push it to production.

1.4 How to read this book?

This book is meant to be read in the prescribed order: from chapter 1 to chapter 7, but you're free, so just read it the way you want. Chapters 4 to 6 contain most of the source code and that's probably where you will learn the most, especially if you code along. However I tried to keep writing code not mandatory by providing explanations and screenshots so you should be able to just read through and start coding later if you want to.

1.5 Errors, bugs, problems?

1.5.1 Typos & bugs

I tried my best to provide the best content possible. Unfortunately, you will probably find typos and bugs. If you do, please report them by sending an email to bo@samurails.com. I will be eternally grateful and will add your name to the thanks page!

1.5.2 About the code in this book

You will find a lot of source code in this book because I think the best way to learn more about coding is by actually seeing code. The code is contained in colored boxes to be easily noticed. In some cases, the code span more than one page. I did my best to keep the same code snippet on one page but sometimes I simply could not.

If you're not sure where some code is supposed to go, just check the first line of the snippet. You will find a path indicating the file. Most of the time, it will start at the top level Ruby on Rails application (SamuraiCRM) but sometimes it was too long to stay on one line. In these cases, you will see the path starting at the engine level. Keep in mind that engines will be stored in **SamuraiCRM/engines/**.

1.5.3 Module & Engine

Throughout this book I will be using the terms **modular** and **engine** a lot. Like a hell of a lot. I use these two terms interchangeably and when I say **module**, I'm talking about the modular module which is an engine and not Ruby modules.

1.5.4 Requirements

To follow the tutorials in this book, you will need some kind of Unix or Linux system. Indeed, I use the command line intensively to work on modular applications. If you're on Windows, you can easily setup a virtual machine with [VirtualBox](#) and [Ubuntu](#).

Chapter 2

The Modular Way

Let's start by defining the concept.

Maybe you don't know what modularity means or what is a modular architecture. Or maybe you have a vague idea of what it represents, but no concrete example in mind. That's okay, I'm here to help.

This first chapter will take you from knowing nothing about modularity to having a clear idea of what it is and how it can serve you.

2.1 What is modularity

I believe that to understand something you need to understand its origin. Module find its source in Latin as ‘modulus’ meaning ‘a small measure’. By extension, modular means ‘composed of interchangeable units’. The antonym of modular is monolithic which originally was a ‘column consisting of a single large block of stone’. We will see both of these terms throughout the book so it is important that you remember them.

As you’ve probably already understood it, something modular is built from a set of smaller, interchangeable pieces. On the other side, we have monolithic things which are only composed of one element.

For example, we could say a computer is a modular object. It is composed of various pieces, such as the CPU, GPU, RAM and so on, that all fit together to deliver us indispensable platforms like Reddit and Facebook. Most components in a computer can be changed, one at a time, without impacting the normal function, except by improving or reducing the computing capacities. When choosing parts for a computer, the main thing to pay attention to is that the parts work together. They need to be able to talk to each other through common interfaces.

Now that you know that modularity is all around us, let’s see how it fits in the programming world.

2.2 Modularity in programming

Let's focus on programming now since that's why you're here.

What's modularity in programming?

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components. — Wikipedia

So modularity is all about isolating the functionalities of your application into independent and interchangeable components.

But what does that mean exactly? I think the best way to understand this is with an example, and what better example than the Ruby on Rails framework itself! Ruby on Rails used to be monolithic but that all changed when Rails 3 was published, after merging with Merb. Indeed, Rails became modular.

Rails is now composed of a set of modules :

- **Active Record**
- **Active Model**
- **Active Support**
- **Action View**
- **Action Pack**
- **Action Mailer**
- And more recently: **Active Job**

You will usually use all of them in a standard Ruby on Rails application but you could also build a Ruby program with just **Active Record**. That would allow you to easily build a Ruby application that can talk with all kinds of databases. As you can guess from the names, each module in the Rails framework encapsulates a specific feature. Active Record contains everything you need to map Ruby classes to a database tables.

Rails modules are obviously highly reusable since they lay the groundwork for any web application you'd like to build. What we will see in this book is a way to build Ruby on

Rails applications on top of these modules by using the same principles: Re-usability and functionality encapsulation.

2.3 Advantages and Drawbacks

As we already said, modularity comes with 2 major advantages, the re-usability of the components and the encapsulation of features into independent chunks of code called modules. There are actually more advantages!

2.3.1 Advantages

The fact that each module is independent (but a module can be built on top of another one) makes the coordination and cooperation of different teams much easier. If two teams are building 2 completely different features on top of a shared **Core**, even without communication, there won't be a lot of conflicts. Why? Simply because they are working on the equivalent of 2 distinct projects.

Let's say now that you're working for a company building complex web applications. These applications will always require users, authentication and authorization. So why not build it one time and re-use it everywhere? We can build a **Core** module which will handle the users' CRUD, the way they can login and what they can do. Once it's done, you can simply reuse it in every future web application produced by your company, simply by adapting the style.

Finally, I'd like to talk about tests. Automated testing is an important part of software creation, be it for web, desktop or mobile. Since features are encapsulated, you can easily write tests that run in isolation. You can also add tests at the top level of your application to ensure that everything is working well together. It's just like the way libraries are tested to be sure that what they provide is working. Then, you test the code where you use those libraries to be sure you are getting the behavior you want.

Just like everything, modularity comes with some drawbacks too.

2.3.2 Drawbacks

One of the main drawbacks of modular applications is that it can get quite time-consuming. Indeed, you are not dealing with one application and one repository. Instead, you will work with a set of repositories, each containing the source code for a specific

functionality. Depending on the technology used, you will have to update the module, test it, push the source code, then link it to the main application, re-test it and then push it. Now, imagine if you have 3, 4 or more modules to update.

That's a longer process than simply making the change and pushing it live but it's totally worth it! Finally, since we have a lot of different mini-projects, we need to keep track of their version. What is currently used in production, in staging, which versions are ready for testing, etc. The best way I've found to deal with that is simply a Google Doc Spreadsheet. It allows you to see what is where and is easy enough to update.

Now you know that modularity comes with some stuff that you will need to deal with if you want to use it in the best way. Before we see more about modular web applications built with Ruby on Rails, I want to tell you about a few case scenarios where you might want to build a modular application and when you should simply avoid it.

2.4 When should you build modular web applications?

A good example of when to create modular applications is the one I talked about above. If you are a web agency, or a freelancer, and you'd like to reuse some code between clients, then go for modular applications. As a web agency, you probably build a lot of web applications, a lot have common features. A way for people to create accounts, login, access some parts of the application and so on. You will also often want an Admin panel to administrate the application. Then you'll probably have some specific code to add for each client, depending on their needs. But still, you just saved a lot of time reusing an existing module. Imagine all the time you can save through 10 or 100 projects!

Another good case scenario for modularity is a startup creating some kind of generic application with a layer of configuration. That might sound abstract so let me give you an example of such an application. Take the Stack Exchange network. If you're looking to build something like that, then modularity is your best friend. You would probably have a set of modules present in every application in your network. Then, you'd have some specific modules added on top of them to customize the application to a specific niche. Maybe for one application, you want a rating system, for another one you want to have commentable questions and answers. All that can be achieved by building modular applications.

You may have heard about Spree Commerce. Spree is a complete open source e-commerce solution built with Ruby on Rails. Spree is completely modular and is composed of 6 modules. Spree was one of my main examples when I first started to create modular applications and I'd like to thank them for the great work they did. If you're looking for an ecommerce solution, take a look at what they offer, it's very nice!

2.5 There's no need to go All-In

With this book, you will learn how to build modular web applications with Ruby on Rails. I will present you fully modular application but you don't have to go to that extent if you don't want. Hybrid applications are also great. The idea is to use a few modules and build a standard Rails application with them. I will talk more about this approach in the next chapter, after we've talked about architecture.

Chapter 3

Modular Rails: The Basics

In this chapter we'll first see what makes Ruby on Rails such a good solution to build modular applications. We will learn about Rails engines, how they work and how we can use them as modules. Then we'll study different modular architectures before picking one and sticking with it for the entire book. Finally, I will talk about the modular web application that we will create in the next chapters.

3.1 Ruby on Rails Magic

In this first part, I want to share with you a few of the things that makes it easy to create modular application with Ruby on Rails.

3.1.1 Interpreted Language

Since Ruby is an interpreted programming language, there are a few things that we can do super easily. For example, re-opening classes and extending them to add more logic.

Before we continue, I want to show you how important this is, so here is a simple example. Let's say we have two modules (not the ruby ones), X and Y. We define a User model in X and we want to extend that model in Y. All we need to do is create a decorator for the **User** model inside Y and use something like **class_eval** to add some methods or associations to it. By doing that inside the Y module, the X module can stay independent and does not care about Y. X can run on its own but will have more features if we add Y. Obviously, since Y is extending X, it will comes with a dependency on X.

This is one of the basic requirements for modularity and this is all possible thanks to Ruby.

3.1.2 Bundler

Bundler is a great dependency management tool.

A quick **bundle install** will download all the dependencies required by your application and your gems dependencies. A good dependency resolver is required to create modular applications since we will create our modules as libraries. We need to be able to easily get them and their dependencies.

In the kind of modular applications I am going to show you, the parent application (containing the modules) is mostly empty. It's just a shell containing a set of engines which contains all the logic. As we will see in a minute, we will be using Rails engines as modules. The good news is that we can package those as gems and use Bundler to handle them for us. That's why Bundler is a great companion for anyone looking to create a modular application.

Once your engines are packaged as gems (we'll see that in the last chapter), getting your modular application together will be as simple as requiring your modules in the **Gemfile**.

```
gem 'module1', '1.0.0'  
gem 'module2', '1.0.0'
```

3.1.3 Deface

Deface is a neat little gem that will allow us to extend Rails views. I don't want to say too much yet, but you will learn very soon how it works and how to use it. This is one of the pillars of modularity.

3.1.4 Rails engines

We will be using Rails engines as the bricks of our application. Let's see what are they and how to use them.

3.2 Rails Engines

With Rails 3, a new feature came out. Rails Engines. They are just like regular Ruby on Rails applications but can't actually live on their own, they need a regular Rails application to run from. Engines and regular Rails applications have actually a lot in common. First of all, they share the same structure.

You will find the following elements in both:

- Models
- Controllers
- Views
- Assets
- Tests

The nice thing about engines is that you can integrate them to any Ruby on Rails application. Once an engine is ready to be deployed, it can be encapsulated as a gem and pushed to RubyGems if you want to share it with the community. You can also keep it private and we will see how later.

3.2.1 Quick overview of Rails engines

Engines are basically miniature Ruby on Rails applications that can provide new functionalities to the parent application including them. The class defining a Ruby on Rails application is **Rails::Application** which inherits a lot of its behavior from **Rails::Engine** which defines an engine. A Ruby on Rails application is just a bit more than a Rails engine, coming with the logic required to run on its own.

3.2.2 Creating an engine

Engines are also closely related to plugins. Plugins are not really used anymore since Rails 3.0 but the command to generate an engine is actually the same used to generate a plugin. We will see how to generate an engine in the next chapter when we start working on our modular application.

3.2.3 Mounting an engine

An engine cannot live on its own, before you can use it you need to integrate it inside a Rails application. There are actually two ways to do this but the one that interests us is this one:

```
Rails.application.routes.draw do
  mount Samurails::Core::Engine, at: '/', as: 'samurails'
end
```

In this example, we mounted the Core engine at the root of a Rails application. If there is nothing inside your parent application, that's the easiest way.

However, if your Rails application has some content you probably want to do the following:

```
Rails.application.routes.draw do
  mount Samurails::Forum::Engine, at: '/forum', as: 'forum'
end
```

Now the **Forum** engine can be accessed by going to **/forum** and the root can be used by the parent application.

3.2.4 Namespacing

Engines can and should be namespaced. That means that two models with the same name, one in an engine and one in the host application, won't clash. When generating an engine, everything is automatically namespaced: tables, models, controllers. If you generate an engine named **Core** and a **User** model, the generated table will be **core_users** and your model will be **Core::User**.

Throughout this book, I recommend going one step further and adding another level of namespacing. If you plan to create a lot of engines and want to avoid any future conflict, you can just namespace all your engines with your project or with your company name.

Instead of having:

```
module ModuleName
  class ModelName < ActiveRecord::Base
  end
end
```

You would have:

```
module GlobalModuleName
  module ModuleName
    Class ModelName < ActiveRecord::Base
      end
    end
  end
end
```

With this system, you can keep your engines organized under a common namespace. You can rest assured that you won't get any problems when adding external engines to your application.

A special thanks to James Adam, Piotr Sarnacki, the Rails Core Team, and a number of other people for their awesome work on the Rails Engines!

3.2.5 Extending an engine

We already talked a bit about extending an engine and we will get into the coding part in the fourth chapter. There are basically three things to extend inside a Ruby on Rails application.

- Models

Can be extended using decorator and **class_eval** or other similar method.

- Views

Can be extended with the **Deface** gem.

- Controllers

Can be extended just like models: decorator and **class_eval**.

There is actually a fourth entity that we will have to extend and we will see how later.

3.2.6 Engines in the wild!

Even if you've just discovered Rails Engines, you've probably been using them for a while. One of the most famous authentication gems is actually an engine.

Of course, I'm talking about Devise! Basically, any gem that will provide you with some

Rails-related component such as migrations, controllers, views or models is not just a gem. It's actually an engine packaged as a gem.

A few examples of famous engines:

- [Forem](#)

Forem is an engine for Rails that aims to be the best little forum system ever. The end goal is to have an engine that can be dropped into an application that provides the basic functionality of forums, topics and posts.

- [Devise](#)

Devise is a Rack Based engine that provides a complete authentication system for any Ruby on Rails application.

- [Spree Ecommerce](#)

Spree is a complete open source e-commerce solution built with Ruby on Rails as set of engines. Spree actually consists of several different gems, each of which are maintained in a single repository and documented in a single set of online documentation.

3.3 Modular Architectures

Creating a modular application is quite different from creating a regular monolithic application. In this section, I want to go over the different architectures that you can use to build a modular application with Ruby on Rails.

3.3.1 What is a monolithic application?

Regular Ruby on Rails applications are monolithic applications. They are self-contained applications that will do what you programmed them for. Note that this is not a bad thing. Ruby on Rails is known for its fast prototyping that will let you create an MVP (Minimum Viable Product) easily.

In software engineering, a monolithic application describes a software application which is designed without modularity. -Wikipedia

So that's exactly what we are trying to avoid by building modular applications. Monolithic have their place, just not in this book!

Note that I don't recommend creating only modular applications. I still create and maintain monolithic applications because I don't always need the modular side. Regular applications are also easier to work with. It's all about picking the right tool for the right job.

3.3.2 What's a modular application?

Now, let's talk about modularity.

There is not just one way to organize your code. Here, I'll go over three ways that I've personally played with. I'm sure there are more and if you have any suggestions, I would be happy to talk about them with you.

As we said earlier, we are going to cut our monolithic application in small modules. But what should we put in these modules? Since we will be using Rails engines, we can put models, controllers, views, migrations, tests and so on. Basically everything that you will find inside a Ruby on Rails application.

So how do we organize all that? Should one module have all the models? Or should one module have a model-view-controller silo?

3.3.3 Understand the difference between Core and Feature engines

In the following section and in the future chapters, I will sometimes talk about **Core** engines and **Feature** engine. When building a modular application, you need at least one **Core** engine. This engine will contains things like the configuration and the basic functionalities. **Feature** engines extend the **Core(s)** and provide specific features.

The best way to understand the difference is by reading the following chapters where we will build a **Core** engine followed by two **Features** engines.

3.3.4 Approach 1 – Three Tier Modules

You should be familiar with the three tier architecture. If not, here's a definition.

Three-tier architecture is a client–server architecture in which the user interface (presentation), functional process logic (“business rules”), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. — Wikipedia

Since the views are so tied to the rest of the application in Ruby on Rails, a good way to follow this pattern is by creating an API and a Javascript frontend application.

The idea is to separate the application into the following modules/engines:

- **Engine 1 - Data Engine**

This engine contains the data layer: models, migrations and models specs.

- **Engine 2 - API**

This engine contains an API allowing external applications to connect to it through RESTful URLs. This engine depends on the Engine 1.

- **Engine 3 - Javascript application**

The Javascript application should be built with the help of a frontend Javascript framework such as Angular.js. The code can be contained inside a Rails engine and included inside the parent application. Another way, which I personally prefer, is to create the Javascript application as a standalone by using tools like Yo, Grunt and Bower.

Here is a diagram representing this architecture. [Figure 3.1](#)

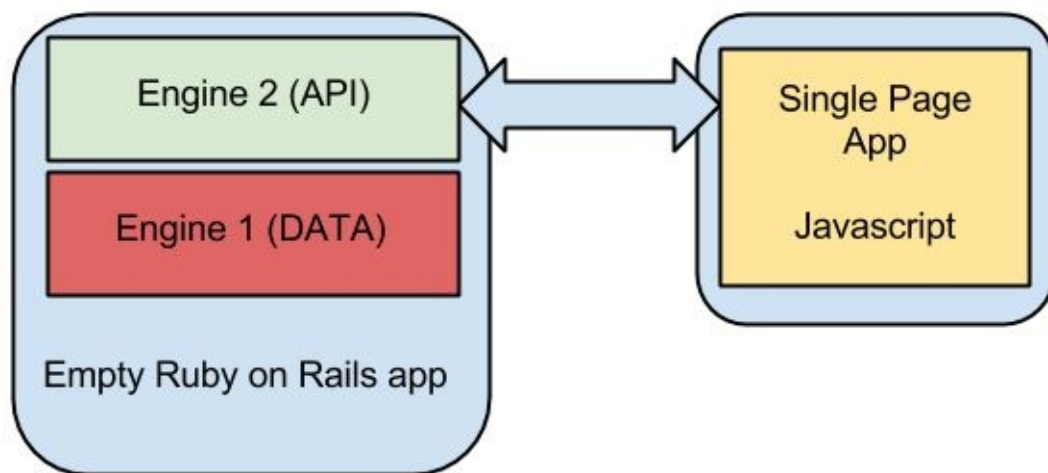


Figure 3.1: Three Tier Modules.

This architecture can be optimized based on your needs. For example, you could put the Engine 2's logic directly inside the parent application. You could also separate the Engine 1 into smaller engines. Keep reading to learn more about other architectures to find the perfect one for your needs!

3.3.5 Approach 2 – Hybrid Component Based

This approach is the first one I tried when I started to work on modular applications. With time, I realized that it's not the best solution. The idea is to have a couple of engines representing the minimal logic of your application. On this set of engine, you can later add new features encapsulated inside new engines.

With this architecture, the 'core' of your application will be composed of at least two engines:

- **Core Engine 1: Models & Migrations engine**
- **Core Engine 2: Controllers & Views engine**

Now, you can start adding more modules which will depend on this ‘core’.

- **Feature 1 engine (models + controllers + views)**
- **Feature 2 engine (models + controllers + views)**
- **Feature 3 engine (models + controllers + views)**

That might look abstract so to give you something more concrete here’s how your **Gemfile** might look like with this architecture:

```
gem 'core_data', '1.0'
gem 'core_view', '1.0'
gem 'feature_1', '1.0'
gem 'feature_2', '1.0'
gem 'feature_3', '1.0'
```

And here is a diagram showing you the complete architecture. [Figure 3.2](#)

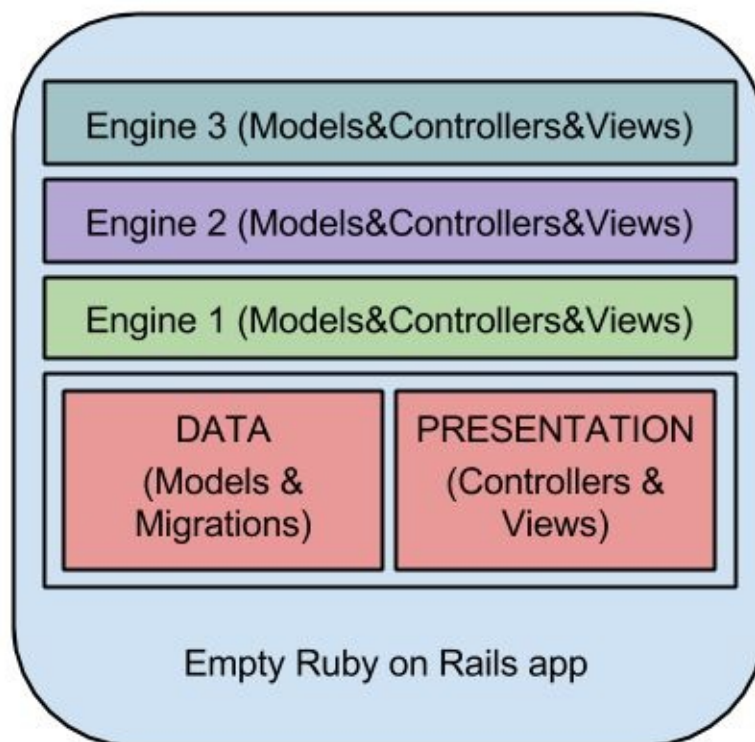


Figure 3.2: Hybrid Component Based.

3.3.6 Approach 3 – Full Component Based

The Full Component Based architecture is a refined version of the previous example. In this one, we remove the ‘core’ engines and the separation of the models, views and controllers into different engines. Instead, each engine encapsulates all the components

required to make it work. That means each engine is a micro Rails application containing models, controllers, views and specs.

- **Core (models + controllers + views)**
- **Feature 1 (models + controllers + views)**
- **Feature 2 (models + controllers + views)**

Here is the diagram. [Figure 3.3](#)

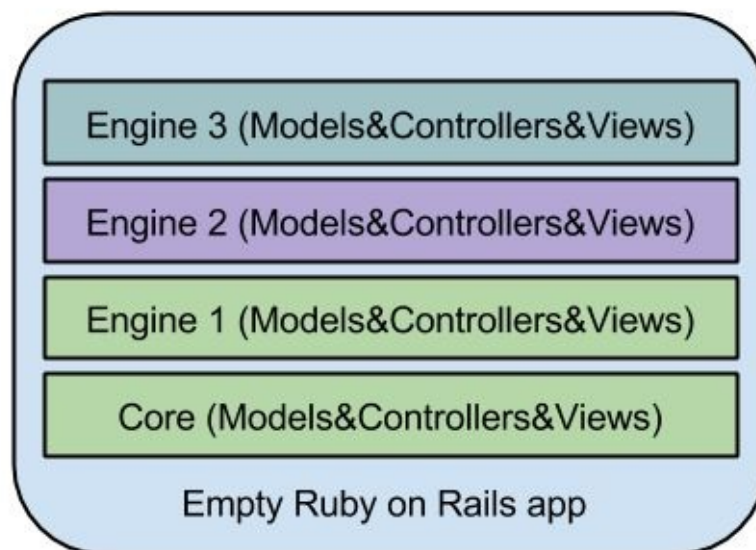


Figure 3.3: Full Component Based.

In this case, the **Core** could be run alone inside the containing application but would offer only limited features. For example, we could simply include the minimum code for users to be able to login and access an empty dashboard. We can then add competently encapsulated features to the core to improve our application.

This is a very good solution to build modular applications with Ruby on Rails which is why we will pick this architecture to build our modular application. By using layers like this, we can easily and completely encapsulate one specific feature inside one engine. Each engine can also be independant from the others, except for their dependency on the **Core**.

