Test-Driven Ember.js

Karol Galanciak

Contents

Test Driven Ember.js - The Definitive Guide **3**

Learn How To Write Effective And Maintainable Tests in Ember.js applications . . . 3

Test Driven Ember.js **4**

Preface **5**

Introduction **6**

Why did I write this book? . . . 6
Who Is This Book For? . . . 7
How To Read It . . . 7
Who Am I? . . . 7

Testing Basics **8**

What Is the Point Of Writing Tests? . . . 8
Test-Driven Development . . . 8
Why Is It Important To Write Tests First? . . . 9
Testing From The Outside-In . . . 10
TDD vs. BDD . . . 10
Classification Of Tests . . . 11
Test Pyramid . . . 11
To Test Or Not To Test . . . 12

Essential Tools **14**

QUnit . . . 14
ember-test-helpers & ember-qunit . . . 20
testem . . . 28
ember-cli-htmlbars-inline-precompile . . . 30
pretender / ember-cli-pretender . . . 31
ember-cli-mirage . . . 33
ember-test-selectors . . . 50
ember-cli-page-object . . . 51

Test-Driving Our Application **56**
Our Example Application - Book.me 56
Starting The Development - Generating New App 57
Adding The First Feature - Sign Up And Sign In 57
Adding The Core Feature - Rentals' CRUD And The Calendar 102

Closing Thoughts **196**

Test Driven Ember.js - The Definitive Guide

Learn How To Write Effective And Maintainable Tests in Ember.js applications

Karol Galanciak

Version: 1.0 Release Candidate 1 (31.09.2017)

Test Driven Ember.js

To Napoleon Hill, the author of Think And Grow Rich, who has been one of my greatest inspirations.

In memory of Rich Piana, the founder of 5% Nutrition and bodybuilder, who planted **Whatever It Takes** mentality in my mind.

> "Whatever the mind can conceive and believe the mind can achieve." **- Napoleon Hill**

> "Are you truly doing whatever it takes to reach your goals?" **- Rich Piana**

Preface

Testing is one of the hottest topics when it comes to software development and building ambitious web applications. Most of the available resources about **Test-Driven Development / Test-First** approach for web applications is based on server-side programming. If you come from Ruby on Rails background or from any other technology, which embraces writing tests, you may think that testing your client-side apps should be similar and when it comes to writing good tests, similar rules should apply.

For testing **client-side applications** this is not always true. Have you heard about the concept of test pyramid and the rules/suggestions stating that you should mostly focus on unit testing, add some integration tests on top and have only a small amount of acceptance (end-to-end) tests? These guidelines are certainly useful when it comes to building server-side apps. However, in **Ember.js** the pyramid of testing could literally be inverted and the majority of the focus could be put on acceptance tests!

Testing **Ember.js** or any other **JavaScript** client-side applications is a skill of its own, and the abundance of asynchronous actions in most of the apps and other things that are not common in server-side programming makes it harder to write efficient and maintainable tests. Thanks to Ember community, there are plenty of tools like ember-cli-mirage that make testing faster and simpler; nevertheless, it doesn't mean that those tests are going to be easy and obvious to write.

Test-Driven Development has been a fundamental concept for me since learning Ruby on Rails, and when I was starting my journey with Ember, it wasn't that clear to me how to do it right. Learning that skill took me a lot of time and effort, and I would love to share this knowledge with you. I hope that after reading this book, you will have a clear idea what tools you should use in your own ambitious Ember projects and what kind of tests you should write to make the development process efficient and smooth. It took me over two years to learn how to do it well, and now you have a chance to learn all these concepts in the next few hours ;). Enjoy!

Introduction

Why did I write this book?

Writing a book seemed to me to be a pretty cool idea for a long time. I remember reading a few years ago a short, self-published e-book about Chef and setting up server infrastructure and besides enjoying the content of the booking and learning a lot from it, I found it pretty fascinating that you can publish a book in such a simple way. That was probably the day when I started thinking about writing something more meaningful and having more impact comparing to writing just blog posts.

In less than 3 years later after reading this booking, I became the CTO of BookingSync which I consider as one of my greatest achievements, especially given the fact that I don't have a formal computer science background and that 5 years earlier I had barely known what a "loop" or "conditional" is. I started to analyze my last few years and figuring out how exactly I managed to do that.

It was pretty clear that despite learning myself how to code, I couldn't really say that I did everything myself. Obviously, I had to do the actual work, but the help was ubiquitous - I was able to find answers to a lot of my questions on StackOverflow, I was reading a lot of blogs with great content, and everything was available for free! And certainly, there is something more which deserves a special award: open source software. Thanks to the awesome community behind **Ruby on Rails** and **Ember.js**, I've managed to achieve all those things, and without the help of the community, I certainly would never have a chance to learn Rails or Ember, simply because those frameworks would have never evolved without everyone contributing to them.