3.4 Build a modular CRM

The best way to learn something is by doing it.

As French people like to say:

It is by forging that one becomes a blacksmith. — French people

So we're going to build a little modular application. I want to show you how to create your own modular application by using the various techniques I learned the hard way.

3.4.1 The Idea

We're going to make a CRM, or Customer Relationship Management, and we're going to call it SamuraiCRM because that's an amazing name. CRM are usually used by salespeople in companies to keep track of their contacts, meetings, calls and so on. So what should we put inside SamuraiCRM?

3.4.2 SamuraiCRM

Our CRM will be composed of the following:

- The Parent Application

This will be an almost empty Ruby on Rails application that will contain all the engines. This will be the container for all the engines.

- The **core** Module

This engine will put the foundations of our CRM in place. We will include everything related to users: login, authentication, authorization, administration. We will also define the style for the application and create the dashboards that will be extended by the modules later.

- The **Contacts** Module

A basic CRUD for contacts that will extend the **core** module.

- The **Tasks** module

A basic CRUD for tasks that will extend the **Core** and **Contacts** modules.

Let's code!

Chapter 4

Building SamuraiCRM : the core

That's it, you're about to create your first module. Feeling excited? You should!

4.1 What's the core module ?

Before we dive in the code, let's define what is the **Core** module. This module sets the foundations of our application. The other engines will be built on top of it by extending its very basic functionalities.

The **Core** will include:

- **The dashboard**

The dashboard is the first thing a user sees after logging in. In the **Core**, it will be pretty much a blank screen. But in the next chapters we will add hooks to it to allow other modules to extend it. This way, we will be able to fill the dashboard a little bit in each module.

- **The admin panel**

The admin panel is an important part of any CRM (Customer Relationship Management). We will build a basic admin panel in the **Core** which will allow us to manage **Users**. We will make it extendable in the future so other modules can also integrate themselves in this panel.

- **Authentication**

Since we are lazy developers, we will be using the great Devise gem to handle Authentication. Devise being an engine, we will meet a few problems while using it inside another engine. Do not worry, I will show you how to make everything work together!

- **Authorization**

Authorization will be achieved by using CanCan. In future chapters, we will see how to extend CanCan abilities in other modules.

- **Specs**

The **Core** module will come with a set of tests to ensure that it's working correctly. I will keep the code and tests simple so you can focus your attention on the modular side. We'll be using Rspec and Factory Girl.

- **Modularity**

Modularity principles will be explained while building the application and summed

up at the end of the chapter.

Now, it's time to start building it!

4.2 Build it

The **Build it** section is where we actually play with some code. You will find one in each of the following chapters.

4.2.1 Part 1: Setup the Rails app

Before we actually work on the **core** engine, we need a parent application. Let's generate one and create our first engine!

Note that this book was written with Ruby version 2.2 and Rails version 4.2.

Step 1: Generate the app

Fire up a terminal and navigate to your workspace before running the following:

```
rails new SamuraiCRM --skip-test-unit
```

Step 2: Go inside your new application folder and generate a mountable engine

```
cd SamuraiCRM && rails plugin new core --mountable --skip-test-unit
```

Step 3: Create an 'engines' folder and move the Core inside

```
mkdir engines && mv core engines/
```

All our engines will go inside this folder.

Step 4: Add one more namespace level

By default, everything inside an engine will be namespaced based on the name you gave it. Since we are building a modular application and a set of engines that are supposed to work together, I recommend adding one more namespace. For this application, I will use the namespace **Samurai**. It will be the global namespace containing all the engines.

For now, let's focus on the lib folder located at **SamuraiCRM/engines/core/lib**. In this folder, we'll find the heart of our engine but first, we have to do a bit of reorganization. Rearrange the lib folder until you have the following structure or run the command below.

```
SamuraiCRM/engines/core/  
  lib/  
    samurai_core.rb  
    samurai/  
      core.rb  
      core/  
        engine.rb  
        version.rb
```

Here's a command to achieve this result.

Run it from **SamuraiCRM/engines/core/**.

```
cd lib && mkdir samurai && mv core core.rb samurai/ && touch samurai_core.rb
```

Let's open these files and fill them a bit.

Step 5: samurai_core.rb and core.rb

This file is a very simple file that is going to make the link between our engine and the parent application. How? By requiring the right files!

```
# SamuraiCRM/engines/core/lib/samurai_core.rb  
require "samurai/core"  
require "samurai/core/engine"
```

And this one is where we load our modules for the first time, before the engine gets loaded.

```
# SamuraiCRM/engines/core/lib/samurai/core.rb  
module Samurai  
  module Core  
    end  
end
```

Step 6: Define a version

Since the **Core** module will be released as a gem, we need to give it a proper version and add our top-level namespace **Samurai**.

```
# SamuraiCRM/engines/core/lib/samurai/core/version.rb  
module Samurai  
  module Core  
    VERSION = "0.0.1"  
  end  
end
```



```
end
```

Step 7: The Engine file

Each Ruby on Rails engine comes with a file named **engine.rb**. This is really the heart of the engine. We need to add our new namespace to it. Note that we are also removing **Core** from **isolate_namespace**. We won't namespace our models and controllers with **Core** in the **Core** module. You will understand why in the next chapter, but for now, you should know that it makes extending the **Core** much easier.

```
# SamuraiCRM/engines/core/lib/samurai/core/engine.rb
module Samurai
  module Core
    class Engine < ::Rails::Engine
      isolate_namespace Samurai
    end
  end
end
```

What is **isolate_namespace**? The **isolate_namespace** method is here to mark a clear separation between the engine controllers, models and routes and the parent application's content. Without it, we might have unwanted conflicts or overrides.

Step 8: Gemspec

Next, we need to give some proper details about our engine inside the **gemspec** file. Head over to **SamuraiCRM/engines/core/core.gemspec**. First, you will want to rename the file to **samurai_core.gemspec** to match the namespaces we are now using. Here's the file with the changes.

```
# SamuraiCRM/engines/core/samurai_core.gemspec
$.push File.expand_path("../lib", __FILE__)
require "samurai/core/version"

Gem::Specification.new do |s|
  s.name           = "samurai_core"           # Rename core to samurai_core
  s.version        = Samurai::Core::VERSION  # Add namespace
  s.authors        = ["Thibault Denizet"]     # Your Name
  s.email          = ["bo@samurails.com"]     # Your Email
  s.homepage       = "http://samurails.com"
  s.summary        = "Core features of SamuraiCRM."
  s.description    = "Core features of SamuraiCRM."
  s.license        = "MIT"

  s.files = Dir["{app,config,db,lib}/**/*", "MIT-LICENSE", "Rakefile",
               "README.rdoc"]
  s.test_files = Dir[""]
  s.add_dependency "rails", "~> 4.2.0"
  s.add_development_dependency "sqlite3"
end
```

That's it for the **gemspec** file!

Step 9: bin/rails

To make everything works smoothly, we need to change one small thing inside the **bin/rails** file.

```
# SamuraiCRM/engines/core/bin/rails
# ...
# Changed from 'lib/core/engine'
ENGINE_PATH = File.expand_path('../../lib/samurai/core/engine', __FILE__)
```

Step 10: The Core routes

If you open the routes file for the **Core** engine located at **SamuraiCRM/engines/core/config/routes.rb**, you will see that the **Samurai** namespace is missing. Obviously, we need to add it!

```
# SamuraiCRM/engines/core/config/routes.rb
Samurai::Core::Engine.routes.draw do
end
```

Step 11: Add the module to the parent Gemfile

Go back to the parent application and open the **SamuraiCRM/Gemfile** file. All we have to do now is add our module:

```
# SamuraiCRM/Gemfile
# ...
gem 'samurai_core', path: 'engines/core'
```

Step 12: bundle install

Now, do a quick bundle install from the parent application and everything should work! If not, review the setup and see if you changed everything correctly.

Step 13: Mount the engine

The **Core** module is now integrated inside the parent application. But it's not accessible

yet. For that, we need to mount it inside the parent's routes file. Don't expect something hard, it's as easy as:

```
# SamuraiCRM/config/routes.rb
Rails.application.routes.draw do
  mount Samurai::Core::Engine => "/", as: 'samurai'
end
```

Step 14: Test!

Et voila, we are finally done with the setup! You should be able to start your rails server now but you will end up on the default rails view if you try to access it in a browser. How about adding some content?

4.2.2 Part 2: The first controller

To keep things simple, we won't be writing tests in a TDD style. However, at the end of this chapter, we'll go over the setup of the testing tools and write a few tests so you can see how to test your modular application.

The first controller we will create is the **dashboard_controller**. It's where the users will land once they logged in. On this page, they should be able to have a summary of everything that happened recently. Their recent contacts, their recent tasks, their current opportunities and so on.

For now, we'll create an empty page because we simply don't have anything to show yet. In the next chapters, we will add content to this page with each module.

Step 1: Reorganize the controllers folder

The first step before continuing is to re-arrange the controller folder inside the **Core** engine. Since we will only use **Samurai** to namespace our controllers, we need to rename the **Core** folder.

Your controllers folder should look like the following:

```
engines/core/app/
controllers/
  samurai/
    application_controller.rb
```

Here's a command you can run from the engine folder to get this structure:

```
mv app/controllers/core app/controllers/samurai
```

You also have to change your ApplicationController to look like this:

```
# SamuraiCRM/engines/core/app/controllers/samurai/application_controller.rb
module Samurai
  class ApplicationController < ActionController::Base
  end
end
```

Step 2: Create the DashboardController.

Now we can add the **dashboard_controller** which will have only one action: index. Here's a command to create it. Run it from the engine.

```
touch app/controllers/samurai/dashboard_controller.rb
```

And its content:

```
# SamuraiCRM/engines/core/app/controllers/samurai/dashboard_controller.rb
module Samurai
  class DashboardController < ApplicationController
    def index
    end
  end
end
```

Step 3: Add the corresponding route.

```
# SamuraiCRM/engines/core/config/routes.rb
Samurai::Core::Engine.routes.draw do
  root to: "dashboard#index"
end
```

Now, you can try to access the application. Run **rails s** and head to **localhost:3000**. Or don't, because it's gonna crash anyway! We have to fix the views before we can see anything.

Step 4: Fix the layout and add the index view for the dashboard

Once again, due to the namespace we added, we need to create an extra folder for the views. Change your **views** folder until you have the following or use the command below.

```
SamuraiCRM/engines/core/app
views/
```

```
layouts/  
  samurai/  
    application.html.erb  
samurai/  
  dashboard/  
    index.html.erb
```

The commands: (Run them from the **Core** engine folder)

```
mv app/views/layouts/core app/views/layouts/samurai
```

```
mkdir -p app/views/samurai/dashboard && \  
touch app/views/samurai/dashboard/index.html.erb
```

Now, if you access **localhost:3000**, the application works and returns a beautiful white screen!

4.2.3 Part 3: Style our app

We're going to use bootstrap-sass to style SamuraiCRM. To do that, we need to add a few gems to the engine **gemspec**.

Step 1: Add the gems!

```
# SamuraiCRM/engines/core/samurai_core.gemspec  
# ...  
s.add_dependency "rails", "~> 4.2.0"  
  
s.add_dependency 'sass-rails', "~> 5.0.1"  
s.add_dependency 'bootstrap-sass', "~> 3.3.3"  
s.add_dependency 'autoprefixer-rails', "~> 5.1.5"  
# ...
```

When used inside an engine, gems are not automatically loaded. To load them, we need to require them inside **SamuraiCRM/engines/core/lib/samurai/core.rb**:

```
# SamuraiCRM/engines/core/lib/samurai/core.rb  
require 'sass-rails'  
require 'bootstrap-sass'  
require 'autoprefixer-rails'  
  
module Samurai  
  module Core  
    end  
end
```

Don't forget to do that when you work on your own modules.

Let's run **bundle install** from the parent application to get all the gems we just added. Don't forget to restart your server.

Step 2: Fix the assets folder

Once again, we have to fix and rearrange the assets folder to add one more level. Change your assets folder inside the **Core** engine to look like this or use the commands below.

```
assets/  
  images/  
    samurai/  
  javascripts/  
    samurai/  
      application.js  
  stylesheets/  
    samurai/  
      application.css
```

The commands: (Run them from the **Core** engine folder)

```
mv app/assets/images/core app/assets/images/samurai
```

```
mv app/assets/javascripts/core app/assets/javascripts/samurai
```

```
mv app/assets/stylesheets/core app/assets/stylesheets/samurai
```

Step 3: Rename application.css and load Bootstrap css

Since we're going to use Bootstrap, we need to change the **application.css** file. Simply rename it to **application.css.scss** and change its content to:

```
// SamuraiCRM/engines/core/assets/stylesheets/samurai/application.css.scss  
@import "bootstrap-sprockets";  
@import "bootstrap";  
body {  
  padding-top: 65px; // For the nav bar  
}
```

Now we're almost ready to use Bootstrap!

Step 4: Load Bootstrap javascript

Let's add Bootstrap javascript files too.

```
// SamuraiCRM/engines/core/assets/stylesheets/samurai/application.js  
//= require_tree .  
//= require jquery  
//= require jquery_ujs  
//= require bootstrap-sprockets
```

Time for some HTML!

Step 5: Update the layout file

Because we changed a bit the assets folders, we need to update the layout file by changing the stylesheet and javascript link locations.

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->
<!DOCTYPE html>
<html>
  <head>
    <title>Core</title>
    <%= stylesheet_link_tag "samurai/application", media: "all" %>
    <%= javascript_include_tag "samurai/application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Step 6: Add content to the layout file

Since we added Bootstrap, we're going to leverage it! You can paste the following code inside the **<body>** tag.

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->
<!-- ... -->
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <%= link_to 'SamuraiCRM', samurai.root_path, class: 'navbar-brand' %>
      </div>
    </div>
  </nav>

  <div class='container' role='main'>
    <div class='jumbotron'>
      <%= yield %>
    </div>
  </div>
</body>
<!-- ... -->
```

This is some pretty basic Bootstrap stuff. We added a navbar that will contain the links of our menu and we surrounded the yield call with the container class provided by Bootstrap. The jumbotron is just here to add a bit of style and break with the super white design.

One important point to note however is the use of **samurai.root_path**. Why don't we just put **root_path**? Well it's due to Devise. Without **samurai** in front of every route, the app will crash when we try to access the views from Devise in the future. There is some kind of path conflict when using Devise inside a namespaced engine.

Is it a good practice to prefix all our paths with **samurai** to avoid any problem in the future.

If you check your browser, you should see the following. [Figure 4.1](#)

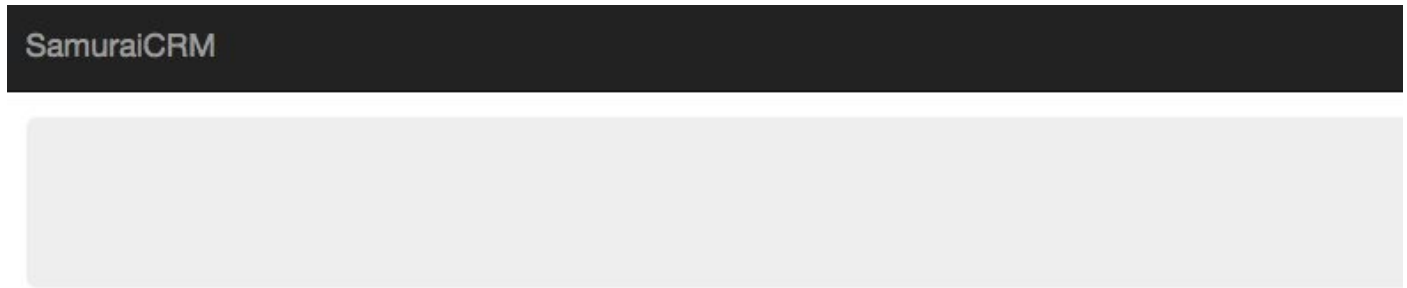


Figure 4.1: So white... So empty...

Step 7: Add a title to the dashboard

Last step before we create our first model, we're going to give a title to the blank dashboard. Just add the following code to the dashboard index view.

```
<!-- SamuraiCRM/engines/core/app/views/samurai/dashboard/index.html.erb -->
<h2>Dashboard</h2>
<hr>
```

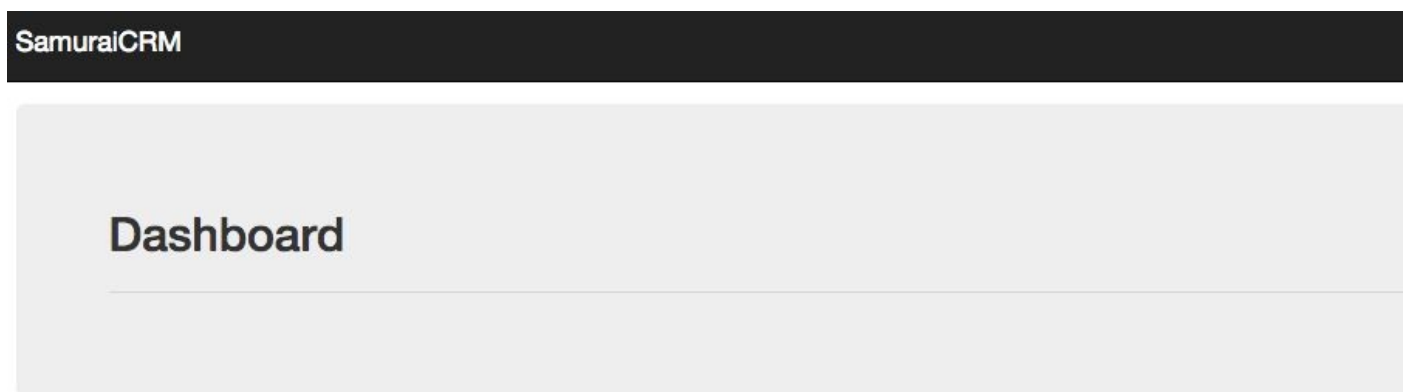


Figure 4.2: Now, it looks pretty good. If you have poor taste!

4.2.4 Part 4: Users & Authentication

Our modular application is still pretty basic. It's time to allow people to create an account and login! We're not going to reinvent the wheel, so let's use Devise, the great

Authentication gem (which is actually built as a Rails engine).

Step 1: Add the Devise gem in the Core module gemspec file.

```
# SamuraiCRM/engines/core/samurai_core.gemspec
# ...
s.add_dependency 'devise', '~> 3.4.1'
```

Require it inside the engine:

```
# SamuraiCRM/engines/core/lib/samurai/core.rb
require 'sass-rails'
require 'bootstrap-sass'
require 'autoprefixer-rails'
require 'devise'

module Samurai
  module Core
    end
  end
end
```

And run **bundle install** from the parent application. Don't forget to restart your webserver too!

Step 2: Generate Devise

Devise comes with a handy generator that we're going to use to create the required files. Run the following command from inside the **Core** engine.

```
rails generate devise:install
```

It should generate a file named **devise.rb** in **core/config/initializers/**. This file holds the configuration for Devise. We have to tweak a few things since we are running Devise from inside an engine (Devise being an engine itself).

Step 3: Configure Devise

And these two lines at the end of **core/config/initializers/devise.rb**.

```
config.router_name = :samurai
config.parent_controller = 'Samurai::ApplicationController'
```

Basically, we're telling Devise that it's going to be running inside an engine and that it should use these values instead of its default ones.

Step 4: Add Flash messages

We're going to add flash messages so Devise can tell a user when something is wrong. To do that and keep it clean, we're going to use a helper method. But first, we need to reorganize the helpers folder in the **Core** engine.

```
core/app/helpers/  
samurai/  
application_helper.rb
```

Here's the command. (Run it from the **Core** engine)

```
mv app/helpers/core app/helpers/samurai
```

We also have to add the **Samurai** namespace to the **application_helper.rb**.

```
# core/app/helpers/samurai/application_helper.rb  
module Samurai  
  module ApplicationHelper  
    end  
  end  
end
```

Since this is a sample app, I'm gonna put my helper methods inside the **ApplicationHelper**. In a real application, you should split your helpers in different files depending on the kind of 'help' they provide.

```
# core/app/helpers/samurai/application_helper.rb  
module Samurai  
  module ApplicationHelper  
    FLASH_CLASSES = {  
      notice: "alert alert-info",  
      success: "alert alert-success",  
      alert: "alert alert-danger",  
      error: "alert alert-danger"  
    }  
    def flash_class(level)  
      FLASH_CLASSES[level]  
    end  
  end  
end
```

Add the following code at the top of the container in the layout file inside the **Core** engine, right before the jumbotron!

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->  
<!-- ... -->  
<div class='container' role='main'>  
  <%= flash.each do |key, value| %>  
    <div class="<%= flash_class(key.to_sym) %>"><%= value %></div>  
  <% end %>  
</div>
```

Step 5: Generate the User model

Now it's time to generate the **User** model with Devise. Devise has a neat generator to do that. We'll have to edit a few things since we are doing non-mainstream stuff but it's gonna be easy.

Run the following command from inside the **Core** engine.

```
rails generate devise User
```

Step 6: Add migrations to parent app paths

Wait, our migration is inside the **Core** engine. How are we going to migrate it from the parent app? We don't want to run the migrations manually from each engine one after the other, that would be annoying. Well, we just have to tell the parent application to look for migrations inside the engines!

To do that, we need to add an initializer to the **Core** engine. Open the engine file located at **core/lib/samurai/core/engine.rb** and add the following inside the class **Engine**.

```
initializer :append_migrations do |app|
  unless app.root.to_s.match(root.to_s)
    config.paths["db/migrate"].expanded.each do |p|
      app.config.paths["db/migrate"] << p
    end
  end
end
```

One more step before we can migrate!

Step 7: Fix Devise Routes

Remember when I said we would have some tweaks to do in order to have Devise working inside another engine? Well, here's one! When we ran the Devise generator, it added a new line to the **routes.rb** file. The line looks like this:

```
devise_for :users, :class_name => "Samurai::User"
```

But it's not gonna work like this. We need to tell Devise that we're running it inside an engine!

```
devise_for :users, class_name: "Samurai::User", module: :devise
```

The **class_name** specify the model we want to use. **module** is here to tell Devise that it's not running inside a regular Rails application.

Step 8: Migrate

To run your first modular migration, go into the parent application folder and run:

```
rake db:migrate
```

The table for the **User** model should be created without any problem! Restart your server before continuing.

Step 9: Copy views from Devise

Devise's default views are nice, but we want to adapt them to our style. Bootstrap style. So we're going to generate the view files and edit them. Run the following command from inside the **Core** engine:

```
rails g devise:views
```

Step 10: Add authenticate_user!

Before we can access these views, we need to actually check if we have a **current_user**. If not, we want the unsigned user to create an account or login. To do that, we just have to add a **before_filter** to the **ApplicationController** located inside the **Core**.

```
# core/app/controllers/samurai/application_controller.rb
module Samurai
  module Core
    class ApplicationController < ActionController::Base
      before_action :authenticate_user!
    end
  end
end
```

Step 11: Update Devise new session view

Now if you try to access the app at **http://localhost:3000**, you should be redirected to the sign in view. But it does not look good at all! [Figure 4.3](#)

You need to sign in or sign up before continuing.

Log in

Email

Password

☐ Remember me

Log in

[Sign up](#)

[Forgot your password?](#)

Figure 4.3

Here's an updated version of the new session file generated by Devise.

```
<!-- SamuraiCRM/engines/core/app/views/devise/sessions/new.html.erb -->
<h2>Sign in</h2><hr>
<%= form_for(resource, as: resource_name, url: session_path(resource_name),
  html: {class: 'form-horizontal'}) do |f| %>
  <div class="form-group">
    <%= f.label :email, class: "col-sm-2 control-label" %>
    <div class="col-sm-6">
      <%= f.email_field :email, autofocus: true ,
        class: "form-control" %>
    </div>
  </div>
  <div class="form-group">
    <%= f.label :password, class: "col-sm-2 control-label" %>
    <div class="col-sm-6">
      <%= f.password_field :password, autocomplete: "off",
        class: "form-control" %>
    </div>
  </div>
  <% if devise_mapping.rememberable? -%>
    <div class="form-group">
      <div class="col-sm-6 col-sm-offset-2">
        <%= f.check_box :remember_me %> <%= f.label :remember_me %>
      </div>
    </div>
  <% end -%>
  <div class="form-group">
    <div class="col-sm-6 col-sm-offset-2">
      <%= f.submit "Sign in", class: 'btn btn-primary' %>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-6 col-sm-offset-2">
      <%= render "devise/shared/links" %>
    </div>
  </div>
```

```
</div>  
</div>  
<% end %>
```

There is nothing interesting enough to point out in this file, being mostly a design change. Let's check the new look. It looks much better now! [Figure 4.4](#)

SamuraiCRM

Sign in

Email

Password

☐ Remember me

Sign in

[Sign up](#)

[Forgot your password?](#)

Figure 4.4

Step 12: Update Devise new registration view

Now let's check the registration. Simply click on Sign Up. [Figure 4.5](#)

Sign up

Email

Password

Password confirmation

Sign up

[Sign in](#)

Figure 4.5

Let's quickly replace this code!

```
<!-- SamuraiCRM/engines/core/app/views/devise/registrations/new.html.erb -->
<h2>Sign up</h2>
<hr>
<%= form_for(resource, as: resource_name, url: registration_path(resource_name),
             html: {class: 'form-horizontal'}) do |f| %>
  <%= devise_error_messages! %>

  <div class="form-group">
    <%= f.label :email, class: "col-sm-2 control-label" %>
    <div class="col-sm-6">
      <%= f.email_field :email, class: "form-control" %>
    </div>
  </div>

  <div class="form-group">
    <%= f.label :password, class: "col-sm-2 control-label" %>
    <div class="col-sm-6">
      <%= f.password_field :password, autocomplete: "off",
                        class: "form-control" %>
    </div>
  </div>

  <div class="form-group">
    <%= f.label :password_confirmation, class: "col-sm-2 control-label" %>
    <div class="col-sm-6">
      <%= f.password_field :password_confirmation, autocomplete: "off",
                        class: "form-control" %>
    </div>
  </div>

  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-6">
      <%= f.submit "Sign up", class: "btn btn-primary" %>
    </div>
  </div>

  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-6">
      <%= render "devise/shared/links" %>
    </div>
  </div>
```

```
</div>
</div>
<% end %>
```

And check how it looks like. [Figure 4.6](#)

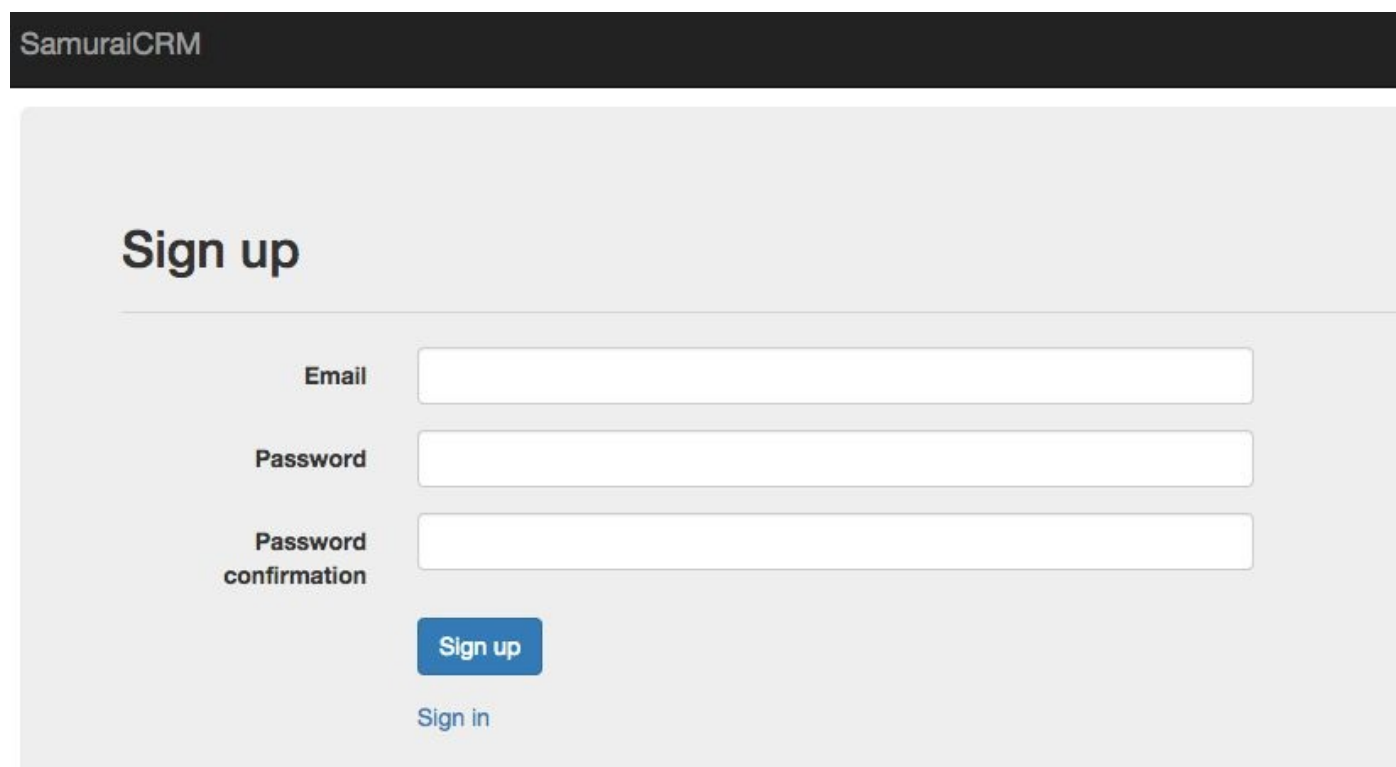
The screenshot shows a web application header with the text "SamuraiCRM" in white on a dark background. Below the header is a light gray box containing a "Sign up" form. The form has the title "Sign up" in bold. It includes three input fields: "Email", "Password", and "Password confirmation". Below the "Password" field is a blue "Sign up" button. At the bottom of the form is a link that says "Sign in".

Figure 4.6

We're going to skip the forgot password and a few other Devise views because, well, it's not really interesting. Let's play with authentication instead!

Step 13: Register & Sign In

Moment of truth. Let's try to create a user and sign it. If everything goes correctly, you should end up on the dashboard we created earlier.

Step 14: Add edit user link

That would be nice if users could edit their information... We're going to add it then. It's also the perfect time to add a few things to our navigation bar which is still quite empty.

Devise gives us the **current_user** method and we're going to use to show the nav bar only to people who logged in.

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->
```



```

<!-- ... -->
<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <%= link_to 'SamuraiCRM', samurai.root_path, class: 'navbar-brand' %>
    </div>
    <%= if current_user %>
      <div class="navbar-collapse collapse" id="navbar">
        <ul class="nav navbar-nav">
          <li>
            <%= link_to 'Home', samurai.root_path %>
          </li>
          <li>
            <%= link_to 'My Account', samurai.edit_user_registration_path %>
          </li>
          <li>
            <%= link_to 'Logout', samurai.destroy_user_session_path,
              method: :delete %>
          </li>
        </ul>
      </div>
    <%= end %>
  </div>
</nav>
<!-- ... -->

```

And we're getting: [Figure 4.7](#)

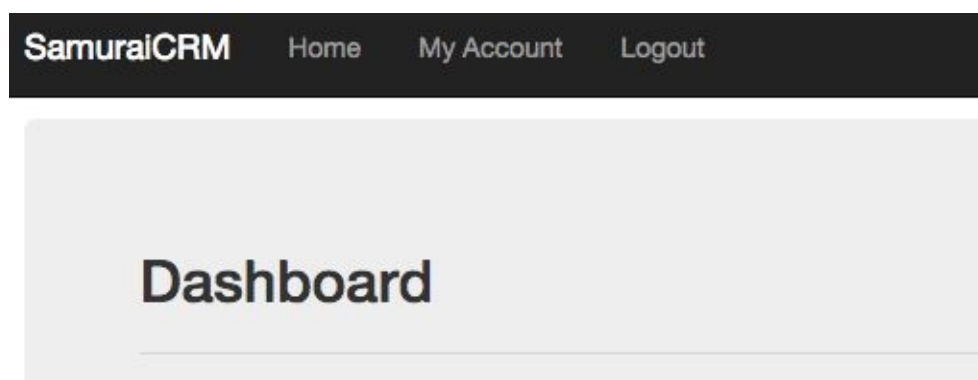


Figure 4.7

Neat!

Step 15: Update Devise edit user view

Click on 'My Account'. [Figure 4.8](#)

Edit User

Email

Password *(leave blank if you don't want to change it)*

Password confirmation

Current password *(we need your current password to confirm your changes)*

Cancel my account

Unhappy?

[Back](#)

Figure 4.8: Ew.

Here's the updated code for that view:

```
<!-- SamuraiCRM/engines/core/app/views/devise/registrations/edit.html.erb -->
<h2>Edit <%= resource_name.to_s.humanize %></h2>
<hr>
<%= form_for(resource, as: resource_name, url: registration_path(resource_name),
              html: { method: :put, class: 'form-horizontal' }) do |f| %>
<%= devise_error_messages! %>

<div class="form-group">
  <%= f.label :email, class: 'col-sm-2 control-label' %>
  <div class="col-sm-6">
    <%= f.email_field :email, class: 'form-control' %>
  </div>
</div>

<div class="form-group">
  <%= f.label :password, class: 'col-sm-2 control-label' %>
  <i>(leave blank if you don't want to change it)</i>
  <div class="col-sm-6">
    <%= f.password_field :password, autocomplete: "off",
                        class: 'form-control' %>
  </div>
</div>
</div>
```

```

<div class="form-group">
  <%= f.label :password_confirmation, class: 'col-sm-2 control-label' %>
  <div class="col-sm-6">
    <%= f.password_field :password_confirmation, autocomplete: "off",
                        class: 'form-control' %>
  </div>
</div>

<div class="form-group">
  <%= f.label :current_password, class: 'col-sm-2 control-label' %>
  <i>(we need your current password to confirm your changes)</i>
  <div class="col-sm-6">
    <%= f.password_field :current_password, autocomplete: "off",
                        class: 'form-control' %>
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-6">
    <%= f.submit "Update", class: "btn btn-primary" %>
  </div>
</div>
<% end %>

<h2>Cancel my account</h2>
<hr>
<p>Unhappy?
  <%= button_to "Cancel my account", registration_path(resource_name),
                data: { confirm: "Are you sure?" },
                method: :delete,
                class: 'btn btn-danger' %></p>
<hr>
<%= link_to "Back", :back, class: 'btn btn-default' %>

```

Here's the result! [Figure 4.9](#)

Edit User

Email	<input type="text" value="bo@samurails.com"/>	
Password	<input type="password"/>	(leave blank if you do not want to change)
Password confirmation	<input type="password"/>	
Current password	(we need your current password to confirm your changes)	
	<input type="password"/>	
	<input type="button" value="Update"/>	

Cancel my account

Unhappy?

Figure 4.9: Glorious, isn't it?

Step 16: Add active link to menu

Knowing on which menu we are currently would be nice, so let's add a highlight. The best way to do that, in my opinion, is to use a Helper method. So let's add an active method that's going to check if the current page matches a specific link.

```
# SamuraiCRM/engines/core/app/helpers/samurai/application_helper.rb
def active(path)
  current_page?(path) ? 'active' : ''
end
```

If the passed path equals the current page, we'll return the string active and use it as the link class. You can use it in the nav bar like this:

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->
<!-- ... -->
<ul class="nav navbar-nav">
  <li class="<%= active(samurai.root_path) %>">
    <%= link_to 'Home', samurai.root_path %>
  </li>
  <li class="<%= active(samurai.edit_user_registration_path) %>">
    <%= link_to 'My Account', samurai.edit_user_registration_path %>
  </li>
  <li>
    <%= link_to 'Logout', samurai.destroy_user_session_path, method: :delete %>
  </li>
</ul>
<!-- ... -->
```

Super easy, try it now! The menu you're currently on should be highlighted. [Figure 4.10](#)

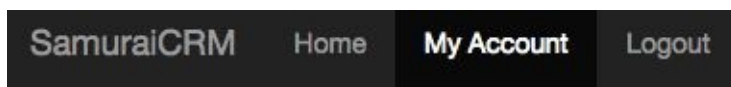


Figure 4.10

Step 17: Test it

We will write some automated tests very soon, but for now, just ensure that everything is working. Login, logout, navigation, everything! If it looks fine, let's continue and start working on the admin panel!

4.2.5 Part 5: The admin panel

In our CRM, we want to give the rights to administrate the CRM to some users. The admin panel will be a new screen available only to administrators where the users (among other things) can be managed. Later, we will extend it to list all contacts, opportunities and see what everyone's doing. To be able to create administrators, we just need to add an admin flag to users.

Step 1: Create new migration

Navigate inside the **Core** engine and run the migration generation command.

```
rails generate migration add_admin_to_samurai_users admin:boolean
```

Step 2: Migrate

Run the migration from the parent application.

```
rake db:migrate
```

Step 3: Change an existing user to Admin

Now, we need an admin! Easiest way to do that is by running the rails console and updating a user that you previously created. Run the following from the parent app.

```
rails console
```

Once the console is loaded:

```
Samurai::User.first.update_column :admin, true
```

Now that we have an admin, we can start implementing the views!

Step 4: Create the Admin folder and controller

To keep the administration logic encapsulated, we're going to add an admin namespace. First, create a folder named **admin** in **core/app/controllers/samurai/**. Inside this new folder, create a controller named **admin_controller.rb** and add the following content inside. You can also run the command below to generate the folder and the file.

```
mkdir app/controllers/samurai/admin && \  
touch app/controllers/samurai/admin/admin_controller.rb
```

The code for the **AdminController**.

```
# SamuraiCRM/engines/core/app/controllers/samurai/admin/admin_controller.rb  
module Samurai  
  module Admin  
    class AdminController < ApplicationController  
      def index  
      end  
    end  
  end  
end
```

Now, we need some routes!

Step 5: Add Admin routes

The admin panel should be available at **/admin** and the default action should be the index created in **AdminController** above. By using the namespace method, we separate the admin part from the rest of the application and we set the default to route to the **index** action on **AdminController**.

```
# SamuraiCRM/engines/core/config/routes.rb
Samurai::Core::Engine.routes.draw do
  devise_for :users, :class_name => "Samurai::User", module: :devise
  namespace :admin do
    get '/' => 'admin#index'
  end
  root to: "dashboard#index"
end
```

Step 6: Add admin link to dashboard

Now that the route is defined, we can add a link by using the path generated by Rails. Let's add the link to the admin panel between 'My Account' and 'Logout'.

```
<!-- SamuraiCRM/engines/core/app/views/layouts/samurai/application.html.erb -->
<!-- ... -->
<li class="<%= active(samurai.edit_user_registration_path) %>">
  <%= link_to 'My Account', samurai.edit_user_registration_path %>
</li>

<%- if current_user.admin? %>
  <li class="<%= active(samurai.admin_path) %>">
    <%= link_to 'Admin', samurai.admin_path %>
  </li>
<% end %>

<li>
  <%= link_to 'Logout', samurai.destroy_user_session_path, method: :delete %>
</li>
<!-- ... -->
```

Don't forget to use **samurai.admin_path**, else we will get a crash when accessing one of Devise views.

Step 7: Add the Index view for Admin

If you click on the link we just created, you will get an exception saying that we don't have a view yet. Let's create the view at the following location. Note that you need to create two folders named 'admin' before creating the actual file. The first folder is for the namespace **Admin** and the second one for our controller named **AdminController**.

```
core/
  app/
    views/
      samurai/
        admin/
          admin/
```

```
index.html.erb
```

Here's the command to create these folders. Run it from the **Core** engine.

```
mkdir -p app/views/samurai/admin/admin && \  
touch app/views/samurai/admin/admin/index.html.erb
```

And paste the following code in **index.html.erb**. Then read it to understand what we're doing!

```
<!-- SamuraiCRM/engines/core/app/views/samurai/admin/admin/index.html.erb -->  
<h2 class='pull-left'>Admin</h2>  
<ul class="nav nav-pills navbar-right">  
  <li role="presentation" class="<%= active(admin_path) %>">  
    <%= link_to 'Dashboard', admin_path %>  
  </li>  
</ul>  
  
<div class='clearfix'></div>  
<hr>  
  
<div class="row">  
  <div class="col-md-6">  
    <!--Will show tables with the last changes for each model -->  
  </div>  
</div>
```

You should see the following when accessing **/admin**. [Figure 4.11](#)

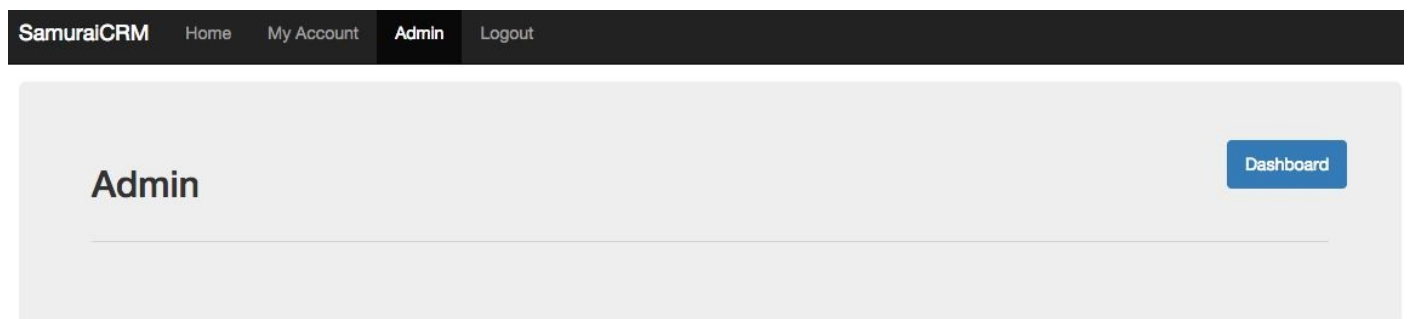


Figure 4.11

We just created a new menu for everything related to the administration of the CRM. Let's see what we can add to our empty admin panel!

Step 8: Create Admin Users Controller

The only thing we can show for now inside the admin dashboard is the list of all the users. To do that, we're going to create a new controller.

```
# SamuraiCRM/engines/core/app/controllers/samurai/admin/users_controller.rb  
module Samurai  
  module Admin  
    class UsersController < AdminController  
      def index
```



```

        @users = Samurai::User.ordered
      end
    end
  end
end

```

Once again, a very simple controller. But let's not forget that we are building a super simple sample application!

Step 9: Add Users resources to Admin Routes

Now we need to add one line to the routes file to access the list of users. We just need the index action since we just want to list the users. If you feel the need to, you can add the rest of the RESTful actions to handle **Users** creation, update and so on.

```

# SamuraiCRM/engines/core/config/routes.rb
# ...
namespace :admin do
  get '/' => 'admin#index'
  resources :users, only: :index
end
# ...

```

Step 10: Add Users link to admin panel

We can now add a link to the users listing view. But since we're going to re-use the admin menu in all the admin views, it's time to extract it to a partial.

Before we do that, there is a neat little trick to avoid having to write long path for our partials due to our namespacing policy. The trick is simply to add the path containing our namespaces to the list of folders looked up by Rails when it is looking for partials. We need to add one line to the **engine.rb** in the **Core** module. Restart your server after adding this code.

```

# SamuraiCRM/engines/core/lib/samurai/core/engine.rb
module Samurai
  module Core
    class Engine < ::Rails::Engine
      isolate_namespace Samurai

      paths["app/views"] << "app/views/samurai"

      # ...
    end
  end
end

```

With this, instead of having to write the following to render a partial:

```

<%= render 'samurai/admin/shared/nav' %>

```

We can just write:

```
<%= render 'admin/shared/nav' %>
```

You have to restart your server before continuing.

You already know the next step! Let's create a folder named `shared` in `core/app/views/samurai/admin/` and a file named `_nav.html.erb`. Inside, we're going to put the Admin menu:

```
<!-- SamuraiCRM/engines/core/app/views/samurai/admin/shared/_nav.html.erb -->
<ul class="nav nav-pills navbar-right">

  <li role="presentation" class="<%= active(samurai.admin_path) %>">
    <%= link_to 'Dashboard', samurai.admin_path %>
  </li>

  <li role="presentation" class="<%= active(samurai.admin_users_path) %>">
    <%= link_to samurai.admin_users_path do %>
      Users
      <span class="badge"><%= Samurai::User.count %></span>
    <% end %>
  </li>
</ul>
<div class='clearfix'></div>
```

In this view, we find the dashboard link that we created earlier as well as a new one linking to the users list. We also used one of Bootstrap element to add a counter next to the name with the total number of users.

Let's replace the old admin menu with this partial inside the admin dashboard view.

```
<!-- SamuraiCRM/engines/core/app/views/samurai/admin/admin/index.html.erb -->
<h2 class='pull-left'>Admin</h2>
<%= render 'admin/shared/nav' %>
<hr>

<div class="row">
  <div class="col-md-6">
    <!--Will show tables with the last changes for each model -->
  </div>
</div>
```

Step 11: Add User listing

Before adding the users table, we're going to add a scope to the `User` model to have them sorted by creation date.

```
# SamuraiCRM/engines/core/app/models/samurai/user.rb
module Samurai
  class User < ActiveRecord::Base
    devise :database_authenticatable, :registerable,
           :recoverable, :rememberable, :trackable, :validatable

    scope :ordered, -> { order('created_at desc') } # NEW SCOPE
  end
end
```

```
end
end
```

Finally, we can add the users listing! We need to create a new folder named **users** in **core/app/views/samurai/admin/**. Add the index file in this new folder.

```
<!-- SamuraiCRM/engines/core/app/views/samurai/admin/users/index.html.erb -->
<h2 class='pull-left'>Users</h2>
<%= render 'admin/shared/nav' %>
<hr>
<div class="panel panel-primary">
  <div class="panel-heading">
    All Users
  </div>

  <table class="table">
    <thead>
      <th>ID</th>
      <th>Email</th>
      <th>Signed Up</th>
      <th>Admin?</th>
    </thead>
    <tbody>
      <%- @users.ordered.each do |user| %>
        <tr>
          <td><%= user.id %></td>
          <td><%= user.email %></td>
          <td><%= user.created_at.strftime("%d %b. %Y") %></td>
          <td><%= user.admin? %></td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
```

If you access the user admin screen, you should now have a beautiful table listing the users in your application! [Figure 4.12](#)

SamuraiCRM

Home

My Account

Admin

Logout

Dashboard

Users 1

Users

All Users

ID	Email	Signed Up	Admin?
1	bo@samurails.com	11 Feb. 2015	true

Figure 4.12

Step 12: Add a summary to the Admin Dashboard

The dashboard in the admin panel is supposed to show you a bit of everything so you can know what's going on quickly. So we're going to add a listing of the last 3 users created.

```
<!-- SamuraiCRM/engines/core/app/views/samurai/admin/admin/index.html.erb -->
<h2 class='pull-left'>Admin</h2>
<%= render 'admin/shared/nav' %>
<hr>

<div class="row">
  <div class="col-md-6">
    <div class="panel panel-primary">
      <div class="panel-heading">
        Last 3 Users
      </div>
      <table class="table">
        <thead>
          <th>ID</th>
          <th>Email</th>
          <th>Signed Up</th>
        </thead>
        <tbody>
          <%- Samurai::User.ordered.limit(3).each do |user| %>
            <tr>
              <td><%= user.id %></td>
              <td><%= user.email %></td>
              <td><%= user.created_at.strftime("%d %b. %Y") %></td>
            </tr>
          <% end %>
        </tbody>
      </table>
      <div class="panel-body text-center">
        <%= link_to '...', samurai.admin_users_path %>
      </div>
    </div>
  </div>
</div>
```

The result: [Figure 4.13](#)

SamuraiCRM

Home

My Account

Admin

Logout

Admin

Dashboard

Users 1

Last 3 Users

ID	Email	Signed Up
1	bo@samurails.com	11 Feb. 2015
...		