It made me think that I partially owe my career to all the awesome people involved in the development process of the languages, frameworks, gems, addons and other tools I've been using from the beginning and to the fact that they gave something back. I've been been contributing to open source software for a while and have been periodically writing articles on my blog, wrote a couple of articles on Ragnarson's blog when I used to work there and also had a chance to write a guest blog post for Code School. I managed to help some developers here and there by own contributions, but I didn't think it was enough compared to how much I'd had benefitted from the open source software. Since I had had a plan to write a book for a long time, I thought it was the right time to do something much more meaningful and truly give back.

One major gap in the Ember ecosystem for a long time has been a lack of information (besides the official docs) how to do the testing in real-world Ember apps and how to do it right. I've been interested in Test-Driven Development for a long time, so that looked like a perfect opportunity to share something that could help other developers in a substantial way.

I hope that by releasing this book for free, I will reach a larger audience comparing to a paid version and "pay" my debt to the open source community.

Who Is This Book For?

Every Ember developer, regardless of the skills, could benefit from this book. If you are a beginner, you will certainly learn more as it may serve as a guide for writing the tests. Nevertheless, if you are an experienced developer who has already written tons of tests, you can still find some useful ideas and techniques about writing tests presented in the book that could improve the way you approach testing or just show a different perspective.

This book assumes a basic knowledge of Ember and popular tools (mostly **ember-cli**). Having some background in testing (not necessarily in JavaScript) will also be beneficial, as this book doesn't really focus on the basics of testing itself, but rather how to do it for Ember applications.

How To Read It

I would suggest reading it from the beginning till the end in chronological order, but feel free to jump to some particular chapter if you *really* want to focus on one particular concept at the moment.

Who Am I?

Hi! I'm Karol Galanciak, CTO of BookingSync. I used to be the Angular fanboy, but after writing my first simple application in Ember in the fall of 2014, I immediately fell love with the framework. I had thought Angular was great, but even back then Ember was on a different level, especially thanks to Ember Data. That is how my journey with Ember has begun.

Privately, I love lifting weights and getting pumped, playing some heavy riffs and technical shreds on guitar, scuba diving and a traveling to cool places. I have a special gift of turning yerba mate (especially Pajarito!) into code. Apparently, I'm a capsaicin addict and cannot eat anything that is not hot.

Testing Basics

What Is the Point Of Writing Tests?

Before starting writing tests, we should ask ourselves the fundamental questions: what's the point of doing that? How are we going to benefit from writing tests? It's a complicated and time-consuming process, do the benefits outweigh the costs?

The essential benefit of writing tests is a confidence that the code we wrote works. The only alternative to writing tests is "testing" manually (e.g., by clicking through the different scenarios using the browser) which requires a lot of effort as well and doing it over and over again will eventually take much more time than writing automated tests.

It is not only when implementing new features that we need to verify if our application works fine - having well-written and the comprehensive test suite is also crucial when modifying existing behavior and doing refactoring. Can you imagine testing manually every possible scenario after applying every minor change? Neither can I. That way we get a powerful anti-regression tool which provides instant feedback. We've modified some code, and we have failing tests? Awesome! We know from the very beginning that *something* is not right and we can quickly fix it.

Another important aspect of writing automated tests is the ease of finding bugs. Imagine that you are maintaining a huge application and the users start complaining that some feature doesn't work as it should. How fast can you identify the culprit without test suite? It would certainly take a lot of time. With well-written tests, it would take much less time to find where the problem is. Not only do you have a feedback from higher-level acceptance tests that are interacting with real UI that *something* is not right, but also the unit tests that deal with much smaller scope will tell you what the issue is.

An extra advantage of writing automated tests (mostly applicable when doing **TDD**) is clarifying the intentions - you need to precisely define the consecutive steps to execute some logic, which helps to break the functionality into smaller, more manageable pieces. That way you easily become more productive and focused on the current task.

Thanks to having automated test-suite we come more productive and confident about our code which makes any changes easier and indirectly, by investing time in writing good tests, we save time on arduous debugging.

Test-Driven Development

You've most likely heard the term **Test-Driven Development** or **Test-First Approach**. What do they mean and how they are different?

According to Growing Object Oriented-Software Guided By Tests by Steve Freeman and Nat Pryce, a classic book about TDD, the idea of it is to *"write the tests for your code before writing the code itself"*. Well, that seems very similar to **Test-First Approach**, which is also about writing the tests before the code. The difference between those is subtle, and quite often those terms are used interchangeably.

Test-First Approach is about starting with a test and writing a minimal amount of code to make the test(s) pass. It is also the case in **TDD**. However, the important aspect of **Test-Driven Development** is **refactoring** phase, which helps with achieving a proper design - that way **TDD** also serves as a design tool, which doesn't matter that much in case of **Test-First Approach**.

To put it simply:

```
Test-First Approach + Refactoring => Test-Driven Development
```

Why Is It Important To Write Tests First?

Now that we know that writing tests is pretty much essential in the long-term perspective, we should answer one more question: why does it even matter to write tests first? Why can't we just write the code and then add a couple of tests to verify if the features work as expected?