Figure 4.13

Step 13: A small bug

While playing with the app, you may have noticed a small bug. When you're on the users listing in the admin section, the admin link in the main navigation bar is not shown as active anymore.

I'm gonna let you find a way to fix it by yourself. By doing that, you will assimilate what we saw and get a better grasp of the current application. You should do it while the application is still simple.

In the next part, we're going to add authorization with the CanCan gem.

4.2.6 Part 6: Authorization

Authorization is here to ensure that users don't access some pages of the application that they are not supposed to see. For example, a regular user is not supposed to see the admin panel. CanCan, created by Ryan Bates, is an easy to use gem that let you do just that!

Step 1: Add the CanCan gem

The first step is to actually add the gem to the **Core gemspec**.

```
# SamuraiCRM/engines/core/samurai_core.gemspec
# ...
s.add_dependency 'devise', '~> 3.4.1'
s.add_dependency 'cancan', '~> 1.6.10'
# ...
```

And add it to the **core.rb** file:

```
# SamuraiCRM/engines/core/lib/samurai/core.rb
require 'sass-rails'
require 'bootstrap-sass'
require 'autoprefixer-rails'
require 'devise'
require 'cancan'

module Samurai
  module Core

    end
  end
end
```

And **bundle install** from the parent application. Don't forget to restart your app!

Step 2: Generate the Ability file

What a user can or cannot do is defined inside the **ability.rb** file. Let's generate this file

right now by using CanCan generator inside the **Core** module.

```
rails g cancan:ability
```

The file **ability.rb** will be generated in **core/app/models/**. Move it to **core/app/models/samurai/** with the following command. Run it from the **Core** engine.

```
mv app/models/ability.rb app/models/samurai/
```

Add the required namespace:

```
# SamuraiCRM/engines/core/app/models/samurai/ability.rb
module Samurai
  class Ability
    include CanCan::Ability
    def initialize(user)
    end
  end
end
```

Step 3: Fix the **current_ability**

CanCan uses a method called **current_ability** to get an ability object. Since we are using custom namespaces, we need to override this method. Do the following in the **ApplicationController** inside the **Core** module.

```
# SamuraiCRM/engines/core/app/controllers/samurai/application_controller.rb
module Samurai
  class ApplicationController < ActionController::Base
    before_action :authenticate_user!
    def current_ability
      @current_ability ||= Samurai::Ability.new(current_user)
    end
  end
end
```

Step 4: Add exception catching

By default, CanCan raises an exception when someone tries to access a forbidden resource. We just want to show a 403 page when this happens so we need to add a catch for the CanCan exception.

```
# core/app/controllers/samurai/application_controller.rb
module Samurai
  class ApplicationController < ActionController::Base
    before_action :authenticate_user!

    rescue_from CanCan::AccessDenied do |exception|
      render :file => "static/403.html", :status => 403, :layout => false
    end

    def current_ability
```

```
@current_ability ||= Samurai::Ability.new(current_user)
end
end
end
```

And create the static 403 file in the views:

```
<!-- SamuraiCRM/engines/core/app/views/samurai/static/403.html.erb -->
No can do.
```

Just put whatever you want inside.

Step 5: Add authorization check in controllers

Now we can start adding authorization checks to our controllers. First, the

DashboardController:

```
# SamuraiCRM/engines/core/app/controllers/samurai/dashboard_controller.rb
module Samurai
  class DashboardController < ApplicationController
    authorize_resource class: false

    def index
    end
  end
end
```

See the **authorize_resource**? That's the method we use to confirm the access for a user. We have to pass it class: false since this controller is not linked to any model.

Then, we need to add the same in the **AdminController**. Since all the other controllers in the admin namespace inherit from it, we just need to put it one time.

```
# SamuraiCRM/engines/core/app/controllers/samurai/admin/admin_controller.rb
module Samurai
  module Admin
    class AdminController < ApplicationController
      authorize_resource :class => false

      def index
      end
    end
  end
end
```

Step 6: Define Abilities

Finally, we have to define the abilities! They are defined in **ability.rb** as you probably know.

```
# SamuraiCRM/engines/core/app/models/samurai/ability.rb
module Samurai
  class Ability
```

```

include CanCan::Ability

def initialize(user)
  if user.admin?
    can :manage, :all
  else
    can :read, :dashboard
  end
end
end
end

```

An admin can do anything, but a user can only access the dashboard, for now at least. Don't forget to restart your server before you play with it. We will give the users more permissions when we create the next modules. Before arriving at the end of this (long) chapter, let's talk about testing!

4.2.7 Part 7: Testing

Testing should be a big part of your coding workflow. I decided to add tests after the code in this chapter to let you focus on the engines and how to create your first one. The truth is that testing an engine is not that different from testing a regular Ruby on Rails application. There are entire books available focusing only on testing and they offer better explanations than I could in just a few pages.

However, there are a few little tricks to know to make the tests run correctly inside your engine. That's the important part when testing a modular application and that's what I'm going to show you right now.

Step 1: Add gems to handle testing

Everyone has their favorite testing environment. Some people love factories, other prefer fixtures. I'm going to show you ONE way to test by using the tools I'm used to. You should adapt the following to your liking. Let's add the gems that we need to the **Core** module **gemspec**.

```

# SamuraiCRM/engines/core/samurai_core.gemspec
# ...
Gem::Specification.new do |s|
  # ...

  s.add_development_dependency "sqlite3"

  s.add_development_dependency "rspec-rails",
  s.add_development_dependency "factory_girl_rails"
  s.add_development_dependency "faker"
  s.add_development_dependency "database_cleaner"
end

```

Since we don't need those gems in production, we can add them as development dependency. As you can see, we're using Rspec, Factory Girl, Faker and Database Cleaner.

Don't forget to run **bundle install** from the **Core** engine this time. If you have the test folder generated by Rails in the **Core** engine, you can delete it. In the next steps, we will create the spec folder with Rspec.

Step 2: Generate Test DB

Let's generate the test database. Run the following command from the parent application folder.

```
rake db:create:all && rake db:migrate RAILS_ENV=test
```

Step 3: Generate Rspec

Navigate to the **Core** engine and run the command to generate the Rspec files.

```
rails g rspec:install
```

Step 4: spec_helper.rb

Since we use Rspec inside an engine, we need to tweak the file a little bit.

Here's the full **spec_helper** with comments about the various changes.

Here's what we did:

- Requiring the other testing tools we will be using: Factory Girl and DatabaseCleaner.
- Change the location of the support files
- Add DatabaseCleaner configuration
- Load the **Core** path helpers

And here's the file.

```
# SamuraiCRM/engines/core/spec/spec_helper.rb
ENV["RAILS_ENV"] ||= 'test'

require File.expand_path("../../../../../config/environment", __FILE__)
```

```

# Require Factory Girl and DatabaseCleaner
require 'rspec/rails'
require 'factory_girl_rails'
require 'database_cleaner'

# Set the ENGINE_RAILS_ROOT variable
ENGINE_RAILS_ROOT=File.join(File.dirname(__FILE__), '../')

# Requires supporting ruby files with custom matchers and macros, etc,
# in spec/support/ and its subdirectories.
Dir[File.join(ENGINE_RAILS_ROOT, "spec/support/**/*.rb")].each {|f| require f }

RSpec.configure do |config|

  config.expect_with :rspec do |expectations|
    expectations.include_chain_clauses_in_custom_matcher_descriptions = true
  end

  config.mock_with :rspec do |mocks|
    mocks.verify_partial_doubles = true
  end

  # Define how we want DatabaseCleaner to work
  config.before(:suite) do
    DatabaseCleaner.strategy = :transaction
    DatabaseCleaner.clean_with(:truncation)
  end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end

  # Load the Core path helpers
  config.include Samurai::Core::Engine.routes.url_helpers

end

```

Step 5: .rspec

I like to add the following to the **.rspec** file to get a better output when running the specs.

```

--color
--require spec_helper
--format documentation

```

Step 6: Run Tests

We can finally try to run the tests! From the **Core** engine folder, run **rspec** and you should get: [Figure 4.14](#)

```

No examples found.

Finished in 0.05399 seconds (files took 0.90254 seconds to load)
0 examples, 0 failures

```

Great, it's working! Let's add some factories and some simple tests now!

Step 7: Create User factory

Create the folders `core/spec/factories/samurai/` and create a file named `user.rb` inside.

```
# core/spec/factories/samurai/user.rb
require 'faker'

module Samurai
  FactoryGirl.define do
    factory :user, class: 'Samurai/User' do |f|
      f.email { Faker::Internet.email }
      password 'password'
      password_confirmation 'password'
      admin false
    end

    factory :admin, parent: :user do |f|
      admin true
    end
  end
end
```

It's a super simple factory as you can see! Note the use of `class: 'Samurai/User'` to link to the `User` model.

Step 8: Add User Tests

With our new factory, we can now create some specs. Create the folders `core/spec/models/samurai/` and create a file named `user_spec.rb` inside.

You can use:

```
mkdir -p spec/models/samurai && touch spec/models/samurai/user_spec.rb
```

There is nothing complicated in the `User` model for now, so we're going to add some pretty basic tests.

```
# spec/models/samurai/user_spec.rb
require 'spec_helper'

module Samurai
  describe User do
    it "has a valid factory" do
      expect(FactoryGirl.build(:user)).to be_valid
    end

    it "is invalid without an email" do
      expect(FactoryGirl.build(:user, email: nil)).to_not be_valid
    end
  end
end
```

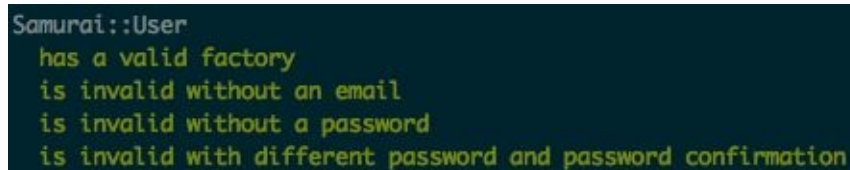
```

it "is invalid without a password" do
  expect(FactoryGirl.build(:user, password: nil)).to_not be_valid
end

it "is invalid with different password and password confirmation" do
  expect(FactoryGirl.build(:user, password: 'pass',
                           password_confirmation: 'pwd')).to_not be_valid
end
end
end

```

Run **rspec** from the **Core** engine folder and you should only get green lights! Now, you know how to tests models inside a Ruby on Rails engine. [Figure 4.15](#)



```

Samurai::User
  has a valid factory
  is invalid without an email
  is invalid without a password
  is invalid with different password and password confirmation

```

Figure 4.15

Step 9: Add Dashboard Tests

Before we add the tests for the dashboard controller, we need to add two support files that will simplify our life. The first file is the configuration for Devise. Create the file **core/spec/support/devise.rb** and paste the following:

```

# core/spec/support/devise.rb
require 'devise'

RSpec.configure do |config|
  config.include Devise::TestHelpers, :type => :controller
  config.extend ControllerMacros, :type => :controller
  config.infer_spec_type_from_file_location!
end

```

That's the default configuration offered by Devise on Github.

The second file we need is to add a simple way to login users and admins in our specs. We also need to set the routes we want to use. Create the file **core/spec/support/controller_macros.rb**.

```

# core/spec/support/controller_macros.rb
module ControllerMacros
  def login_admin
    before(:each) do
      @request.env["devise.mapping"] = Devise.mappings[:admin]
      sign_in FactoryGirl.create(:admin)
    end
  end

  def login_user
    before(:each) do

```

```

    @request.env["devise.mapping"] = Devise.mappings[:user]
    user = FactoryGirl.create(:user)
    sign_in user
  end
end
def set_engine_routes
  before(:each) do
    @routes = Samurai::Core::Engine.routes
  end
end
end
end

```

Now we can use `login_admin` and `login_user` in our controller specs to get a logged in user. We can also call `set_engine_routes` to use the routes defined in the `Core`. If you don't call that method, you will get error when running `rspec`.

Finally, let's add some specs for `DashboardController`. Create the folders `core/spec/controllers/samurai/` and add a file named `dashboard_controller_spec.rb`.

You can use this command:

```

mkdir -p spec/controllers/samurai && \
touch spec/controllers/samurai/dashboard_controller_spec.rb

```

Here are the dashboard tests:

```

# spec/controllers/samurai/dashboard_controller_spec.rb
require 'spec_helper'

module Samurai
  describe DashboardController do
    set_engine_routes

    context 'signed out' do
      describe 'GET Index' do
        it 'does not have a current_user' do
          expect(subject.current_user).to be_nil
        end

        it 'redirects the user to login page' do
          get :index
          expect(subject).to redirect_to new_user_session_path
        end
      end
    end

    context 'user' do
      login_user

      describe 'GET Index' do
        it 'has a current_user' do
          expect(subject.current_user).to_not be_nil
        end

        it 'should get :index' do
          get :index
          expect(response).to be_success
        end
      end
    end
  end
end

```

```

    it 'renders the :index view' do
      get :index
      expect(response).to render_template :index
    end

  end

end

context 'admin' do
  login_admin

  it 'has a current_user' do
    expect(subject.current_user).to_not be_nil
  end

  it 'has a current_user who is an admin' do
    expect(subject.current_user.admin).to be true
  end

  it 'should get :index' do
    get :index
    expect(response).to be_success
  end

  it 'renders the :index view' do
    get :index
    expect(response).to render_template :index
  end
end
end
end
end

```

The tests are self-explanatory. You should read them! You can run them with **rspec** from the **Core** engine folder. [Figure 4.16](#)

```

Samurai::DashboardController
  signed out
    GET Index
      does not have a current_user
      redirects the user to login page
  user
    GET Index
      has a current_user
      should get :index
      renders the :index view
  admin
    has a current_user
    has a current_user who is an admin
    should get :index
    renders the :index view

Samurai::User
  has a valid factory
  is invalid without an email
  is invalid without a password
  is invalid with different password and password confirmation

Finished in 0.61128 seconds (files took 1.03 seconds to load)
13 examples, 0 failures

```

Figure 4.16

4.3 Round Up

We finally reached the end of this chapter. Let's go over everything we've learned!

4.3.1 What did we learn?

In this chapter, we created our first module.

To do so, we first generated an empty Ruby on Rails application which we called the parent application. Not much will be added to this app throughout the book since we will use modules to add new features.

Then, we actually generated the **core** module. This module is the foundation of our application and future modules will be built on top of it. Inside this module, we set in place the authentication and authorization. We also created the views that will be extended by other modules in the next chapters.

We also learned how to setup the testing tools inside a Ruby on Rails engine.

4.3.2 Main points to remember

- Engines are generated with **rails plugin new core -mountable**
- We need an empty Ruby on Rails application to hold our engines as gems
- When using gems inside an engine, they need to be required when the engine is loaded
- Some gems require special attention when we integrate them inside engines instead of regular Ruby on Rails applications

4.3.3 Next Step

In the next chapter, we will create the second module of our modular application. We will add a new feature to our CRM: the creation and management of contacts. We will also learn how to extend the functionalities of the **core** module inside another module.

Chapter 5

SamuraiCRM: The Contacts module

In the previous chapter, we built our first engine. I hope you learned a few things on the way. Things are about to get more interesting in this chapter. We're going to build the **Contact** engine on top of the **Core** by extending its classes and views.

Let's get started!

5.1 What's the Contacts module

In most CRM, you will find a **Contact** feature. This is where users can fill in information about their clients, leads and so on. SamuraiCRM will follow the same principle. We will add everything to handle contacts and link them to a user. We will also extend the dashboard to add a summary of a user's last contacts.

The **Contact** module will include:

- **Contact CRUD (models/controllers/views)**

We're going to create the basic views to create, edit and delete contacts. A user will be able to manage his contacts with these views. At first, we will simply list all the contacts.

- **A contacts link in the main navigation bar**

We want to be able to easily access the list of contacts from the menu.

- **Users and Contacts relationship**

Linking **User** with **Contact** will give us the option to show only the relevant contacts for each user.

- **Contact overview on the dashboard**

The dashboard is still quite empty. We're going to extend it and add a list of the last contacts.

5.2 Build it

5.2.1 Part 1: Generate the Contacts engine

To avoid having name conflicts with an engine named **Contact** and the model inside also named **Contact**, we need to give a more appropriate name to this engine. In this case, we're simply going to pluralize it and call it **Contacts**.

In the previous chapter, we had to do a lot of folder rearranging which was super boring. Actually, there is a better way to generate a modular engine the way it should look. But for the first chapter, I wanted you to understand deeply how and why we organize the engine like this so we went through all the annoying steps. Now, we're about to generate a new engine, a **feature** one (that gets plugged on a **Core** type engine).

To do that, I wrote a small and simple **gem** that we can use.

Step 1: Add the gems

Add the gems **modular_engine** and **deface** to the Parent **Gemfile** and run **bundle install**.

```
# SamuraiCRM/Gemfile
# ... gems...
gem 'modular_engine'
gem 'deface' # We will use it soon

gem 'samurai_core', path: 'engines/core'
```

Step 2: Generate the Contacts engine.

Run the following from the parent application folder:

```
rails g modular:engine engines/contacts --namespace=Samurai
```

And we get an almost ready to use engine! Great, isn't it?!

Step 3: Fix the new engine gemspec

We have to edit a few values in the **gemspec** (homepage, summary, description) and add

the **Core** engine as a dependency.

```
# SamuraiCRM/engines/contacts/samurai_contacts.gemspec
$.push File.expand_path("../lib", __FILE__)

# Maintain your gem's version:
require "samurai/contacts/version"

# Describe your gem and declare its dependencies:
Gem::Specification.new do |s|
  s.name        = "samurai_contacts"
  s.version     = Samurai::Contacts::VERSION
  s.authors    = ["Thibault Denizet"]    # Your Name
  s.email      = ["bo@samurails.com"]    # Your Email
  s.homepage   = "http://samurails.com"
  s.summary    = "Contact feature for SamuraiCRM."
  s.description = "Contact feature for SamuraiCRM."
  s.license    = "MIT"

  s.files = Dir["{app,config,db,lib}/**/*", "MIT-LICENSE", "Rakefile",
               "README.rdoc"]

  s.add_dependency "rails", "~> 4.2.0"
  s.add_dependency "samurai_core"
  s.add_development_dependency "sqlite3"
end
```

Step 4: Add the Core gem to the Contacts engine gemfile

We are working with local gems and the method `s.add_dependency` won't let us specify a local path. To load the gems correctly, we need to add **samurai_core** in the **Contacts Gemfile** with the local path.

```
# SamuraiCRM/engines/contacts/Gemfile
# ... other stuff...
gem 'samurai_core', path: '../core'
```

Step 5: Change the routes file

Since we are building this engine on top of the **Core** and we want to make it easy to work with a set of engines, all our **feature** engines will extend the **Core** routes instead of implementing their own. How do we do that? Well, it's very simple. Just open the **Contacts** routes file and change **Contacts** to **Core**.

```
# SamuraiCRM/engines/contacts/config/routes.rb
Samurai::Core::Engine.routes.draw do
end
```

Step 6: Add the new engine to the Parent Gemfile

```
# SamuraiCRM/Gemfile
```

```
# ... gems...
gem 'samurai_core', path: 'engines/core'
gem 'samurai_contacts', path: 'engines/contacts'
```

Now you can run a quick **bundle install** from the parent app, restart your server and everything should work! See, that wasn't that hard thanks to the **modular_engine** gem. Let's start adding some content to the **Contacts** engine.

5.2.2 Part 2: Contact Model

The first step to build the **Contacts** engine is to create the model. We'll then work on the controller.

Step 1: Navigate in the Contacts folder and run bundle install.

Step 2: Generate the Contact model:

```
rails g model Contact first_name:string last_name:string company:string \
email:string phone:string user:references --no-test-framework
```

Step 3: Rename the table in the generated migration.

Change **create_table :samurai_contacts** to **create_table :samurai_contacts_contacts** and **add_foreign_key :samurai_contacts, :users** to **add_foreign_key :samurai_contacts_contacts, :samurai_users**.

Step 4: Move the model at the right place.

The model was created under **app/contacts/samurai/** but we want it in **app/contacts/samurai/contacts/**. You can move it yourself or run the following command from the engine folder.

```
mv app/models/samurai/contact.rb app/models/samurai/contacts/contact.rb
```

Step 5: Migrate!

Run **rake db:migrate** from the parent app folder.

5.2.3 Part 3: Contacts Controller

Let's generate a controller for our new model.

Step 1: Generate it!

```
rails g scaffold_controller Contact --no-test-framework --no-helper
```

Step 2: Fix the files location

Once again, we need to move the controller and its views at the right place. You can run the following commands to get the desired result.

Run them from the engine folder.

```
mv app/controllers/samurai/contacts_controller.rb \
  app/controllers/samurai/contacts/contacts_controller.rb
```

```
mkdir app/views/samurai/contacts/contacts && \
mv app/views/samurai/contacts/*.erb app/views/samurai/contacts/contacts/
```

In the first command, we move the **contacts_controller.rb** file in **app/controllers/samurai/contacts/**. In the second one, we move all the views generated to a subfolder.

Step 3: Update the controller.

First, we need to change the parent of **ApplicationController** to the **ApplicationController** present in the **Core**.

```
# SamuraiCRM/engines/contacts/app/controllers/samurai/contacts/application_controller
module Samurai
  module Contacts
    class ApplicationController < Samurai::ApplicationController
    end
  end
end
```

We have a few things to change in the controller generated by Rails. First, we need to

prefix all the path helpers with samurails. Then, we need to update the `contact_params` method to include the parameters we want to keep.

Here's the entire controller, including comments about what we changed, so you can just copy/paste it and read it.

```
# contacts/app/controllers/samurai/contacts/contacts_controller.rb
module Samurai
  module Contacts
    class ContactsController < ApplicationController
      before_action :set_contact, only: [:show, :edit, :update, :destroy]

      def index
        @contacts = Contact.all
      end

      def show
      end

      def new
        @contact = Contact.new
      end

      def edit
      end

      def create
        @contact = Contact.new(contact_params)
        @contact.user = current_user
        if @contact.save
          # Add samurai to access the correct path
          redirect_to [samurai, @contact],
                    notice: 'Contact was successfully created.'
        else
          render :new
        end
      end

      def update
        if @contact.update(contact_params)
          # Add samurai to access the correct path
          redirect_to [samurai, @contact],
                    notice: 'Contact was successfully updated.'
        else
          render :edit
        end
      end

      def destroy
        @contact.destroy
        # Add samurai to access the correct path
        redirect_to samurai.contacts_url,
                  notice: 'Contact was successfully destroyed.'
      end

      private

      def set_contact
        @contact = Contact.find(params[:id])
      end

      def contact_params
        # Add the parameters we allow
        params.require(:contact).permit(:first_name, :last_name, :company,
                                         :email, :phone, :user_id)
      end
    end
  end
end
```

Step 4: Update the views.

I prepared some views so you don't have to think about it. They are basic views with **Bootstrap** forms and elements. The only thing that you should remember in those views in the use of **samurai.my_path** instead of just **my_path**.

index.html.erb

```
<!-- contacts/app/views/samurai/contacts/contacts/index.html.erb -->
<%= link_to 'New Contact', samurai.new_contact_path,
  class: 'pull-right btn btn-primary' %>

<h2>Listing Contacts</h2>
<hr>
<div class="panel panel-primary">
  <div class="panel-heading">
    My Contacts
  </div>

  <table class="table">
    <thead>
      <th>ID</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Company</th>
      <th>Email</th>
      <th>Phone</th>
      <th></th>
    </thead>
    <tbody>
      <% @contacts.each do |contact| %>
        <tr>
          <td><%= contact.id %></td>
          <td><%= contact.first_name %></td>
          <td><%= contact.last_name %></td>
          <td><%= contact.company %></td>
          <td><%= contact.email %></td>
          <td><%= contact.phone %></td>
          <td>
            <%= link_to 'Show', [samurai, contact],
              class: 'btn btn-primary' %>
            <%= link_to 'Edit', samurai.edit_contact_path(contact),
              class: 'btn btn-primary' %>
            <%= link_to 'Destroy', [samurai, contact],
              class: 'btn btn-primary', method: :delete,
              data: { confirm: 'Are you sure?' } %>
          </td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
<br>
```

form.html.erb

```
<!-- contacts/app/views/samurai/contacts/contacts/_form.html.erb -->
<div class="row">
  <div class="col-md-8">
    <% if @contact.errors.any? %>
      <div id="error_explanation">
        <h2>
          <%= pluralize(@contact.errors.count, "error") %>
          prohibited this contact from being saved:</h2>
        <ul>
          <% @contact.errors.full_messages.each do |message| %>
            <li><%= message %></li>
          <% end %>
        </ul>
      </div>
    </div>
  </div>
</div>
```

```

    </ul>
  </div>
<% end %>
<div class="form-group">
  <%= f.label :first_name, class: "control-label" %>
  <%= f.text_field :first_name, class: "form-control" %>
</div>
<div class="form-group">
  <%= f.label :last_name, class: "control-label" %>
  <%= f.text_field :last_name, class: "form-control" %>
</div>
<div class="form-group">
  <%= f.label :company, class: "control-label" %>
  <%= f.text_field :company, class: "form-control" %>
</div>
<div class="form-group">
  <%= f.label :email, class: "control-label" %>
  <%= f.email_field :email, class: "form-control" %>
</div>
<div class="form-group">
  <%= f.label :phone, class: "control-label" %>
  <%= f.text_field :phone, class: "form-control" %>
</div>
</div>
</div>

```

new.html.erb

```

<!-- contacts/app/views/samurai/contacts/contacts/new.html.erb -->
<h2>New Contact</h2>
<hr>

<%= form_for([samurai, @contact]) do |f| %>
  <%= render 'form', f: f %>

  <div class="form-group">
    <div>
      <%= f.submit "Create Contact", class: "btn btn-primary" %>
      <%= link_to 'Back', samurai.contacts_path, class: 'btn btn-default' %>
    </div>
  </div>

<% end %>

```

show.html.erb

```

<!-- contacts/app/views/samurai/contacts/contacts/show.html.erb -->
<h2><%= @contact.first_name %></h2>
<hr>

<div class="row">
  <div class="col-md-8">
    <strong>First Name:</strong>
    <%= @contact.first_name %>
    <br/>
    <strong>Last Name:</strong>
    <%= @contact.last_name %>
    <br/>
    <strong>Company:</strong>
    <%= @contact.company %>
    <br/>
    <strong>Email:</strong>
    <%= @contact.email %>
    <br/>
    <strong>Phone:</strong>
    <%= @contact.phone %>
  </div>
</div>
<hr>
<%= link_to 'Edit', samurai.edit_contact_path(@contact), class: "btn btn-primary" %>
<%= link_to 'Back', samurai.contacts_path, class: 'btn btn-default' %>

```


edit.html.erb

```
<!-- contacts/app/views/samurai/contacts/contacts/edit.html.erb -->
<h2>Editing Contact</h2>
<hr>

<%= form_for([samurai, @contact]) do |f| %>
  <%= render 'form', f: f %>
  <div class="form-group">
    <div>
      <%= f.submit "Update Contact", class: "btn btn-primary" %>
      <%= link_to 'Back', samurai.contacts_path, class: 'btn btn-default' %>
    </div>
  </div>
<%= end %>
```

That's it for the views!

Step 5: Add the routes!

If we want to access the contacts somehow, we need to define some routes for it.

```
# SamuraiCRM/engines/contacts/config/routes.rb
Samurai::Core::Engine.routes.draw do
  scope module: 'contacts' do
    resources :contacts
  end
end
```

Note that we are just extending the **Core** routes. That means that in any of our **Samurai** engines, all the routes are easily accessible by using **samurai.my_path**.

Step 6: Try it

Let's see how it looks like in the browser. Access **http://localhost:3000/contacts** and you should see: [Figure 5.1](#)

[New Contact](#)

Listing Contacts

My Contacts

ID	First Name	Last Name	Company	Email	Phone
----	------------	-----------	---------	-------	-------

Figure 5.1

Ow, not that great huh! Why don't we see the beautiful layout that we created in the

core? Simply because we have a layout in **Contacts** that overrides the **core** version.

Step 7: Remove the layouts folder.

Simply delete the layouts folder in the **Contacts** engine or run the following command from the **Contacts** folder.

```
rm -r app/views/layouts
```

Step 8: Try it again!

And after a server restart and a refresh. [Figure 5.2](#)

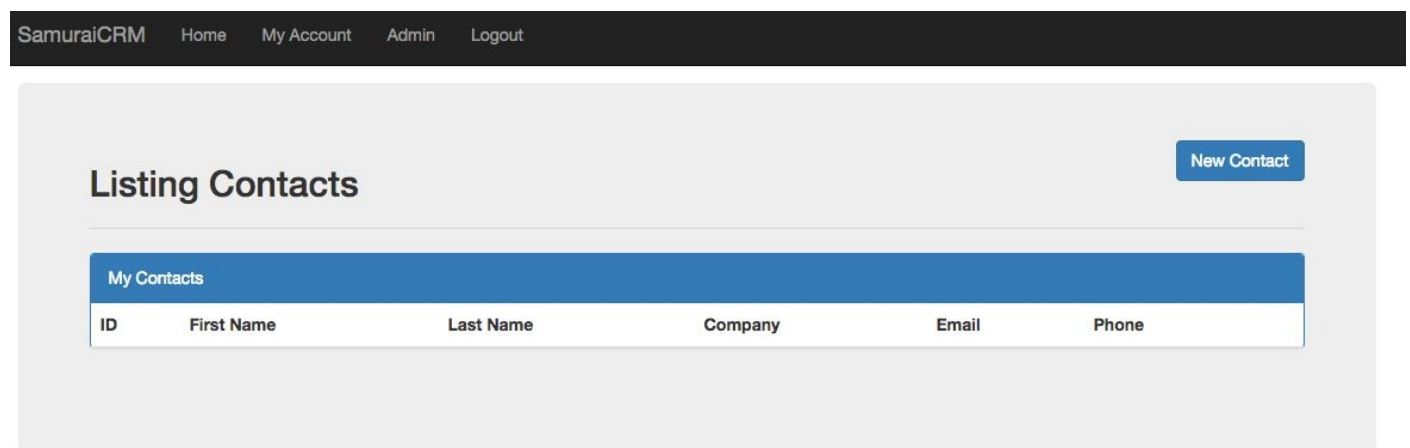


Figure 5.2

Nice! But hey, having to type the url is annoying, we want a link in our menu!

Let's fix that right now.

5.2.4 Part 4: Extending the Core views

So how are we going to extend the views that we wrote in the **core** engine from the **Contacts** engine? We're going to use a neat little gem called [Deface](#). It was created by [Spree](#).

Thanks to this gem, we can define hooks in our views and easily extend them. It's really easy to extend views as you're about to see.

Step 1: The Deface Gem

First, we need to add the Deface gem to our engine gemspec.

```
# SamuraiCRM/engines/contacts/samurai_contacts.gemspec
# ...
s.add_dependency "samurai_core"
s.add_dependency "deface", '~> 1.0.1'
# ...
```

Step 2: Require Deface

Then, if you remember correctly, we need to require this gem in

SamuraiCRM/engines/contacts/lib/samurai/contacts.rb:

```
# SamuraiCRM/engines/contacts/lib/samurai/contacts.rb
require "deface"

module Samurai
  module Contacts
    end
  end
end
```

Step 3: Bundle install

Let's run **bundle install** from the parent app to get Deface and restart the server to load our new gem.

Step 4: Add your first hook

Now, we can start on the interesting stuff.

First, what's a hook? In Deface terms, it's an element in your view that you can use to add some code. Either before, after, inside and so on. Theory sucks so let's create a hook right now to understand what it really is!

We're going to add a hook inside the **Core** engine in the layout file.

Locate the following piece of code:

```
<!-- core/app/views/layouts/samurai/application.html.erb -->
<!-- ... code... -->
<%= if current_user %>
  <div class="navbar-collapse collapse" id="navbar">
    <ul class="nav navbar-nav">

      <li data-samurai-hook='main_nav' class="<%= active(samurai.root_path) %>">
```

```

    <%= link_to 'Home', samurai.root_path %>
  </li>

  <li class="<%= active(samurai.edit_user_registration_path) %>">
    <%= link_to 'My Account', samurai.edit_user_registration_path %>
  </li>
  <%- if current_user.admin? %>
    <li class="<%= active(samurai.admin_path) %>">
      <%= link_to 'Admin', samurai.admin_path %>
    </li>
  <% end %>
  <li>
    <%= link_to 'Logout', samurai.destroy_user_session_path,
      method: :delete %>
  </li>
</ul>
</div>
<% end %>
<!-- ... code... -->

```

Did you notice what we added?

```

<li data-samurai-hook='main_nav' class="<%= active(samurai.root_path) %>">
  <%= link_to 'Home', samurai.root_path %>
</li>

```

See **data-samurai-hook='main_nav'**?

With this, we basically defined this list element as a hook. With Deface, we can now add some code before or after it very easily. Note that it doesn't have to be a **data-attribute**. It can be literally anything: an ID, a class or any kind of selectable element.

To keep the code clean, I like to use a unique **data-attribute** with different values. It makes it easy to find your hooks.

Now that we have a hook, we can create an override!

Step 5: Create an override

A Deface override is a small piece of code that will detect a hook and insert the code we tell it to. There is actually another way to use Deface, using their custom **DSL**, but I personally prefer the solution I'm going to show you. You can learn more about the other technique on the GitHub page.

If you didn't see it yet, there is a folder named overrides in **SamuraiCRM/engines/contacts/app**. It was generated by **modular_engine** and will be used to store our overrides.

Let's create a file named **add_contacts_link_to_nav.rb** and put the following content inside:

```

# SamuraiCRM/engines/contacts/overrides/add_contacts_link_to_nav.rb

```

```
Deface::Override.new(:virtual_path => "layouts/samurai/application",
                     :name => "add_contacts_link_to_nav",
                     :insert_after => "[data-samurai-hook='main_nav']",
                     :partial => "overrides/contacts_link",
                     :namespaced => true,
                     :original => 'f5fe48b6dc6986328e0b873b3ffa1b228dd52a7c')
```

Basically, we're instantiating the **Deface::Override** class with a set of options.

Wondering what each option means? The following is from the Deface GitHub and can be found [here](#).

- **virtual_path**

The template / partial / layout where the override should take effect eg:

shared/_person, **admin/posts/new** this will apply to all controller actions that use the specified template.

- **name**

Unique name for override so it can be identified and modified later. This needs to be unique within the same **:virtual_path**.

- **insert_after**

Inserts after all elements that match the supplied selector. One of the many options available like **insert_before**, **insert_top** and so on. Check the documentation for more options!

- **partial**

Relative path to a partial.

- **namespaced**

Namespace the override to avoid conflicts with other engines.

- **original**

String containing original markup that is being overridden. If supplied Deface will log when the original markup changes, which helps highlight overrides that need attention when upgrading versions of the source application. Only really warranted for **:replace** overrides. NB: All whitespace is stripped before comparison.

This is just the options we'll be using in this book. Don't hesitate to check what else you can do and adapt what I'm showing you to match your needs! Now, let's see how it looks like: [Figure 5.3](#)

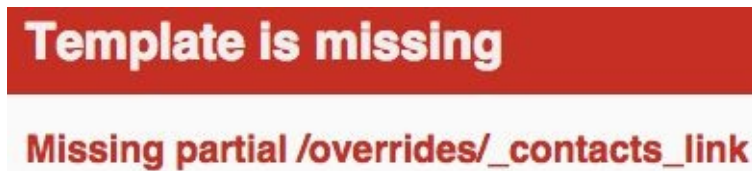


Figure 5.3

Oops. We didn't create the override view that's supposed to be inserted.

Step 6: Create a view for the override

We can create the override view in `app/views/samurai/contacts/overrides/` and the name needs to be the same as we defined in the previous step in `:partial`. In our case, the file should be named `_contacts_link.html.erb`:

```
<!-- contacts/app/views/samurai/contacts/overrides/_contacts_link.html.erb -->
<li class="<%= active(samurai.contacts_path) %>">
  <%= link_to 'Contacts', samurai.contacts_path %>
</li>
```

And after a quick refresh, we get: [Figure 5.4](#)

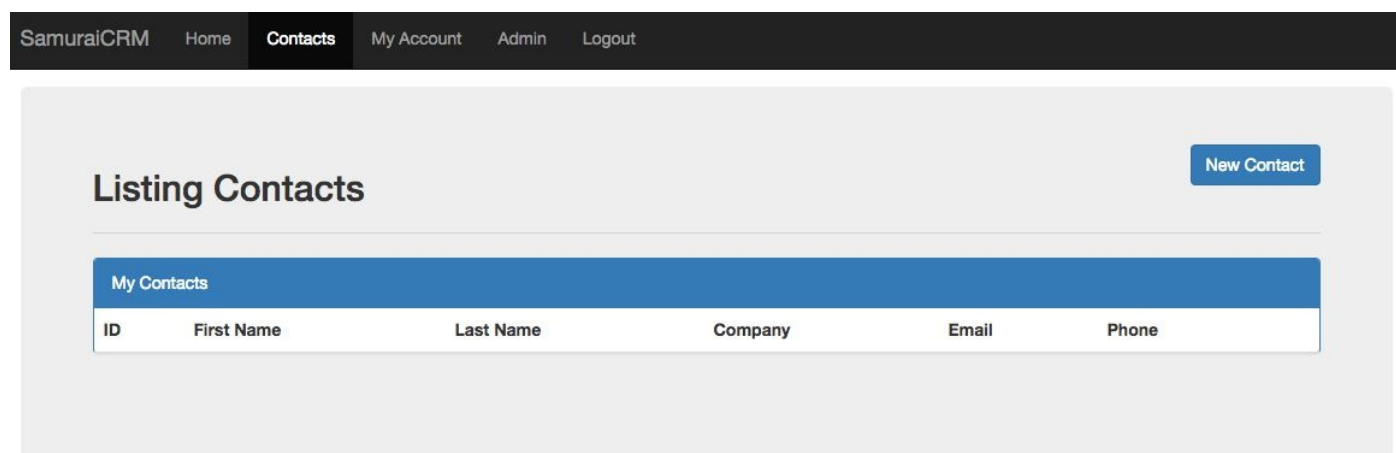


Figure 5.4

Wow, isn't that great? You can play with it, try removing the `Contacts` engine from the `Gemfile`, run `bundle install`, restart the server and refresh the page. No contacts link anymore! Now you know how to easily extend a view in another engine. Just to be sure, we'll do it a few more times in this chapter and in the next.

How about we extend some models now?

5.2.5 Part 5: Extending the Core models

Extending models is much easier than extending views. Thanks to Ruby open classes, we can always add more methods to an existing class.

Let's see how.

Step 1: The Decorator folder

To keep it organized, we'll extend models and controllers in files called decorators.

By using the **modular_engine** generator, you should have a **decorators** folder in your engine **app/** folder. In there, you will find two more folders: **models** and **controllers**. We're going to create one decorator for each class that we want to extend. But first, let's make sure our decorators will be loaded by **Rails** automatically.

Step 2: Load the decorators

To load the content of your decorators, we need to tell **Rails** to load them. Actually, with the gem we used to generate the engine, the piece of code to do it should already be there, but let's take a look.

Open the **engine.rb** file in the **Contacts** engine (located in **lib/samurai/contacts**).

You should see the following somewhere inside the engine file:

```
# ...
config.to_prepare do
  Dir.glob(Engine.root.join("app", "decorators", "**", "*_decorator*.rb")) do |c|
    Rails.configuration.cache_classes ? require(c) : load(c)
  end
end
# ...
```

And that's all the magic we need to load our decorators!

Step 3: Create a decorator for the User model

Now, let's try it. Create a **decorator** for **User** and put the following code inside:

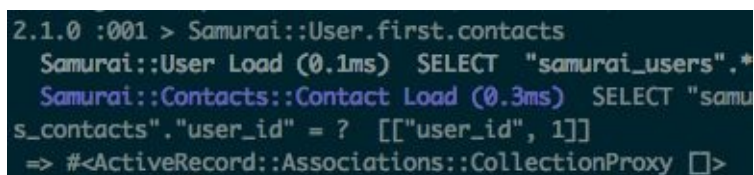
```
# SamuraiCRM/engines/contacts/app/decorators/models/user_decorator.rb
Samurai::User.class_eval do
  has_many :contacts, class_name: Samurai::Contacts::Contact
end
```

And that's it! You can put anything you'd like to add to the **User** model in the **class_eval** block. Note however, that we need to give the namespaced name of the module we're linking else it will try to find the **Contact** model in the **Samurai::User** scope!

Step 4: A quick test in the console

We should try it to be sure it's working. Let's fire up the **rails console** and type **Samurai::User.first.contacts** (Obviously, you need a user but you should have one by now).

You should get a **CollectionProxy** object showing up in your terminal. [Figure 5.5](#)



```
2.1.0 :001 > Samurai::User.first.contacts
Samurai::User Load (0.1ms) SELECT "samurai_users".*
Samurai::Contacts::Contact Load (0.3ms) SELECT "samurai_contacts"."user_id" = ? [{"user_id", 1}]
=> #<ActiveRecord::Associations::CollectionProxy >
```

Figure 5.5

Step 5: Update the Contacts controller

A few steps back, we added the possibility to show all the contacts to a user. But that's not the behavior we want. We want to show a user only his contacts.

To do this, we need to change the value we give to **@contacts** in the **index** action of the **ContactsController**.

From:

```
def index
  @contacts = Contact.all
end
```

To:

```
def index
  @contacts = current_user.contacts
end
```

Thanks to the **has_many** relationship that we added, it's working! Time to extend the dashboard!

5.2.6 Part 6: Extending the Dashboard

So you've just learned how to extend anything in any way you might want. But to learn something, you need to do, do, do and do again. That's what we're going to do now! The dashboard is so white it's hurting my eyes, I think it would be nice to see a table on this page showing the last contacts a user created.

Step 1: Add a hook

You know the deal already, let's add a hook to the dashboard view so we can easily extend it in the **Contacts** engine. We're going to call this hook dashboard.

```
<!-- SamuraiCRM/engines/core/app/views/samurai/core/dashboard/index.html.erb -->
<h2>Dashboard</h2>
<hr>

<div class="row">

  <!-- The Dashboard Hook -->
  <span data-samurai-hook='dashboard'></span>

</div>
```

A simple span is enough for the dashboard. We're going to insert code right after it with Deface.

Step 2: Add an override

Time to create a new override that will be responsible for adding a list of contacts on the dashboard making SamuraiCRM much better.

```
# SamuraiCRM/engines/contacts/app/overrides/add_contacts_list_to_dashboard.rb
Deface::Override.new(:virtual_path => "samurai/dashboard/index",
  :name => "add_contacts_list_to_dashboard",
  :insert_after => "[data-samurai-hook='dashboard']",
  :partial => "overrides/contacts_list",
  :namespaced => true)
```

Step 3: Add an override view

And finally, the list of contacts. This is a simple view with a table showing a user's contacts.

```
<!-- contacts/app/views/samurai/contacts/overrides/_contacts_list.html.erb -->
<div class="col-md-6">
```

```

<div class="panel panel-primary">
  <div class="panel-heading">
    Last 3 Contacts
  </div>
  <table class="table">
    <thead>
      <th>ID</th>
      <th>Name</th>
      <th>Email</th>
      <th>Created On</th>
    </thead>
    <tbody>
      <%= current_user.contacts.each do |contact| %>
        <tr>
          <td><%= contact.id %></td>
          <td><%= contact.first_name %> <%= contact.first_name %></td>
          <td><%= contact.email %></td>
          <td><%= contact.created_at.strftime("%d %b. %Y") %></td>
        </tr>
      <%= end %>
    </tbody>
  </table>

  <div class="panel-body text-center">
    <%= link_to '...', contacts_path %>
  </div>
</div>

```

Did you see how we got the list of contacts? We used `current_user.contacts` directly in the view. That would be better to put it in the controller and I'm gonna show you how to do it in a minute. First let's take a look.

Step 4: Admire!

[Figure 5.6](#)

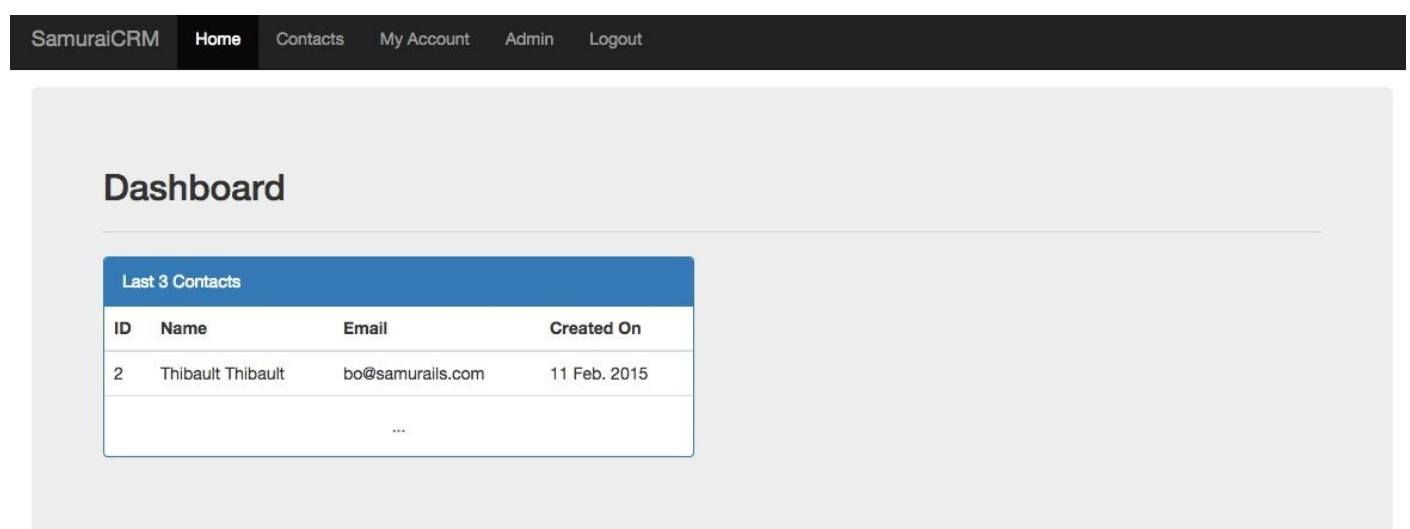


Figure 5.6

Now let's change how we get the list of contacts!