There are few reasons behind it. The fundamental one is that you can't be really sure that the test indeed works unless you've seen it failed. It is quite easy to have a false-positive in a test - the situation where the functionality doesn't work as intended, yet the tests are still passing. It might be just a coincidence (maybe something returned `undefined` but not because such value was supposed to be returned, but because it's never been defined in the first place!). Maybe you misused some testing library's feature which resulted in having such problem. You can still work around those issues and add tests later which happens quite often when working on the legacy apps. Nevertheless, it's simply safer, and it just takes less time to write the tests first without trying to apply some hacks later.

The other reason in that the tests may guide the design. When writing the implementation code, you're probably not focusing on the ease of testing as the primary thing, but rather the ease of writing the code itself. However, the easier the code is to test, the easier it is to use. It quite often results in better interfaces and overall design with less coupling between objects and having more cohesive units. Writing tests first doesn't necessarily guarantee that the code will always be well-designed; nevertheless, it's more likely to be the case than when writing the tests after the implementation.

Keep in mind that those rules are not hard laws or dogmas. Sometimes you may be implementing a feature that you've implemented already few times before or maybe something is just trivial, and you already have an idea about the entire design. Tests are for making out lives as developers easier. If you think there is no risk of writing the code first, then go ahead. I just wouldn't suggest making it a default approach and would rather stay with writing the tests first unless I'm 100% sure that I can do otherwise.

Also, when thinking about the tests as a tool for guiding the design, it's pretty convenient to do the testing from the outside in.

Testing From The Outside-In

Testing from the outside in simply means starting from the higher-level tests and working from that point to lower-level tests.

Usually, it means writing a failing acceptance test for a particular feature, then writing some integration tests (in Ember applications those will be mostly components' integration tests) and ending with some unit tests for models, services, etc. It also means that there might be a reverse order of passing tests - the first passing test could be a model unit test, then component integration test and the last passing one could be the acceptance test that was written first.

The opposite approach to outside-in is the inside-out approach where you focus on isolated units and defer the decision how they are going to interact with each other. This strategy isn't necessarily wrong though - sometimes you may not know how to test given feature from the higher-level perspective. This might happen when you are implementing non-trivial functionality with some complex behavior - one example would be drag and drop. In such case, you may prefer starting with some lower level test (maybe with an integration one or even unit tests) and get back to acceptance tests later. You may lose some benefits of full TDD approach like the tests guiding your design, but still, there is a place for the outside-in approach. However, I believe by default you should always start with the outside-in approach and only try the inside-out approach if the former doesn't work well in particular use case.

TDD vs. BDD

So far I haven't mentioned anything about **Behavior-Driven Development (BDD)** - a term which seems to be used quite often and sometimes even as the opposite approach to TDD.

Nevertheless, I believe there is no any major difference between those two. BDD advocates starting development by writing acceptance tests first, focusing on behavior. But this is the same as TDD done with the outside-in approach! Also, Growing Object Oriented-Software Guided By Tests, which is a classic on TDD, clearly says that starting with end-to-end (acceptance) tests is essential and that one should focus on describing the behavior, not the API of the objects.

Just to keep things simple and not introduce any confusion, I'll stick to **TDD** term later in the book, which might mean the same thing as **BDD** if you are more used to that phrase.

Classification Of Tests

Based on the scope of tests and a number of layers of the application they involve, there are three levels of tests:

- Acceptance tests (end-to-end tests) - these tests cover all layers of the application and simulate user interaction, e.g., by filling forms and clicking the buttons in the browser. They are automated equivalent of manual testing by interacting with the application yourself.
- Integration tests - these tests involve multiple layers of an application, but they don't run the entire app. In context of Ember applications, those are mostly component tests which also involve components' collaborators (like injected services) or isolated and simplified user interaction (e.g., rendering and submitting a form from the component, but outside the context of the rest of the application)
- Unit tests - they are checking if the behavior of a particular unit is right. Note that it isn't necessarily about one object - a unit can be composed of multiple objects, but some of them they might be implementation details, or they don't exist outside given context.

Test Pyramid

If you are a seasoned developer you've most likely heard about the concept of Test Pyramid which illustrates the idea that unit tests should be the fundament of your test suite on top of which you should have some integration tests and few acceptance tests, which will verify if the app truly works.

In case of server-side applications (or in general non-client-side applications) this makes a lot of sense - you don't want your UI interactions to make the majority of your tests as it's quite tricky to test all the edge cases that way and such tests are simply slow comparing to the unit or even integration tests. However, when it comes to Single Page Applications, it's quite the opposite - the entire application serves the purpose of the UI! The priorities are different, so is the test pyramid. In Ember apps (or any other SPAs) the acceptance tests may not necessarily make the majority of your tests (well, at least in terms of the amount, they are most likely going to take most of the time when running the entire test suite though), but as you will see in the next chapters the unit tests don't matter that much as you may initially think.

Some layers like routes and their hooks might not even be worth testing with unit tests at all as the acceptance tests will cover them as well. Unit testing actions in components also might not bring much value - perhaps the logic inside that action works fine, but you don't know if they are going to be executed properly. It's much safer to test the actions via component integration tests, which will provide us with a feedback not only about the logic itself but also if the component is wired-up properly and if it executes proper actions when handling some event.

However, unit tests remain quite important for testing edge cases and complex computed properties as they require much less overhead and are faster.

Considering the tests' structure in typical Ember application, it's probably no longer valid to talk about "pyramid" of tests. Rather, we may end up with Testing Cube without a clear foundation and a clear peak like in case of a pyramid, but with all three layers (acceptance, integration, and unit) mixed in different proportions and with all of them creating the foundation of solid test-suite.

To Test Or Not To Test

As already stated before, there are some layers or part of those layers that might be not worth unit testing at all. I believe this is mostly true for testing hook methods' behavior in routes like `beforeModel`, `model` or `afterModel`, actions and most of the computed properties in components and arguably controllers. Let's break these three layers down and see what the better way to test them is:

- **routes** - hooks like `beforeModel`, `model` or `afterModel` are in most cases pretty simple and they in general fetch data from the remote API, do some transitions based on some conditions, etc. As those are hook methods, they are strictly connected to the request of the application and executed under specific circumstances, so such methods make a poor unit and it doesn't make much sense to test them in isolation. It's more convenient and safer to test them using acceptance tests as you will make sure that the route behaves properly in real-world interaction. Actions are a different story though. Some actions could be tested via acceptance tests as well, but if you have some complex logic there with multiple edge cases, writing unit tests would certainly help. You can either test all the edge cases in the route action itself or extract the logic to another object an just check if the right method is called on this object and the expected arguments are passed.
- **controllers** - they are said to be replaced eventually by routable components, but until this feature is ready, we need to learn how to test them. Controllers in Ember serve a role of a top-level component for given route. It's not currently possible to totally give up on controllers as they handle

query params and transitions (though the latter can be handled on route-level with route actions). Testing controllers highly depends on the current design of your application and if it's a new controller or some already existing controller which is packed with logic. For new controllers, I would suggest keeping some actions and computed properties to a minimum in the controller and moving actions to routes (that are modifying the data, just like the ol' good Data Down Actions Up paradigm recommends) and moving other actions and computed properties to components. Query params should be tested via acceptance tests as it doesn't make much sense in isolation, they are bound to request cycle. That way we would end up with no unit tests for controllers. For already existing ones, you may approach testing actions exactly in the same way as for route actions - unit test the behavior or delegation. For computed properties though you should write proper unit tests and test all the edge cases unless it's something trivial that is already covered in acceptance tests.

- **components** - most of the tests you should be writing for components should be integration tests unless you have a very specific reason for writing unit tests. The integration tests are pretty simple to set up, and even if you have complex actions, you can just extract the behavior to a service, inject stubbed service in the test and verify if the right method on the service was called with expected arguments. Actions are never the isolated units; they are called as the result of some user interaction so they should be tested that way. Some user interactions might be difficult to simulate in integration tests, and such actions could be potential candidates for unit testing, but by default, it's better to always start with an integration test. And what about the computed properties? Unless they are very complex, you should also stick to integration tests.

All this means that the majority of the unit tests you will be writing will cover models and services.

Essential Tools

There are multiple tools that we'll be using in the next chapters of the book. Some of them will be introduced ad hoc when needed, but for now, let's focus on the essential ones.

QUnit

Introduction

QUnit is the default testing tool for Ember. Even though I used to be a jasmine fanboy before I knew Ember, I quite quickly got used to QUnit and appreciated it for its simplicity. The API is pretty limited, yet it offers everything you may need to write proper tests for your applications. Let's take a closer look how it works.

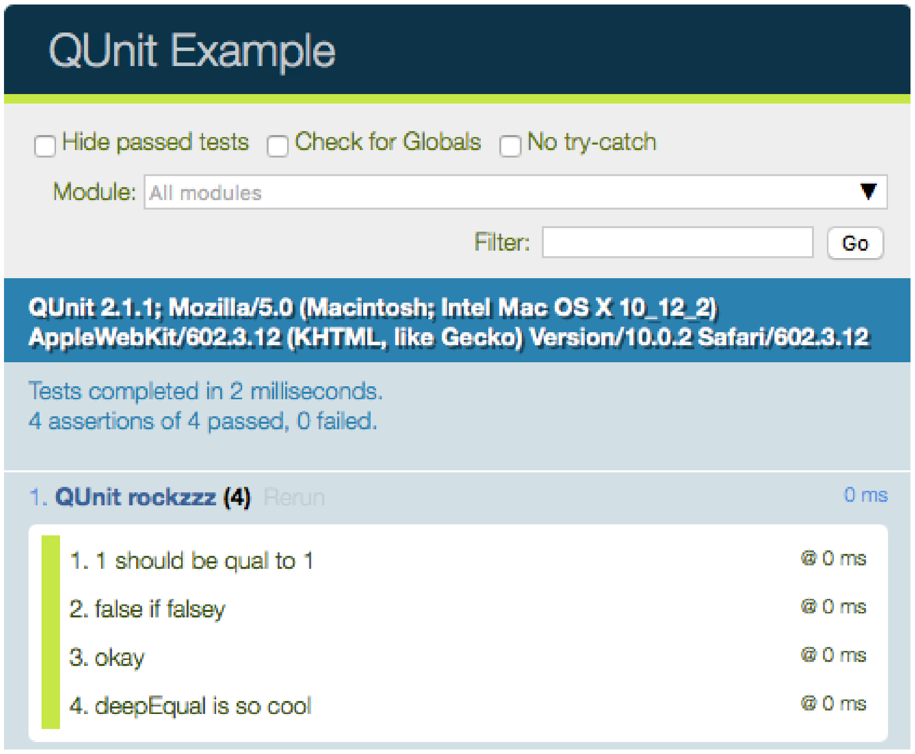
Here's a super short setup for QUnit:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="https://code.jquery.com/qunit/qunit-2.1.1.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script src="https://code.jquery.com/qunit/qunit-2.1.1.js"></script>
  <script src="tests.js"></script>
</body>
</html>
```

What we are doing here is fetching some stylesheets to have a clean and smooth testing page and the QUnit code itself. Let's add some tests to `tests.js` file and see QUnit in action:

```js
QUnit.test("QUnit rockzzz", function(assert) {
  assert.ok(1 === 1, '1 should be equal to 1');
  assert.notOk(false, 'false if falsey');
  assert.equal(1 + 1, 2);
  assert.deepEqual([1, 2, 3], [1, 2, 3], 'deepEqual is so cool');
});
```

After opening the `tests.html` in a browser, we should see something like this:



QUnit Example

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch

Module: All modules

Filter: Go

QUnit 2.1.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12

Tests completed in 2 milliseconds.
4 assertions of 4 passed, 0 failed.

1. **QUnit rockzzz (4)** Rerun 0 ms

```
1. 1 should be qual to 1        @ 0 ms
2. false if falsey              @ 0 ms
3. okay                         @ 0 ms
4. deepEqual is so cool         @ 0 ms
```

Figure 1: Qunit example

API Overview

Assertions

Let's start with explaining the syntax and then we will get back to the details of this tests' UI.

QUnit has a pretty limited API, it's not as flexible and elaborate as Jasmine, which syntax was inspired by powerful Ruby testing framework - RSpec, but it's straightforward and easy to understand. We write test scenarios with `QUnit.test` method, which takes two arguments - the description of the test and callback where we put the test's body. The argument of the callback is `assert` object which contains the assertions. We can use `ok` or `notOk` assertions which check for truthiness or falseness and also assertions such as `equal` which performs non-strict comparison or `deepEqual` which according to the docs performs "a deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates, and function". It is possible also to provide an optional description of particular assertion as the last argument, which helps identify failures of the tests.

These are by no means the only capabilities of QUnit. Dealing with async functions is bread and butter when it comes to JavaScript. To make sure QUnit will wait for an asynchronous operation that is not finished you can use `async()` function. That way QUnit will wait until `done()` is called.

```
QUnit.test('async is awesome', function(assert) {
  const done = assert.async();

  setTimeout(function() {
    assert.ok(true, 'called from async function');
    done();
  }, 100);
});
```

QUnit doesn't provide any API for spying and mocking. Nevertheless, it is still possible to make sure some method or function is called. Just write an assertion within a given function and ensure that the right amount of assertions is called using `expect()`:

```
QUnit.test('testing with assert.expect', function(assert) {
  assert.expect(1);

  var $btn = $('btn');

  $btn.on('click', function() {
    assert.ok(true, 'the button was clicked');
  });