5.2.7 Part 7: Extending the Core controllers

To fix `current_user.contacts` and replace it by `@contacts` with an assignment in the controller, we need to override the `DashboardController`.

Step 1: Create a decorator for the Dashboard controller

Just like when we extended a model, we're going to create a **decorator**. A controller **decorator** to be precise. We cannot override the index because that would mean overriding it again in the next chapter. A better option is to use a `before_action` to assign the `@contacts` value.

```
# contacts/app/decorators/controllers/dashboard_controller_decorator.rb
Samurai::DashboardController.class_eval do
  before_action :set_contacts, only: :index

  private

  def set_contacts
    @contacts = current_user.contacts
  end
end
```

Step 2: Update the override view

Now we can change the override view that we created a few steps back and use `@contacts` instead of `current_user.contacts`.

```
<!-- contacts/app/views/samurai/contacts/overrides/_contacts_list.html.erb -->
<div class="col-md-6">
  <!-- ... -->
  <%= @contacts.each do |contact| %>
  <!-- ... -->
</div>
```

Step 3: Take a look

It should still be working! So now you know how to extend models, controllers and views! Is there anything else we have to extend? The answer is yes!

5.2.8 Part 8: Extending the abilities

There are some stuff that cannot be easily extended. A good example of this is the ability file provided by **CanCan**. You can use the following techniques for any classes that you want to add logic to but cannot simply open the class.

Step 1: Update the original ability.rb file

First, we need to add a bit of content to the ability file located in the **Core**.

```
# SamuraiCRM/engines/core/app/models/samurai/ability.rb
module Samurai
  class Ability
    include CanCan::Ability

    class_attribute :abilities
    self.abilities = Set.new

    # Allows us to go beyond the standard cancan initialize method which makes
    # it difficult for engines to modify the default {Ability} of an
    # application. The registered ability should behave properly as a
    # stand-alone class
    # and therefore should be easy to test in isolation.
    # @param ability [Ability] a class that includes the CanCan::Ability module.
    def self.register_ability(ability)
      self.abilities.add(ability)
    end

    # Remove a registered ability.
    # @param ability [Ability] a class that includes the CanCan::Ability module.
    def self.remove_ability(ability)
      self.abilities.delete(ability)
    end

    def initialize(user)
      Rails.logger.info self.abilities.inspect
      if user.admin?
        can :manage, :all
      else
        can :read, :dashboard
      end

      # Include any abilities registered by extensions, etc.
      Ability.abilities.each do |klass|
        ability = klass.send(:new, user)
        @rules = rules + ability.send(:rules)
      end
    end
  end
end
```

First, we added a **class_attribute** to the **Ability** class named **abilities**.

```
class_attribute :abilities
self.abilities = Set.new
```

Then, we added two class methods. These classes give us access to the abilities attribute and allow us to add or remove something from it. But what?

```
def self.register_ability(ability)
  self.abilities.add(ability)
end

def self.remove_ability(ability)
```

```
self.abilities.delete(ability)
end
```

Finally, at the end of the **initialize** method, we added this:

```
Ability.abilities.each do |klass|
  ability = klass.send(:new, user)
  @rules = rules + ability.send(:rules)
end
```

This code loop through the content of the abilities attribute and instantiate each element by passing the user object. Then, we get the rules, which is a **CanCan** method and contains all the permissions, from this new instance and add them to the rules of the current ability file.

Thanks to this code, we can simply create any number of **Ability** classes, define our permissions inside and add it to the abilities attribute of the original **Ability** file. If it sounds abstract, keep reading, we're going to do it right now!

Step 2: Create an ability decorator in the Contacts engine

Since the original **Ability** file is stored in the models folder, we're going to create our new **Ability** in the **decorator** folder as a model **decorator**.

```
# SamuraiCRM/engines/contacts/app/decorators/models/ability_decorator.rb
require 'cancan'
module Samurai
  module Contacts
    class AbilityDecorator
      include CanCan::Ability

      def initialize(user)
        unless user.admin?
          can :manage, Samurai::Contacts::Contact, user_id: user.id
        end
      end
    end
  end
end

# Registers the defined abilities.
Samurai::Ability.register_ability(Samurai::Contacts::AbilityDecorator)
```

See, we simply defined a new **Ability** class, called it **AbilityDecorator** and put some permissions. The magic happens thanks to this line:

```
Samurai::Ability.register_ability(Samurai::Contacts::AbilityDecorator)
```

We're calling the **register_ability** defined earlier to add this class to the abilities attribute.

Step 3: Add `authorize_resource` to `ContactsController`

To be sure it's working, let's add a `authorize_resource` to the contacts controller.

```
# SamuraiCRM/engines/contacts/controllers/samurai/contacts/contacts_controller.rb
module Samurai
  module Contacts
    class ContactsController < ApplicationController
      before_action :set_contact, only: [:show, :edit, :update, :destroy]
      authorize_resource class: Samurai::Contacts::Contact

      # ...
    end
  end
end
```

Step 4: Test

Restart your server and try to access a contact that don't belong to your current user.

You might have to create a new one in the console by using

`Samurai::Contacts::Contact.create`. Oh, and you should not be an admin. In my case, I'm getting the 403 page that we created in the previous chapter!

Box 5.1. Extending anything!

As I said earlier, this mechanism can be used for various purposes. I once used it to create an entity duplicator. The idea was to duplicate a record and all the linked records. So you would have `Entity1Duplicator`, `Entity2Duplicator` and so on. Each duplicator is a simple Ruby class that deals with duplicating the associated model. The problem was that some of the models I wanted to duplicate were in other engines. By using this technique, I was able to load a list of all the duplicators and call them one by one.

5.2.9 Part 9: Do It Yourself! Extend the admin Panel

I don't know if you've noticed but we didn't touch the admin panel in this chapter. But we have some work to do on it! There are three things that we should add.

- First, we need an admin contacts controller just like the one we created for users. The complete list of contacts should be accessible for admins only at `/admin/contacts`.
- The second thing we want is a link to the list of contacts we just created. We need to

add this link to the admin navigation menu.

- Finally, we should show the last 3 contacts created in the admin dashboard just like what we did for the last 3 users.

The good news is that I'm not going to show you how to do it. I want you to try it by yourself by using what you've learned in this chapter. I'm going to give you the steps to follow but you'll have to write the code yourself. Here are also a few screenshots to help you get started.

Here are the steps.

Step 1: Create an admin controller for contacts, an index view and admin routes

Step 2: Add hook in admin navigation

Step 3: Add override

Step 4: Add override view with Contacts link in Admin

Step 5: Add hook in admin dashboard

Step 6: Add override

Step 7: Add override view with Contacts list

Step 8 (Bonus): Create a decorator for the AdminController

Contacts

Dashboard

Users **2**

Contacts **3**

All Contacts					
ID	First Name	Last Name	Company	Email	Phone
1	Thibault	Denizet		bo@samurails.com	123
2	Thibault	Denizet		bo@samurails.com	
3	a	b		a@mail.co	

Figure 5.7: The Contacts list in the Admin Panel

Admin

Dashboard

Users **2**

Contacts **3**

Last 3 Users		
ID	Email	Signed Up
2	test@mail.com	11 Feb. 2015
1	bo@samurails.com	11 Feb. 2015
...		

Last 3 Contacts			
ID	Name	Email	Created On
1	Thibault Denizet	bo@samurails.com	11 Feb. 2015
2	Thibault Denizet	bo@samurails.com	11 Feb. 2015
3	a b	a@mail.co	11 Feb. 2015
...			

Figure 5.8: The Contacts summary in the Admin Dashboard

Good luck!

5.3 Round Up

We finally reached the end of this chapter. Let's go over everything we've learned!

5.3.1 What did we learn?

In this chapter, we learned everything about extending anything! Models, Controllers, Views and special cases like the **Ability** file.

We built our first **feature** module on top of the **Core** that we created in the last chapter. First, we created the **Contact** model, the associated controller and its views.

Then, we started the interesting part: extending the **Core** views to add some info about the contacts. But we also had to extend the **Core** models to get what we wanted. Finally, to remove some logic from the views, we also extended the **Core** controller.

We also learned that some classes cannot be extended in a simple way and we need to use a different technique to get it to work. That's what we did for the **Ability** class.

5.3.2 Main points to remember

Anything can be extended, but you need to keep it clean and organized. Deface is a great gem to extend views by using overrides. Adding the routes to the **Core** routes makes it easier to use them in all our engines (by calling **samurai.my_path**).

5.3.3 Next Step

In the next chapter, we're going to build a new module: **Tasks**. With this module, a user will be able to create tasks and link them to a contact if he wants to. We'll reuse what we learned in this chapter but also talk about how we can keep the **Contacts** module and the **Tasks** module independent.

Chapter 6

SamuraiCRM: The Tasks module

In the previous chapter, we built our first **feature** engine. We saw how we could extend the **Core** module by using Deface and **class_eval**. In this chapter, we're going to keep extending code from the **Core** module but also from the **Contacts** module. But we don't want **Tasks** to depend on **Contacts** so we'll see how we can link them while keeping the option to run them separately.

6.1 What's the Tasks module

In this chapter we're going to create the **Tasks** module which will add tasks to our CRM. The first part of this chapter might be a bit boring because we will be redoing the same steps that we did in the last chapter. You might want to just copy paste the code until the fourth part.

In the fourth part, we'll start extending the **Contacts** module and we'll see how we can simply check if an other engine is loaded or not. By doing this, we'll keep the **Tasks** engine independent which will allow us to run it even if we decide to remove the **Contacts** module.

The **Tasks** module will include:

- **Task CRUD (models/controllers/views)**

We're going to create the basic views to create, edit and delete **Tasks**. A User will be able to manage his tasks and assign them to other users and contacts.

- **A tasks link in the main navigation**

Typing **/tasks** in the URL bar every time we want to see our tasks wouldn't be very user-friendly.

- **Linking between User, Contact and Task**

A user **has_many** contacts and tasks and a contact can have many tasks associated. For example, a contact could have the following tasks linked to him 'call him at 09:00 tomorrow' or 'send email to check if still interested'. You get the idea. So we will show a list of tasks under each contacts.

6.2 Build it

6.2.1 Part 1: Generate the Tasks engine

Let's generate the **Tasks** engine and update some of its files. Just follow the steps!

Step 1: Generate the Tasks engine.

```
rails g modular:engine engines/tasks --namespace=Samurai
```

Step 2: Fix the new engine gemspec

```
# SamuraiCRM/engines/tasks/samurai_tasks.gemspec
$.push File.expand_path("../lib", __FILE__)
require "samurai/tasks/version"

# Describe your gem and declare its dependencies:
Gem::Specification.new do |s|
  s.name        = "samurai_tasks"
  s.version     = Samurai::Tasks::VERSION
  s.authors     = ["Thibault Denizet"]           # Your Name
  s.email       = ["bo@samurails.com"]           # Your Email
  s.homepage    = "http://samurails.com"
  s.summary     = "Task feature for SamuraiCRM."
  s.description = "Task feature for SamuraiCRM."
  s.license     = "MIT"

  s.files = Dir["{app,config,db,lib}/**/*", "MIT-LICENSE", "Rakefile",
               "README.rdoc"]
  s.add_dependency "rails", "~> 4.2.0"
  s.add_dependency "samurai_core"
  s.add_dependency "deface", "~> 1.0.1"
  s.add_development_dependency "sqlite3"
end
```

Step 3: Add the Core gem to the Tasks engine gemfile

Because we cannot specify a local path in the **gemspec**, we need to load the gem in the engine **gemfile**.

```
# SamuraiCRM/engines/tasks/Gemfile
# ... other stuff...
gem 'samurai_core', path: '../core'
```

Step 4: Change the routes file

```
# SamuraiCRM/engines/tasks/config/routes.rb
Samurai::Core::Engine.routes.draw do
end
```

Step 5: Add the new engine to the Parent Gemfile

```
# SamuraiCRM/Gemfile
# ... gems...
gem 'samurai_core', path: 'engines/core'
gem 'samurai_contacts', path: 'engines/contacts'
gem 'samurai_tasks', path: 'engines/tasks'
```

Step 6: bundle install and restart your server.

6.2.2 Part 2: Task Model

Now, it's time to create the **Task** model. Once again, it's pretty much the same steps that you found in the previous chapter, so let's go through it quickly.

Step 1: Navigate to the Tasks folder and run bundle install.

Step 2: Generate the Task model.

A task belongs to a user and to a potential contact. Run this command from the **Tasks** engine.

```
rails g model Task title:string content:text user:references \
contact:references --no-test-framework
```

Step 3: Rename the table in the generated migration.

Rename the table in the migration we just created.

```
# SamuraiCRM/engines/tasks/db/migrate/xxx_create_samurai_tasks
class CreateSamuraiTasks < ActiveRecord::Migration
  def change
    create_table :samurai_tasks_tasks do |t| # <- here
      t.string :title
      t.text :content
      t.references :user, index: true
      t.references :contact, index: true
      t.timestamps null: false
    end
  end
end
```

```
add_foreign_key :samurai_tasks_tasks, :samurai_users # <- here
add_foreign_key :samurai_tasks_tasks, :samurai_contacts_contacts # <- here
end
end
```

Step 4: Move the model at the right place.

```
mv app/models/samurai/task.rb app/models/samurai/tasks/task.rb
```

Step 5: Migrate!

Run **rake db:migrate** from the parent app folder.

6.2.3 Part 3: Tasks Controller & Views

In this part, we're going to add a controller and the basic views for our tasks.

Step 1: Generate the controller in the Tasks engine

```
rails g scaffold_controller Task --no-test-framework --no-helper
```

Step 2: Fix the files location

Fix the controller location:

```
mv app/controllers/samurai/tasks_controller.rb \
  app/controllers/samurai/tasks/tasks_controller.rb
```

Fix the views location:

```
mkdir app/views/samurai/tasks/tasks && \
mv app/views/samurai/tasks/*.erb app/views/samurai/tasks/tasks/
```

Step 3: Update the controllers.

The **ApplicationController**:

```
# SamuraiCRM/engines/tasks/app/controllers/samurai/tasks/application_controller.rb
module Samurai
```

```

module Tasks
  class ApplicationController < Samurai::ApplicationController
  end
end
end

```

And here's the code for the **TasksController**:

```

# SamuraiCRM/app/engines/tasks/app/controllers/samurai/tasks/tasks_controller.rb
module Samurai::Tasks
  class TasksController < ApplicationController
    before_action :set_task, only: [:show, :edit, :update, :destroy]

    def index
      @tasks = Task.all
    end

    def show
    end

    def new
      @task = Task.new
    end

    def edit
    end

    def create
      @task = Task.new(task_params)

      if @task.save
        redirect_to [samurai, @task], notice: 'Task was successfully created.'
      else
        render :new
      end
    end

    def update
      if @task.update(task_params)
        redirect_to [samurai, @task], notice: 'Task was successfully updated.'
      else
        render :edit
      end
    end

    def destroy
      @task.destroy
      redirect_to samurai.tasks_url, notice: 'Task was successfully destroyed.'
    end

    private
    def set_task
      @task = Task.find(params[:id])
    end

    def task_params
      params.require(:task).permit(:title, :content, :user_id, :contact_id)
    end
  end
end

```

Step 4: Update the views.

Copy/Paste the following views in the correct files.

index.html.erb

```

<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/index.html.erb -->
<%= link_to 'New Task', samurai.new_task_path,
  class: 'pull-right btn btn-primary' %>
<h2>Listing Task</h2>
<hr>
<div class="panel panel-primary">
  <div class="panel-heading">
    My Tasks
  </div>
  <table class="table">
    <thead>
      <th>ID</th>
      <th>Title</th>
      <th>Content</th>
      <th>Created At</th>
      <th></th>
    </thead>
    <tbody>
      <% @tasks.each do |task| %>
        <tr>
          <td><%= task.id %></td>
          <td><%= task.title %></td>
          <td><%= task.content %></td>
          <td><%= task.created_at.strftime("%d %b. %Y") %></td>
          <td>
            <%= link_to 'Show', [samurai, task], class: 'btn btn-primary' %>
            <%= link_to 'Edit', samurai.edit_task_path(task),
              class: 'btn btn-primary' %>
            <%= link_to 'Destroy', [samurai, task],
              class: 'btn btn-primary', method: :delete,
              data: { confirm: 'Are you sure?' } %>
          </td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
<br>

```

_form.html.erb

```

<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/_form.html.erb -->
<div class="row">
  <div class="col-md-8">
    <% if @task.errors.any? %>
      <div id="error_explanation">
        <h2>
          <%= pluralize(@task.errors.count, "error") %>
          prohibited this task from being saved:</h2>
        <ul>
          <% @task.errors.full_messages.each do |message| %>
            <li><%= message %></li>
          <% end %>
        </ul>
      </div>
    <% end %>

    <div class="form-group">
      <%= f.label :title, class: "control-label" %>
      <%= f.text_field :title, class: "form-control" %>
    </div>

    <div class="form-group">
      <%= f.label :content, class: "control-label" %>
      <%= f.text_area :content, class: "form-control" %>
    </div>

    <div class="form-group">
      <%= f.label :user_id, class: "control-label" %>
      <%= f.select :user_id, Samurai::User.all.collect { |p| [ p.email, p.id ] },
        { selected: current_user.id }, class: "form-control" %>
    </div>
  </div>
</div>

```


new.html.erb

```
<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/new.html.erb -->
<h2>New Task</h2>
<hr>
<%= form_for([samurai, @task]) do |f| %>
  <%= render 'form', f: f %>

  <div class="form-group">
    <div>
      <%= f.submit "Create Task", class: "btn btn-primary" %>
      <%= link_to 'Back', samurai.tasks_path, class: 'btn btn-default' %>
    </div>
  </div>
<%= end %>
```

show.html.erb

```
<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/show.html.erb -->
<h2><%= @task.title %></h2>
<hr>
<div class="row">
  <div class="col-md-8">
    <strong>Title:</strong>
    <%= @task.title %>
    <br/>
    <strong>Content:</strong>
    <%= @task.content %>
    <br/>
    <strong>User:</strong>
    <%= @task.user.email %>
    <br/>
  </div>
</div>
<hr>
<%= link_to 'Edit', samurai.edit_task_path(@task), class: "btn btn-primary" %>
<%= link_to 'Back', samurai.tasks_path, class: 'btn btn-default' %>
```

edit.html.erb

```
<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/edit.html.erb -->
<h2>Editing Task</h2>
<hr>

<%= form_for([samurai, @task]) do |f| %>
  <%= render 'form', f: f %>
  <div class="form-group">
    <div>
      <%= f.submit "Update Task", class: "btn btn-primary" %>
      <%= link_to 'Back', samurai.tasks_path, class: 'btn btn-default' %>
    </div>
  </div>
<%= end %>
```

Step 5: Add the routes!

```
# SamuraiCRM/engines/tasks/config/routes.rb
Samurai::Core::Engine.routes.draw do
  scope module: 'tasks' do
    resources :tasks
  end
end
```

Step 6: Remove the layouts folder.

```
rm -r app/views/layouts
```

Step 7: Take a peek

Finally, we can see the result of our efforts! Restart your server and access **/tasks**.

[Figure 6.1](#)

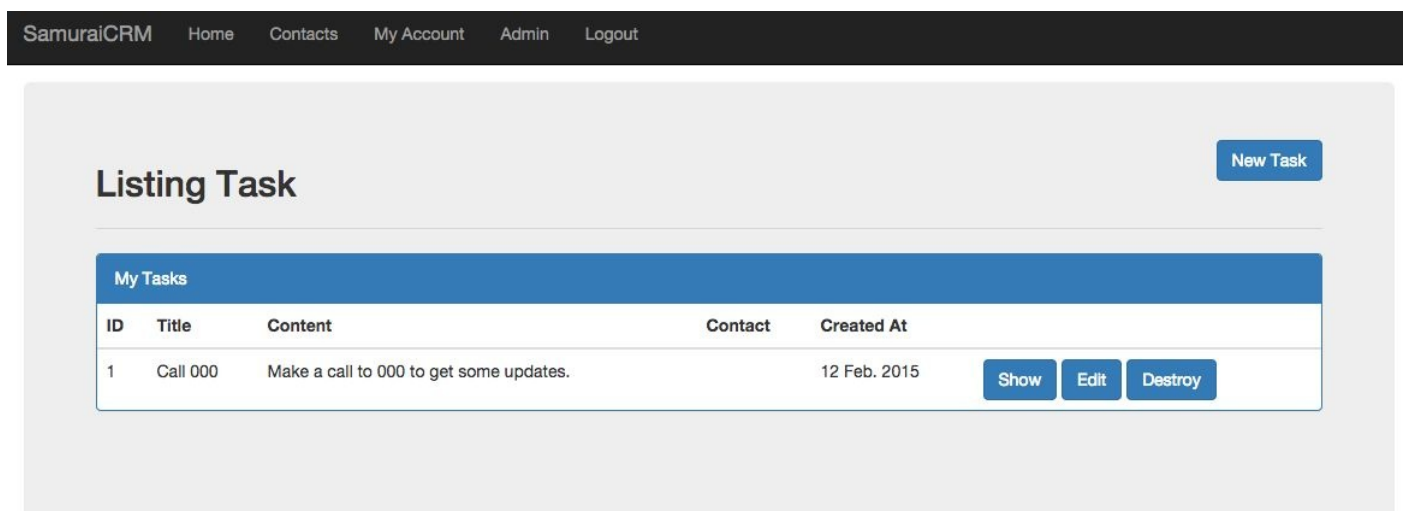


Figure 6.1: It looks Legen... wait for it.. dary!! Ahem.

6.2.4 Part 4: Adding tasks to the navigation menu

We're almost at the interesting part! Before we start extending the **Contacts** module, let's add a link in the navigation menu to access the list of tasks.