  $body.click();
});
```

If the total amount of assertions is not equal to the `count` argument passed to `expect()`, the tests will fail. That way we can make sure that all the expected assertions were performed.

There are also many other assertions available in QUnit, such as `assert.throws()` for catching the exceptions and I highly recommend to read the official docs for the quick overview, just to be aware of all of them.

Module And Hooks

In QUnit it's possible to group some tests in a `module` for clearer organization purposes, but also for providing hooks for what should happen `before` running tests, `beforeEach` test, `afterEach` test and `after` all tests. Check the following example:

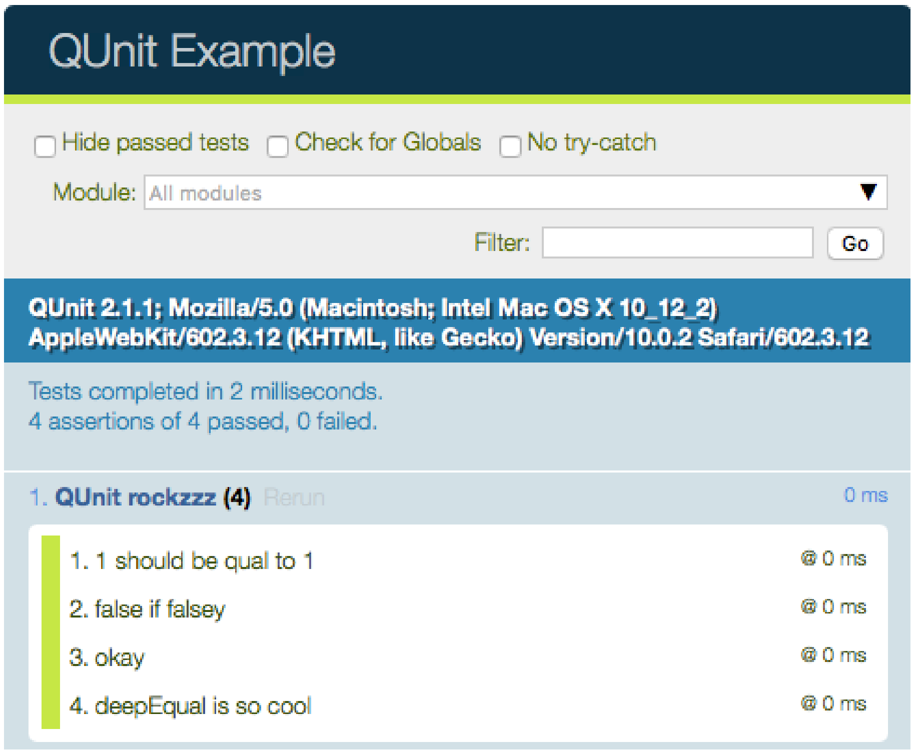
```
QUnit.module('Awesomeness check', {
  before: function() {
    console.log('run me before all tests');
  },
  beforeEach() {
    console.log('run me before each test');
    this.awesomeFramework = 'Ember';
    this.isQUnitAwesome = true;
  },
  afterEach() {
    console.log('run me after each test');
  },
  after() {
    console.log('run me at the end');
  }
});

QUnit.test('Ember should be the awesome framework', function(assert) {
  assert.equal(this.awesomeFramework, 'Ember', 'Ember should be awesome');
});

QUnit.test('QUnit should be awesome', function(assert) {
  assert.ok(this.isQUnitAwesome, 'QUnit should be awesome');
});
```

We grouped some tests under a module with an awesome description and provided some hooks to demonstrate how they work. After running our test suite, all tests should pass, which will mean that the setup from `before` hook and assigning some values to `this` context works as expected and also the sequence of running the hooks is the same as discussed. After opening the console, we should see the following logs:

```
run me before all tests
run me before each test
run me after each test
run me before each test
run me after each test
run me at the end
```

QUnit UI

Let's get back to the cool UI we saw at the beginning of this chapter:


QUnit Example

Hide passed tests, Check for Globals, No try-catch

Module: All modules

Filter: Go

QUnit 2.1.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12

Tests completed in 2 milliseconds.
4 assertions of 4 passed, 0 failed.

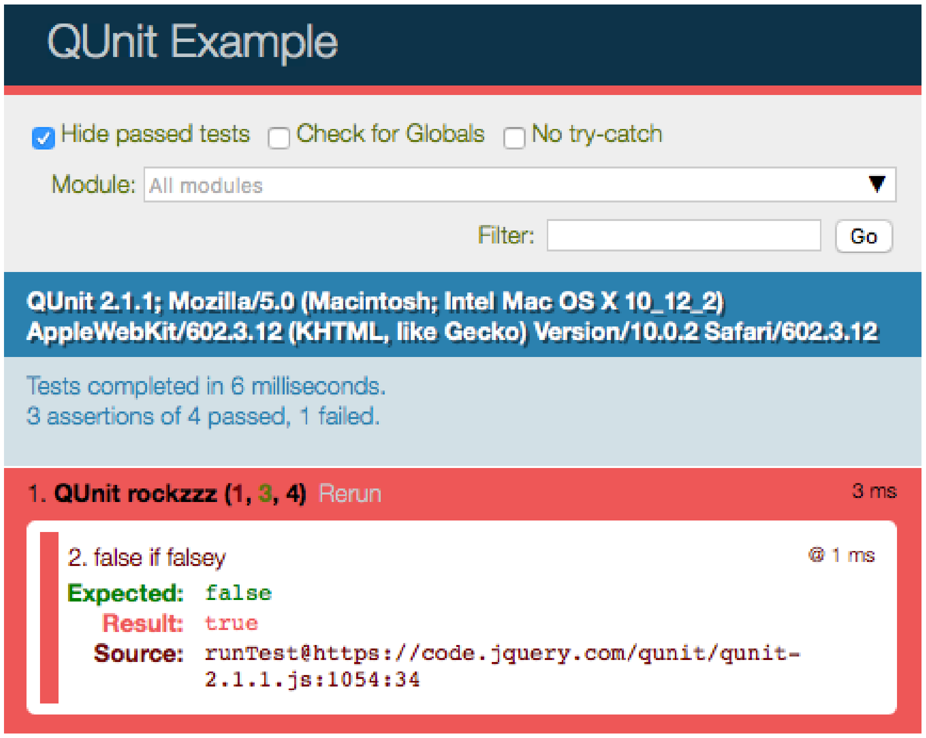
1. QUnit rockzzz (4) Rerun 0 ms

1. 1 should be qual to 1 @ 0 ms
2. false if falsey @ 0 ms
3. okay @ 0 ms
4. deepEqual is so cool @ 0 ms

Figure 2: Qunit example

Besides showing all the tests and their statuses whether they are currently passing or not we have some options for customization:

- Hide passed tests - this options allows to not display the passed tests and only show the failed ones, which is what we want in most cases. If we tried to force some failure from the previous example and checked this checkbox, we would have the following result:



QUnit Example

☑ Hide passed tests ☐ Check for Globals ☐ No try-catch

Module: All modules

Filter: Go

QUnit 2.1.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12

Tests completed in 6 milliseconds.
3 assertions of 4 passed, 1 failed.

1. QUnit rockzzz (1, 3, 4) Rerun 3 ms

2. false if falsey @ 1 ms