Step 1: Create an override

```
# SamuraiCRM/engines/tasks/app/overrides/add_tasks_link_to_nav.rb
Deface::Override.new(:virtual_path => "layouts/samurai/application",
  :name => "add_tasks_link_to_nav",
  :insert_after => "[data-samurai-hook='main_nav']",
  :partial => "overrides/tasks_link",
  :namespaced => true)
```

Step 2: Create a view for the override

```
<!-- tasks/app/views/samurai/tasks/overrides/_tasks_link.html.erb -->
<li class="<%= active(samurai.tasks_path) %>">
  <%= link_to 'Tasks', samurai.tasks_path %>
</li>
```

Step 3: Reload the app

Figure 6.2

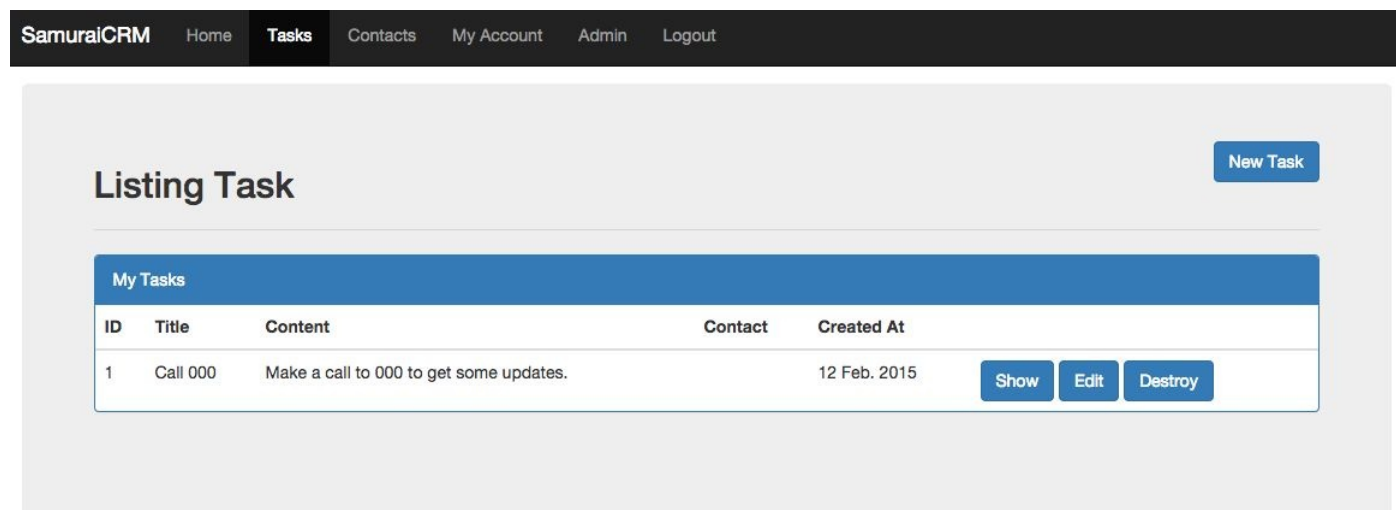


Figure 6.2

6.2.5 Part 5: Adding a list of tasks to Contacts

Now, we'd like to show a list of tasks on the **contact** show view. To do that, we need to link the **Task** and **contact** models. But what if we decide later to remove the **contacts** module? Everything would break!

The good news is that we can easily prevent this. We just have to check if the **contacts** module is present or not. This is the best solution I could come up with to deal with this kind of situations. I'm still looking for a better way to do it.

Basically, we're going to create a method in the **Core** to check if a specific module is defined. With this method, we can check if the engine containing the module **contacts** is present or not. Let's create this method.

Step 1: Add the available? method

We're going to define this method in the `core.rb` file and call it `available?`. The method is just going to check if the passed symbol (`:contacts`) is defined as a namespace under `Samurai` in the current application.

```
# SamuraiCRM/engines/core/lib/samurai/core.rb
require 'sass-rails'
require 'bootstrap-sass'
require 'autoprefixer-rails'
require 'devise'
require 'cancan'

module Samurai
  module Core
    def self.available?(engine_name)
      Object.const_defined?("Samurai::#{engine_name.to_s.camelize}")
    end
  end
end
```

Step 2: Update the Task form

With the `available?` method, we can update the task form to add a contact field. Now, we can link a task to a contact!

```
<!-- SamuraiCRM/app/engines/tasks/app/views/samurai/tasks/tasks/_form.html.erb -->
<!-- ... -->
<%= if Samurai::Core.available?(:contacts) %>
  <div class="form-group">
    <%= f.label :contact_id, class: "control-label" %>
    <%= f.select :contact_id,
      Samurai::Contacts::Contact.all.collect { |p| [ p.email, p.id ] },
      { allow_blank: true }, class: "form-control" %>
  </div>
<% end %>
</div>
</div>
```

We'll see the result in a minute.

Step 3: Add relations to our models

It's time to add a few relationships between our models. Note how we check if the `Contacts` module is present or not before trying to extend the `Contact` model.

```
# SamuraiCRM/engines/tasks/app/decorators/models/contact_decorator.rb
if Samurai::Core.available?(:contacts)
  Samurai::Contacts::Contact.class_eval do
    has_many :tasks, class_name: Samurai::Tasks::Task
  end
end
```

Nothing special for `User`. Just a simple `decorator`.

```
# SamuraiCRM/engines/tasks/app/decorators/models/user_decorator.rb
```

```
Samurai::User.class_eval do
  has_many :tasks, class_name: Samurai::Tasks::Task
end
```

Step 4: Add the related relations to Task

We can now add the other side of the relationships in the **Task** model. Once again, we reuse the **available?** method before linking **Task** to **Contact**.

```
# SamuraiCRM/engines/tasks/app/models/samurai/tasks/task.rb
module Samurai::Tasks
  class Task < ActiveRecord::Base
    belongs_to :user

    if Samurai::Core.available?(:contacts)
      belongs_to :contact, class_name: 'Samurai::Contacts::Contact'
    end
  end
end
```

Step 5: Check how it looks like!

Restart your server and you will see the result of the last steps. We now have a contact linked to a task.

[Figure 6.3](#)

The screenshot shows the 'New Task' form in the SamuraiCRM application. The form is located on a page with a dark header containing the application name and navigation links. The form itself has a light gray background and contains the following elements:

- Title:** A text input field.
- Content:** A larger text area for detailed notes.
- User:** A dropdown menu with 'bo@samurails.com' selected.
- Contact:** A dropdown menu with 'a@mail.co' selected.
- Buttons:** A blue 'Create Task' button and a white 'Back' button.

Figure 6.3: New Task

Figure 6.4

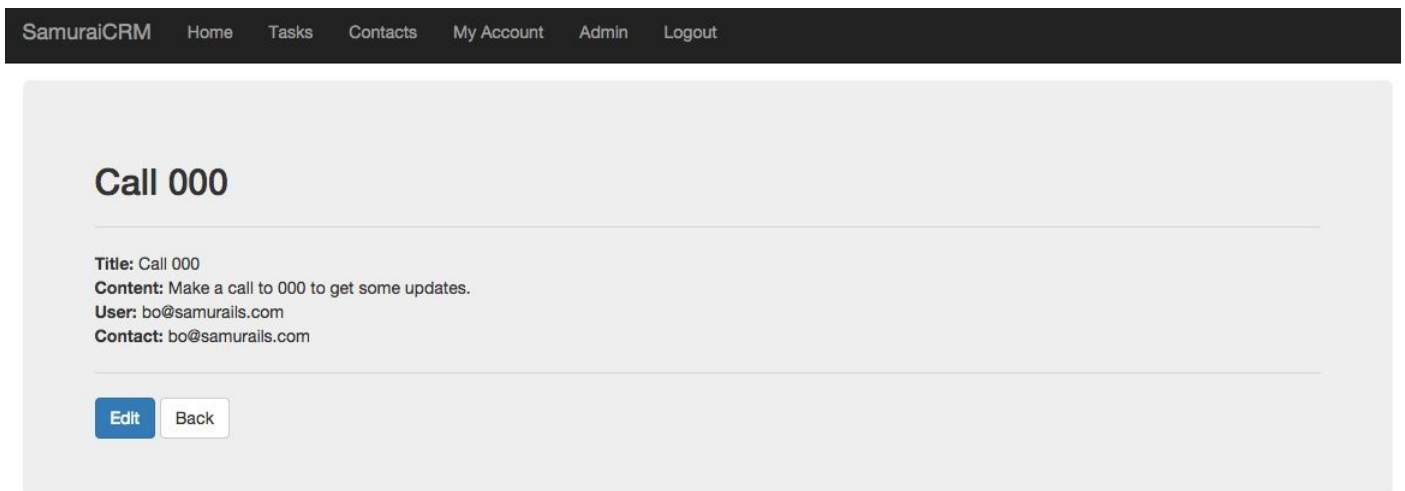


Figure 6.4: Show Task

Step 6: Create a hook in the contacts/show view

Before we add an override, we need a hook in the show view for contacts. Add the following line at the end of the file.

```
<!-- contacts/app/views/samurai/contacts/contacts/show.html.erb -->
<span data-samurai-hook='contacts_show'></span>
```

Step 7: Create an override

I'm sure you're an expert with Deface by now and you can extend views even with your eyes closed. Here's the override to add a list of tasks to a contact.

```
# SamuraiCRM/engines/tasks/app/overrides/add_tasks_to_contact.rb
if Samurai::Core.available?(:contacts)
  Deface::Override.new(:virtual_path => "samurai/contacts/contacts/show",
    :name => "add_tasks_to_contact",
    :insert_after => "[data-samurai-hook='contacts_show']",
    :partial => "overrides/contact_tasks_list",
    :namespaced => true)
end
```

Step 8: Add an override view

```
<!-- tasks/app/views/samurai/tasks/overrides/_contact_tasks_list.html.erb -->
<hr>
<div class="panel panel-primary">
  <div class="panel-heading">
    Tasks for this contact
```

```

</div>
<table class="table">
  <thead>
    <th>ID</th>
    <th>Title</th>
    <th>Content</th>
    <th>Created On</th>
    <th></th>
  </thead>
  <tbody>
    <% @contact.tasks.each do |task| %>
      <tr>
        <td><%= task.id %></td>
        <td><%= task.title %></td>
        <td><%= task.content %></td>
        <td><%= task.created_at.strftime("%d %b. %Y") %></td>
        <td>
          <%= link_to 'Show', [samurai, task], class: 'btn btn-primary' %>
          <%= link_to 'Edit', samurai.edit_task_path(task),
            class: 'btn btn-primary' %>
          <%= link_to 'Destroy', [samurai, task],
            class: 'btn btn-primary', method: :delete,
            data: { confirm: 'Are you sure?' } %>
        </td>
      </tr>
    <% end %>
  </tbody>
</table>
</div>

```

Step 9: Take a look

Let's see how it looks like.

Figure 6.5

SamuraiCRM
Home
Tasks
Contacts
My Account
Admin
Logout

Thibault

First Name: Thibault
Last Name: Denizet
Company:
Email: bo@samurails.com
Phone:

Edit
Back

Tasks for this contact

ID	Title	Content	Contact	Created On	
1	Call 000	Make a call to 000 to get some updates.	bo@samurails.com	12 Feb. 2015	Show Edit Destroy

Figure 6.5: Sweet!

Step 10: Remove the Tasks engine and take a look

But what happens if we remove the **Tasks** engine? Let's try.

Comment out `gem samurai_tasks, path: 'engines/tasks'` in the parent gemfile:

```
gem 'samurai_core', path: 'engines/core'  
gem 'samurai_contacts', path: 'engines/contacts'  
#gem 'samurai_tasks', path: 'engines/tasks'
```

`bundle install`, restart your server and reload the page.

[Figure 6.6](#)

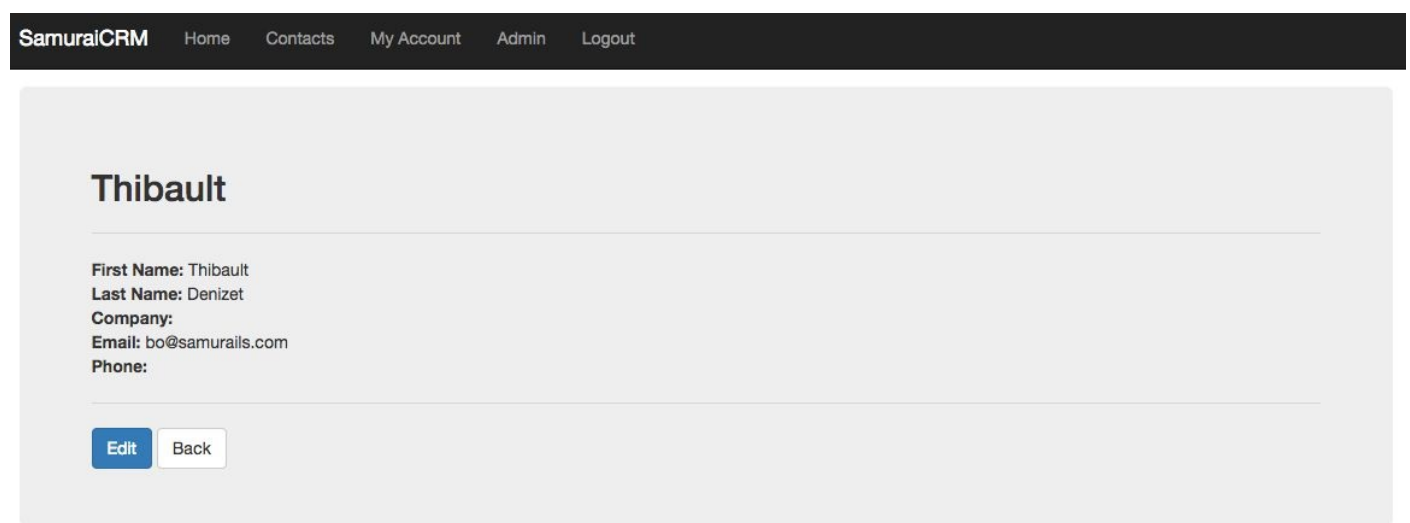


Figure 6.6: Still working and no tasks, great!

Step 11: Remove the Contacts engine and take a look

Let's try to remove the **Contacts** engine now.

```
gem 'samurai_core', path: 'engines/core'  
#gem 'samurai_contacts', path: 'engines/contacts'  
gem 'samurai_tasks', path: 'engines/tasks'
```

`bundle install`, restart your server and reload the page.

[Figure 6.7](#)

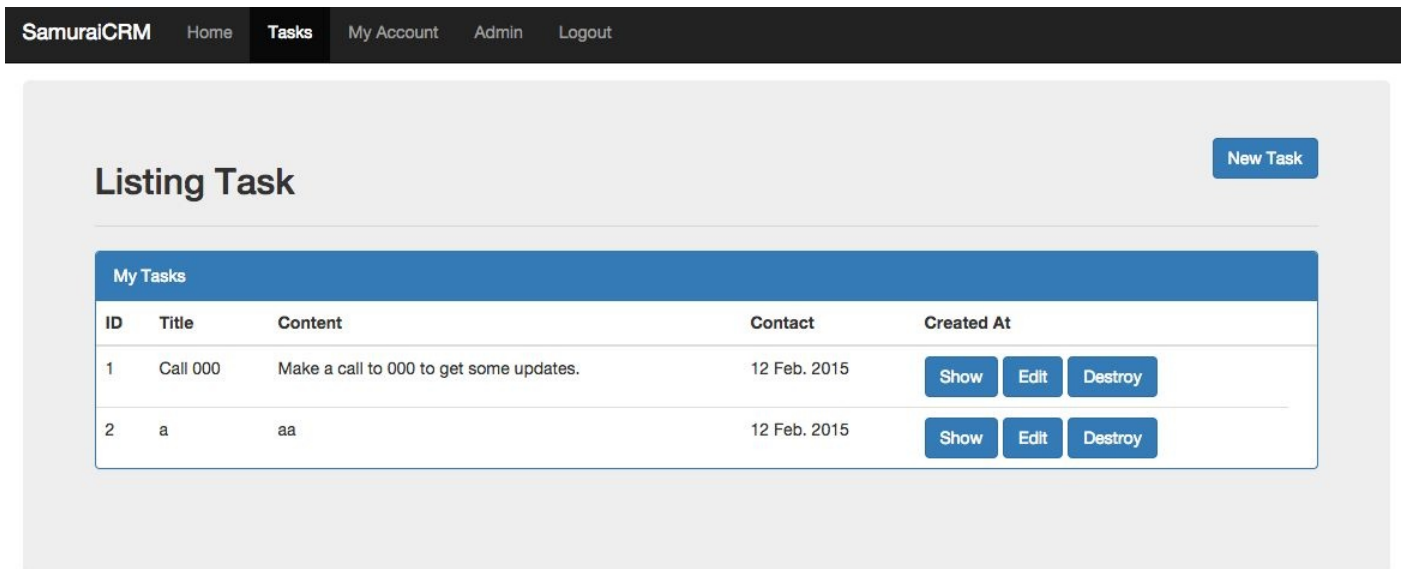


Figure 6.7: Still working and no contacts, great!

Step 12: Put back everything and relax.

Before we continue, let's uncomment all our modules, re-run **bundle install** and restart our server one more time.

6.2.6 Part 6: Extend the dashboard

Well that's it! You know everything about extending and working with different engines. Let's wrap up this chapter by adding a list of tasks to the dashboard view.

Step 1: Add an override

The Deface override to plug some code in the dashboard:

```
# SamuraiCRM/engines/tasks/app/overrides/add_tasks_list_to_dashboard.rb
Deface::Override.new(:virtual_path => "samurai/dashboard/index",
  :name => "add_tasks_list_to_dashboard",
  :insert_after => "[data-samurai-hook='dashboard']",
  :partial => "overrides/dashboard_tasks_list",
  :namespaced => true)
```

Step 2: Add an override view

And the associated override view which is basically a list of the last 3 tasks created.

```
<!-- tasks/app/views/samurai/tasks/overrides/_dashboard_tasks_list.html.erb -->
<div class="col-md-6">
  <div class="panel panel-primary">
    <div class="panel-heading">
      Last 3 Tasks
    </div>
    <table class="table">
      <thead>
        <th>ID</th>
        <th>Title</th>
        <th>Content</th>
        <th>Created On</th>
      </thead>
      <tbody>
        <%= current_user.tasks.limit(3).each do |task| %>
          <tr>
            <td><%= task.id %></td>
            <td><%= task.title %></td>
            <td><%= task.content %></td>
            <td><%= task.created_at.strftime("%d %b. %Y") %></td>
          </tr>
        <% end %>
      </tbody>
    </table>
    <div class="panel-body text-center">
      <%= link_to '...', tasks_path %>
    </div>
  </div>
</div>
```

Step 3: Take a quick look

Let's see if it worked. [Figure 6.8](#)

SamuraiCRM

HomeTasksContactsMy AccountAdminLogout

Dashboard

Last 3 Tasks

ID	Title	Content	Contact	Created On
1	Call 000	Make a call to 000 to get some updates.	12 Feb. 2015	
2	a	aa	12 Feb. 2015	
...				

Last 3 Contacts

ID	Name	Email	Created On
2	Thibault Thibault	bo@samurails.com	11 Feb. 2015
...			

Figure 6.8: It looks fine to me, nice!

6.2.7 Part 7: Extending the abilities

We saw in the previous chapter how to extend the **Ability** file provided by **CanCan**. We can super easily add a new decorator for the permissions related to tasks.

Step 1: Create an AbilityDecorator

```
# SamuraiCRM/engines/tasks/app/decorators/models/ability_decorator.rb
require 'cancan'

module Samurai
  module Tasks
    class AbilityDecorator
      include CanCan::Ability

      def initialize(user)
        unless user.admin?
          can :manage, Samurai::Tasks::Task, user_id: user.id
        end
      end
    end
  end
end

# Registers the defined abilities.
Samurai::Ability.register_ability(Samurai::Tasks::AbilityDecorator)
```

Step 2: There is no step 2!

(But you can try to access a task that does not belong to your current user.)

6.2.8 Part 8: Do It Yourself! Extend the admin panel

Ah! Your favorite part. The one when I can rest and you get to work. I really recommend doing the following exercises to be sure you understand everything. It's by doing that you will learn. Plus, it's the same thing that you did in the previous chapter. Yes, it's a bit repetitive, but that's how you will get it and be able to work on your own engines.

You will find the steps below as well as a few screenshots.

Step 1: Create an admin controller for tasks, the index view and the routes

Step 2: Add override for Tasks link in Admin Nav

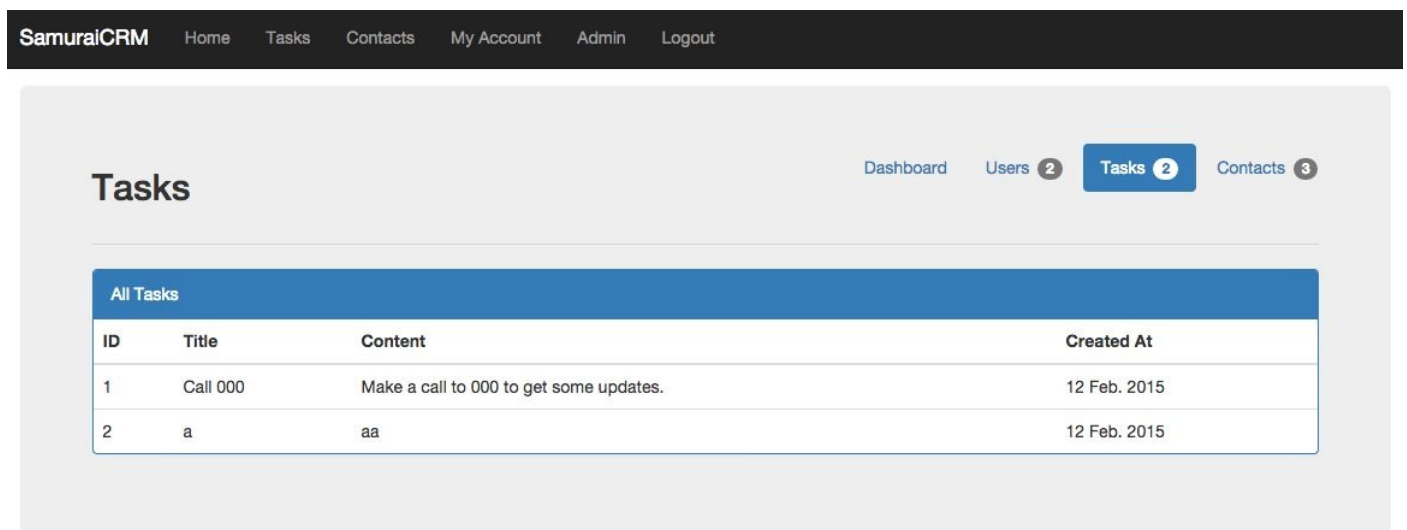
Step 3: Add override view with Tasks link in Admin

Step 4: Add override for Admin Dashboard

Step 5: Add override view with Tasks list for Admin Dashboard

Step 6 (Bonus): Create a decorator for the AdminController

[Figure 6.9](#)



The screenshot shows the SamuraiCRM Admin Dashboard. At the top is a dark navigation bar with links: Home, Tasks, Contacts, My Account, Admin, and Logout. Below this is a light gray dashboard area. On the left, the word 'Tasks' is displayed. On the right, there are navigation tabs: Dashboard, Users (2), Tasks (2), and Contacts (3). The 'Tasks' tab is selected. Below the tabs is a table titled 'All Tasks' with the following data:

ID	Title	Content	Created At
1	Call 000	Make a call to 000 to get some updates.	12 Feb. 2015
2	a	aa	12 Feb. 2015

Figure 6.9: The Tasks list in the admin panel

[Figure 6.10](#)

Admin

Dashboard

Users **2**

Tasks **2**

Contacts **3**

Last 3 Users

ID	Email	Signed Up
2	test@mail.com	11 Feb. 2015
1	bo@samurails.com	11 Feb. 2015
...		

Last 3 Tasks

ID	Title	Content	Created At
1	Call 000	Make a call to 000 to get some updates.	12 Feb. 2015
2	a	aa	12 Feb. 2015
...			

Last 3 Contacts

ID	Name	Email	Created On
1	Thibault Denizet	bo@samurails.com	11 Feb. 2015
2	Thibault Denizet	bo@samurails.com	11 Feb. 2015
3	a b	a@mail.co	11 Feb. 2015
...			

Figure 6.10: The Tasks summary in the Admin Dashboard

6.3 Round Up

And one more chapter done! Let's see what we learned.

6.3.1 What did we learn?

In this chapter, we learned how to extend an engine without defining a dependency on it. We also did more module extensions with Deface and `class_eval`.

First, we created the `Task` model, added a controller and the corresponding views. We also extend the `Core` to add some logic related to `Tasks` to dashboard and to the navigation menu.

Then we extended the `Contacts` module to show a list of tasks for each contact.

Finally, we added one more `AbilityDecorator` for the permissions related to tasks.

6.3.2 Main points to remember

- You can check if an engine is present by checking if the module is defined with `defined?`.
- Creating overrides get easier and faster.
- Creating engines gets boring.

6.3.3 Next Step

We're not going to build more engines in this book, that was the last one. It's getting boring and you already know how to do it. In the next chapter, we're going to see how to work with your engines and how to package them as gem.

Chapter 7

From development to production

In the last chapters, you have built a complete modular Rails application.

Congratulations!

SamuraiCRM is only composed of three modules but I hope you learned a lot while building them. Since we built a ‘web’ application, it should be available online, right? There are a few steps before we can actually push our application to staging/production. Before we do that, I also want to show you how to work with your modular application and how to maintain it. That’s exactly what we’re going to see in this chapter.

The first thing we will do in this chapter is learn how to work with a growing set of engines. Indeed, we will see how to deal with the source code and how to manage your modules as gems. Finally, we will push SamuraiCRM to Heroku.

7.1 Packaging your engines

Until now, we've kept the source code of our modules inside the parent application. It was fine because we were creating our first engines and we didn't care about distribution yet. But now, we want our modules to be easily reused and shared between different applications. Let's see how we can do it.

7.1.1 What's an engine?

We said earlier that an engine is a 'fat free' Ruby on Rails application. But it also got a **gemspec**. That means we can package it as a gem! By encapsulating the source code of each of our module inside a gem, it becomes super easy to share them!

First, we would have to push all our packaged modules to a gem server, like Rubygems for example. Then, all we would have to do is put them inside an application **Gemfile** and run **bundle install**. That would be awesome! That also means we need to version our engines. We need to give each module a version like **1.0.0**, then **1.0.1** when we change something and so on.

7.1.2 Why package your engines as gems

There are a lot of reasons for packaging your engines as gems. First of all, when building a modular application, you actually don't have that many choices. To manage your module source code, either you package them as gems, or you use **git subtree**.

Honestly, **git subtree** is a bit of a pain to work with. I wouldn't recommend it. We actually tried it for a while but it was unproductive and boring. Using gems and simply changing a version number is so much easier as we will see soon.

Here are the main reasons to encapsulate your engines as gems:

- Gems can be pushed to a gems server and be easily loaded from anywhere.
- Can be loaded in your application with **Bundle**.
- Can be versioned.

7.1.3 Make your first module gem

Let's package one of our module as an engine to see how easy it is. Navigate inside the Tasks module folder and run the following command.

```
gem build samurai_tasks.gemspec
```

And it will generate you a new file named **samurai_tasks-0.0.1.gem**. That's it, we got ourselves a gem!

7.1.4 Adding the source

If you now want to make your gem public and available to everyone, you can just run `gem push samurai_tasks-0.0.1.gem`. It's probably not going to work if someone else pushed a gem with the same name and version so you should try with your own modules!

The version is still **0.0.1** since we didn't give it a proper version yet. We will do that in the next part after we've talked about how you should version your engines.

7.2 Managing your engines

When you work with one or more modular applications, your coding workflow changes quite a bit. It gives you some advantages but also comes with annoying inconvenient.

7.2.1 Development vs Production application

When working on a modular application, you will need two ‘versions’ of the same application. We’ll do it for SamuraiCRM at the end of this chapter, but for now, let’s talk theory. We’re going to use SamuraiCRM as an example.

Our current SamuraiCRM application is the development version. It contains the source code of each engine and you will need to pull the source code of any new module in the **engines/engine_name** folder. That’s the copy you’ll work with when you want to add new features.

If I change something in the **core** module, I will commit my changes and push the **core** module from this application. Once the code is pushed, we will be able to generate a gem.

With this new gem pushed somewhere, we’ll switch to the production application. If you don’t have one yet, you just need to pull the code from the development version. Then, you’ll have to delete the engines folder, update the **Gemfile** to use the gems instead of the local folders and **bundle install**. With this, you got a clean production application that you can push to a new repository.

Here are the mains points to remember about development vs production applications:

Development Application

- Used for development.
- Contains the source codes of all the engines (with each engine being pushed to its own repository).
- Cannot be push to production.

Production Application

- Engines are referenced by path in the **Gemfile**.
- Cannot be used for development.

- Does not contain the engines source code.
- Can be pushed to production.
- Engines are referenced by their gems version and source and loaded with Bundle.

7.2.2 Working with Git

Git is an awesome tool to keep your source code versioned and work with other people. If you want to work on a modular application and keep it smooth, you need to have your engines source code versioned. Using **Git** (or any **VCS**) will allow you to easily work on your modules with your colleagues and safely add new features by using branches!

But what's the best approach? Should you just version the parent application with the source code of the modules? That's one solution, but in my humble opinion, not the best one.

I recommend versioning your parent application of course, but without the modules inside. If you use **Git** (`git init .`) inside each engine, they won't be versioned with the parent app. You will have to push the parent application and each module to their own repository but it will give you a log for each module instead of a confusing log only for the parent application.

7.2.3 Where should you push your source code

- [Github](#)

Github is a great platform to store your source code. As long as you're doing open source projects, everything will be free. If you need private repositories however, it might not be the optimal solution. Pricing on GitHub is based on the number of repositories that you have and with modular applications, you will need a lot of repositories: one for each module and for each parent application. It can get pricey!

- [Bitbucket](#)

Bitbucket offers a different pricing strategy: you pay for the number of users and not the number of repositories. If you work alone, this is your best option because you won't have to pay anything! Indeed, it's free for up to 5 users. That's a great solution for

modular applications.

- Your Own Server: [Digital Ocean](#) + [Gitlab \(optional\)](#)

If you have access to any kind of server, you can configure your own git repositories. It won't cost you anything except for the server price. You can even install some web applications to access Bitbucket/Github-like interfaces. If you're looking for this kind of thing, checkout Digital Ocean + Gitlab!

7.2.4 How to use branches

Branches are an awesome feature of **git** which makes working on different features at the same time much easier. Working on modules is not much different than working on a regular Rails application.

You want to create a new branch for a module anytime you have to change something. The code should only be merged into Master once it's been tested and is ready to go live.

When you're working on massive changes to your modular application, I recommend creating a new branch on all your engines and on the parent application with the same name. It's easy to know what you're working on and branches can easily be merged on each module.

7.2.5 Versioning your modules

I recommend following the good practices described at semver.org to version your modules. Here's the summary but you should definitely take a look at the website semver.org.

Box 7.1. Semantic Versioning 2.0.0

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner,
and
- PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

When you want to test a packaged module, you should add the branch name to the version number:

MAJOR.MINOR.PATCH.BRANCH

1.0.0 will become **1.0.0.branch_name**. This should mainly be used to test gems in a staging environment.

7.2.6 Eating your own dog food

Have you ever heard the saying ‘Eating your own dog food’? If you don’t know it, it means using the tools you create to ensure their quality and capabilities.

Let’s say you’re working on a CMS, like WordPress. Obviously, you want to write it in Ruby. One of the most interesting feature of WordPress is how extendable it is. You can easily create a plugin by hooking into WordPress. Of course you want your CMS to be as famous as WordPress, so having a plugin-ready application is a big plus. With everything you’ve learned in this book, you now have two choices.

Either you create a regular Ruby on Rails application. When it’s done, you start to think about how people could extend your application. Basically, you’re not eating your own dog food because you won’t be using this extension system yourself. And everybody loves dog food!

The other option is to build a modular application the way I showed you. You’re making your application extendable for yourself so you can detect the flaws and fix them before you let other people create plugins. You could create the **Core** as a super basic system and create your own modules to add more features like comments, tags, and so on. You can even make it so people can disable some feature super easily by not loading the engines! Next time you’re building an application that has to be extendable, think about it, modularity might be the way to go.

7.3 Publishing your engines

Now that your engines are packaged as gems, they can easily be loaded in your application by using bundle. But if we want Bundle to find them, they need to be available from somewhere online!

How do we do that?

7.3.1 Public engines vs Private engines

To push your gems online, you have two solutions. It really depends on what kind of work you are doing and if you are working on an open-source engine or on an engine that's supposed to stay private.

7.3.2 Public? Rubygems!

If you want to share your engine with the world, your life will be super easy. That's what the karma is all about right? If you are ready to share it, you can just use Rubygems. That's it.

Rubygems stores all gems for free and you can put yours up there too. Anyone will be able to use it and see your code so if that's not a problem for you, go for it!

Unfortunately, it's not always possible. Even if you want to, you might not have the right to share your code. The corporate world doesn't like it when you share their secrets! That's understandable so we need another solution, but don't expect it to be free (It's fine, the company will pay...).

7.3.3 Private? Pick one

Lucky us, if Rubygems can do it, why can't we? Other people have thought about this too. I guess that's how Gemfury was born.

- Gemfury

Gemfury gives you a gem server where you can store private and public gems. You control if you want this gem to be public and this one to stay private. That's up to you! Unfortunately, Gemfury is not free. Their plans are interesting except if you plan on pushing lots of gems.

It's probably not the best solution for a modular application!

- Make Your Own

So if Gemfury is too pricey for you, how about making our own? It will have a cost too. But it's going to be a small and flat fee. The only thing we need to create our own gem server is a server. VPS, dedicated, it doesn't matter. Any cheap provider like Digital Ocean or Linode will do. The server will just store packaged codes and send them when someone run the bundle install command. Nothing too intense. All we need is for it to be accessible remotely.

Got a server? Good! But we still have to set up a gem server. The good news is that some people have created nice gems that we can use to set up a gem server. I'm going to show you right now how to build a private gem server with Gem in a box on a \$5 Digital Ocean Droplet. You can get a server and follow the steps but honestly, it's not that hard so reading should be enough to understand how it works.

You can use any server that you have access to. If you want to get your own Digital Ocean Droplet, you can [sign up here](#). This is an affiliate link and you will get a credit of \$10 if you use it. That means you can try building your private gem server for free! (I will also get a commission if you become a regular user.)

Let's get started!

7.4 Build a private home for your engines

7.4.1 Setting up Geminabox

Thanks to this neat little gem, Geminabox, we can very easily create a simple Gem Server. The following commands will be run from a Digital Ocean Droplet. I will use RVM to install Ruby. I will put the gems web application in `/var/www/gemserver` and store the actual gems in `/var/thibault/gems/` because this is my personal server and nobody else has access to it.

Step 1: SSH to your server

Step 2: Download RVM and Ruby

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

Step 3: Wait. A. Long. Time.

Step 4: Load RVM

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

Step 5: Download RVM dependencies

```
rvm requirements
```

Step 6: Install Geminabox

```
gem install geminabox --no-doc
```


Step 7: Create a folder for the gems application

```
mkdir /var/www/gemserver
```

Step 8: Create a folder to store the gems

```
mkdir home/your_folder/gems
```

Step 9: Put the following code in /var/www/gemserver/config.ru

Change `a_very_secret_key` to something very secret.

```
require "rubygems"
require "geminabox"

Geminabox.data = "/home/your_folder/gems" # ... or wherever

use Rack::Auth::Basic, "Samurails Secret Gem Server" do |username, password|
  username == 'samurails' && 'a_very_secret_key' == password
end

run Geminabox::Server
```

If you want to keep your server open, you can remove HTTP authentication by removing these lines:

```
use Rack::Auth::Basic, "Samurails Secret Gem Server" do |username, password|
  username == 'samurails' && 'ao9836j5cGI0hC4dH4qz9EmIFBEVE5GX' == password
end
```

Step 10: Fix the permissions.

```
sudo chown -R www-data:www-data /var/www/gemserver
```

Step 11: Install a web server. I like Puma.

```
gem install puma
```

Step 12: Start Puma as a daemon

```
puma --port 9292 config.ru -d
```

Step 13: Access our gem server

And that's it, we've got ourselves a private gem server! [Figure 7.1](#)



Figure 7.1

7.4.2 Pushing gems to your private gem server

So how do we push gems now? Let's push the **Core** module to see how it works. The following commands can be run from your local computer.

First, navigate to **SamuraiCRM/engines/core**.

Step 1: Encapsulate the Core in a gem

```
gem build samurai_core.gemspec
```

Step 2: Configure Geminabox

Set the remote host for Geminabox.

```
gem inabox -c
```

Step 3: Push the gem

```
gem inabox samurai_core-0.0.1.gem
```

Step 4: Remove the local gem

```
rm samurai_core-0.0.1.gem
```

And we got our gem on the Gems Server. [Figure 7.2](#)

404 to get /quick/Marshal.4.8/
xxxx-0.4.2.gemspec.rz · Issue #2 ·
geminabox/geminabox

Gem in a Box

If you did not need any credentials to get to this page

```
gem sources -a http://samurails.com:9292/
```

If you needed some credentials to get to this page

```
gem sources -a http://username:password@samurails.com:9292/
```

and then ...

```
gem install geminabox  
gem inabox [gemfile]
```

Upload Another Gem

a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z

samurai_core (0.0.1)

```
gem install samurai_core -v "0.0.1"
```

Core features of SamuraiCRM.
– Thibault Denizet

Upload Another Gem

Figure 7.2

Step 5: Add your Gem Server to your default sources

Adding the source to the default sources will allow us to query for our gems from

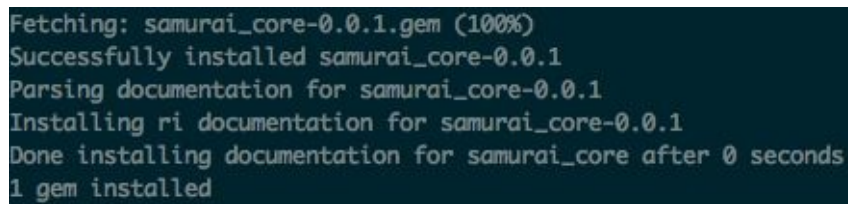
anywhere on your computer.

```
gem sources -a http://samurails:a_very_secret_key@samurails.com:9292/
```

Step 6: Try to install samurai_core

```
gem install samurai_core
```

Output: [Figure 7.3](#)



```
Fetching: samurai_core-0.0.1.gem (100%)  
Successfully installed samurai_core-0.0.1  
Parsing documentation for samurai_core-0.0.1  
Installing ri documentation for samurai_core-0.0.1  
Done installing documentation for samurai_core after 0 seconds  
1 gem installed
```

Figure 7.3

7.4.3 Bundle install your gems

We just added our gem server as a source for the **rubygems** gem. Now, let's see how we can load gems in your Rails application using Bundle.

Step 1: Add your Gem server as a source to your application Gemfile

```
# Gemfile  
source 'https://rubygems.org'  
source "http://samurails:a_very_secret_key@samurails.com:9292"
```

Step 2: Secure your secret key

I recommend taking your secret key out of the **Gemfile** and store it in an environment variable.

```
# Gemfile  
source 'https://rubygems.org'  
source "http://samurails:#{ENV['GEM_SERVER_KEY']}@samurails.com:9292"
```

Step 3: And update your zshrc/bashrc/other (for development purposes):

```
export GEM_SERVER_KEY='a_very_secret_key'
```

Step 4: Reload your config

```
source .zshrc
```

Step 5: Use the new gem

Remove the path option for the Core module inside your Gemfile.

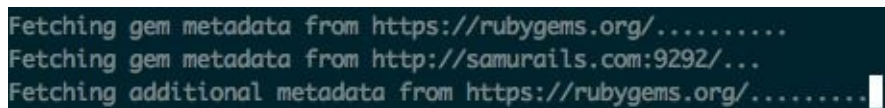
```
# Gemfile  
gem 'samurai_core'
```

Step 6: Run bundle update

```
bundle update
```

Step 7: It works!

[Figure 7.4](#)



```
Fetching gem metadata from https://rubygems.org/.....  
Fetching gem metadata from http://samurails.com:9292/...  
Fetching additional metadata from https://rubygems.org/.....
```

Figure 7.4: Our gem gets loaded, awesome!

7.4.4 Pushing all our modules

Now that we setup our private Gem server, it's time to push our modules! But first, let's update the versions. We're going to update all our modules to **1.0.0** and officially release them! Do the four following steps for each module: **Core**, **Contacts** and **Tasks** and replace {module} with the appropriate module name.

Step 1: Change version

```
# SamuraiCRM/engines/{module}/lib/samurai/{module}/version.rb  
  
def version  
  '1.0.0'  
end
```

Step 2: Create the gem

```
gem build samurai_{module}.gemspec
```

Step 3: Push the gem

```
gem inbox samurai_{module}-1.0.0.gem
```

Step 4: Remove the local gem

```
rm samurai_{module}-1.0.0.gem
```

7.5 Push your application

Our engines are now easily accessible and can be used in any Ruby on Rails application simply by specifying the source and the gem name. That means we can push SamuraiCRM to production! Well, ‘production’ is a big word but you get the idea.

I’m going to push SamuraiCRM to Heroku but feel free to push it wherever you want. The free dyno from Heroku will be more than enough to handle the heavy usage of SamuraiCRM!

Let’s push it!

Step 1: Create an account on Heroku

Step 2: Create a new application on Heroku

I will call mine **samuraicrm**!

Step 3: Duplicate SamuraiCRM

```
cp SamuraiCRM SamuraiCRMProd
```

Step 4: Remove the engines folder.

Navigate inside SamuraiCRMProd and remove the engines folder. We won’t be needing it for the production application.

```
cd SamuraiCRMProd && rm -r engines
```

Step 5: Update the Gemfile

```
source "http://samurails:#{ENV['GEM_SERVER_KEY']}@samurails.com:9292"  
# .. GEMS ...  
gem 'samurai_core', '1.0.0'  
gem 'samurai_contacts', '1.0.0'  
gem 'samurai_tasks', '1.0.0'
```

Step 6: Add Heroku specific gems

Note that you have to move sqlite to the development and test group else Heroku will crash!

```
group :development, :test do
  gem 'sqlite3'
end
group :production do
  gem 'pg'
  gem 'rails_12factor'
end
```

Step 7: Bundle

```
bundle update
```

Step 8: Commit your changes

```
git add .
git commit -m 'Switch to production application'
```

Step 9: Add secret key to our Gem server in heroku config

```
heroku config:set GEM_SERVER_KEY=a_very_secret_key
```

Step 10: Add git remote

```
git remote add heroku git@heroku.com:samuraicrm.git
```

Step 11: Push your application

```
git push heroku master && heroku run rake db:migrate -a samuraicrm
```

Step 12: Enjoy!

[Figure 7.5](#)

Listing Contacts

[New Contact](#)

My Contacts					
ID	First Name	Last Name	Company	Email	Phone
1	Thibault	Denizet		bo@samurails.com	

[Show](#)[Edit](#)[Destroy](#)*Figure 7.5*

7.6 The modular workflow

Before finishing this chapter, I thought it would be interesting for you to see a complete work flow when changing some code inside a modular application. So here it is!

Step 1: Create a new branch

Go inside one of your module inside the development application and create a new branch with:

```
git checkout -b my_branch
```

Step 2: Make your changes

Add this new amazing feature to your application.

Step 3: Run the tests

Always be sure that you didn't make changes that breaks your previous hard word!

Step 4: Add the branch name to your module version

Update the module version by adding the branch name:

```
'1.0.0' -> '1.0.0.my_branch'
```

Step 5: Commit your changes

```
git add .  
git commit -m 'Amazing New Feature'
```

Step 6: Push

Push your changes to the module repository.

```
git push
```

Step 7: Build gem

```
gem build my_module.gemspec
```

Step 8: Push gem and delete it from your computer

```
gem inabox my_module-1.0.0.my_branch.gem  
rm my_module-1.0.0.my_branch.gem
```

Step 9: Create a new branch in the Production application

```
git checkout -b my_branch
```

Step 10: Change gem version

Update the **Gemfile** of the production application to use your beta gem:

```
gem 'my_module', '1.0.0.my_branch'
```

Step 11: Bundle install

```
bundle install
```

Step 12: Run the tests from the production application

Step 13: Commit the changed version in your production application

```
git add .  
git commit -m 'Updated my_module to 1.0.0.my_branch'
```

Step 14: Push to staging

- Heroku

```
git push staging my_branch:master
```

- Capistrano

```
cap staging deploy
```

Step 15: Merge the module changes in the development application

Happy with your new feature? Time to merge the changes then.

```
git checkout master  
git merge my_branch
```

Step 16: Update version

```
'1.0.0.my_branch' -> '1.1.0'
```

Step 17: Push gem

```
gem build my_module.gemspec
```

```
gem inabox my_module.1.0.gem
```

```
rm my_module.1.0.gem
```

Step 18: Update in gemfile in production application

```
gem 'my_module', '1.0.0.my_branch'
```

Step 19: push to production

- Heroku

```
git push production master
```

- Capistrano

```
cap production deploy
```

7.7 Round Up

A lot of theory in this chapter with just enough practice. Let's go over what we learned!

7.7.1 What did we learn?

In this chapter, we learned more about the best ways to work with a modular application. We saw how to manage our modules and how to work on them. We also saw how we could create a private gem server to keep our precious code from the dangerous world. Finally, we pushed SamuraiCRM to Heroku and discovered how easy it was.

7.7.2 Main points to remember

- Following some rules is important to prevent your modular app from becoming a mess.
- If you want to keep your modules private, you'll have more work!
- Gem Servers are not that hard to setup and are quite cheap.
- Pushing a modular application is super easy.

7.7.3 Next Step

There is no next step! I've taught you everything I know about building modular applications. I hope you enjoyed the ride and picked some new stuff on the way!

Chapter 8

The End

You've reached the end of Modular Rails! I hope you learned a lot while reading this book. If you still have questions about modularity, you can send me an email at bo@samurails.com and I will write a blog post to answer you. Let's see everything we've learned!

First of all, you learned that there were other types of architecture than the regular Ruby on Rails application. Don't get me wrong, the way Rails is organized is awesome and you can do a lot of things with it. But sometimes, you need something a bit different. Something more modular.

Don't rush to modularity every time you have to build a new application. You need to carefully weigh the pros and cons before taking a decision. Yes, modular applications are awesome but are hard to maintain and can become a pain in the a**.

Be careful. With great powers come great responsibility.

Now it's your turn! You can start building modular applications. Feel free to reuse my gem `modular_engine`, I will try to maintain it and improve it. Like with everything, you need to practice to learn it. So build some simple modular applications before you attack a real application. Be sure to understand how everything is connected.

If you need help, don't hesitate to contact me at bo@samurails.com. I will be happy to answer you!