```
Expected: false
  Result: true
  Source: runTest@https://code.jquery.com/qunit/qunit-
          2.1.1.js:1054:34
```

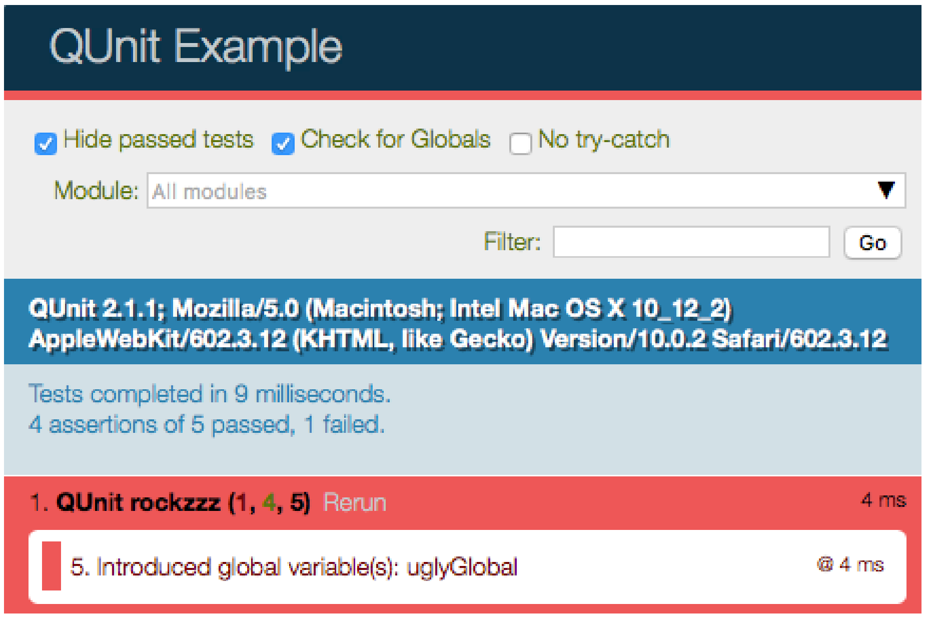
Figure 3: Qunit failure example

- Check for Globals - checking this options will make the QUnit check if some globals were introduced in any of the tests and fail those tests if that's the case. Let's get back to our initial example, introduce some global and take a look at the result:

```
QUnit.test("QUnit rockzzz", function(assert) {
  assert.ok(1 === 1, '1 should be qual to 1');
  assert.notOk(false, 'false if falsey');
  assert.equal(1 + 1, 2);
  assert.deepEqual([1, 2, 3], [1, 2, 3], 'deepEqual is so cool');

  window.uglyGlobal = 'fail the tests!';
});
```

After running the test suite, we will see the following result:

- No try-catch - by default QUnit runs tests inside `try/catch` block, catches all the exceptions and then displays that some error has happened. Some-


QUnit Example
[✓] Hide passed tests [✓] Check for Globals [ ] No try-catch
Module: All modules
Filter:
Go

QUnit 2.1.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_2) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0.2 Safari/602.3.12

Tests completed in 9 milliseconds.
4 assertions of 5 passed, 1 failed.

1. QUnit rockzzz (1, 4, 5) Rerun 4 ms

5. Introduced global variable(s): uglyGlobal @ 4 ms

Figure 4: Qunit Check for Globals example

times you may want QUnit to not catch these errors and get the "native" exception in the console - this is particularly useful when debugging as you will get the exact info where the problem happened.

- Module select - this select lets you pick a particular module that should be the subject of testing, useful especially in huge test suites.
- Filter input - this input allows to filter tests with a description matching the provided phrase, quite beneficial when you need to run only a particular subset of tests.

ember-test-helpers & ember-qunit

ember-test-helpers provide the essential helpers required to test Ember applications. As this library is testing-framework-agnostic it requires some extra integration layer for some of the helpers, in our case it is ember-qunit, but it could also be something different like ember-mocha.

Let's take a look at the provided helpers and how we can use them:

- `TestModule` (exposed as `moduleFor` in ember-qunit) - it's a basic building block for tests handling all the necessary setup and teardown essential to test given subject in the Ember ecosystem. It works for anything that could be looked up by Ember Resolver (models, components, services, etc.), but some of these layers have more dedicated helpers like `moduleForComponent`. Its basic usage is quite straight-forward: you just need to provide the name of the subject with its type:

```
moduleFor('service:current-user');
```

You can also pass an optional description, e.g., "Unit | Service | current -user" and config options, which is the most interesting part here.

With config options, you can pass callbacks such as `beforeEach` and `afterEach`, but also options which influence how the tests are being run.

When writing an integration test, you will need to pass `integration: true` option. And when you are unit testing some piece which depends on other parts of the application, you will need to pass the array of dependencies with `needs` option, like `needs: ['service:current-user']`. Other helpers which extend the behavior of `TestModule` (`moduleFor`) such as `TestModuleForComponent` (`moduleForComponent`) introduce some more specific options, e.g. `unit` flag to indicate that you want to write a component unit test.

You may be wondering why passing the exact subject name is important. Let's take a look at this example of a basic service test:

```
import { moduleFor, test } from 'ember-qunit';

moduleFor('service:current-user', 'Unit | Service | current user');

test('it exists', function(assert) {
  const service = this.subject();

  assert.ok(service);
});
```

Somehow the service we wanted to test was properly initialized using `subject` function. `moduleFor` requires this full name exactly for this purpose - to find a factory and instantiate the object under the test.

Another important thing which `TestModule` (`moduleFor`) handles is contextualization of the tests. Check the following example:

```
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';

moduleFor('service:invoice-price-calculator', 'Unit | Service |
  invoice-price-calculator');

test('it calculates net price for the invoice', function(assert) {
  const calculator = this.subject();

  const service_1 = Ember.Object.create({
    netPrice: 100,
  });
  const service_2 = Ember.Object.create({
    netPrice: 200,
  });
  const service_3 = Ember.Object.create({
    netPrice: 300,
  });
  const services = [service_1, service_2, service_3];

  assert.equal(calculator.calculateNetPrice(services), 600);
});

test('it calculates gross price for the invoice', function(assert) {
  const calculator = this.subject();

  const service_1 = Ember.Object.create({
    netPrice: 100,
  });
  const service_2 = Ember.Object.create({
    netPrice: 200,
  });
  const service_3 = Ember.Object.create({
    netPrice: 300,
  });
  const services = [service_1, service_2, service_3];
  const tax = Ember.Object.create({
    percentage: 10,
  });
  consts taxes = [tax];

  assert.equal(calculator.calculateGrossPrice(services, taxes), 660);
});
```

In both tests, we had to set up the same services, which is maybe not that bad for two tests but would certainly be a problem for a huge amount of tests. The great thing about the setup is that there is a shared context (`this`) between tests and setup callbacks! So we can assign all the values to `this` inside `beforeEach`:

```
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';

moduleFor('service:invoice-price-calculator', 'Unit | Service |
  invoice-price-calculator', {
  beforeEach() {
    const service_1 = Ember.Object.create({
      netPrice: 100,
    });
    const service_2 = Ember.Object.create({
      netPrice: 200,
    });
    const service_3 = Ember.Object.create({
      netPrice: 300,
    });
    this.services = [service_1, service_2, service_3];
  },
});

test('it calculates net price for the invoice', function(assert) {
  const calculator = this.subject();

  assert.equal(calculator.calculateNetPrice(this.services), 600);
});

test('it calculates gross price for the invoice', function(assert) {
  const calculator = this.subject();

  const tax = Ember.Object.create({
    percentage: 10,
  });
  consts taxes = [tax];

  assert.equal(calculator.calcualteGrossPrice(this.services, taxes), 660);
});
```

Looks much better and much more DRY!

- `TestModuleForComponent` (`moduleForComponent`) - a specialized version of `TestModule` (`moduleFor`) help intended for testing components. By default, it's supposed to test components via integration tests, but you can also write unit tests by providing `unit: true` flag or the list of dependencies via `needs: []` in callbacks. This helper also provides `render` method which lets you render and test the component. Here's an example of a component and its integration test (which uses `htmlbars-inline-precompile` described later in this chapter):

```
// app/components/display-ember-awesomeness-status.js
import Ember from 'ember';

set {
  get,
} = Ember;

export default Ember.Component.extend({
  shouldShowStatus: false,

  actions: {
    showStatus() {
      set(this, 'shouldShowStatus', true);
    },
  }
});
```

```
<!-- app/templates/components/display-ember-awesomeness-status.hbs -->
{{#if shouldShowStatus}}
  <p data-test="status">Ember is Awesome!</p>
{{else}}
  <button data-test="show-status-btn">Show status</button>
{{/if}}
```

```
// tests/integration/components/display-ember-awesomeness-status-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';

const {
  set,
} = Ember;

moduleForComponent('display-ember-awesomeness-status', 'Integration | Component |
  display ember awesomeness status', {
  integration: true
});

test('it shows status after clicking the button', function(assert) {
  assert.expect(2)

  const {
    $,
  } = this;

  this.render(hbs`{{display-ember-awesomeness-status}}`);

  assert.notOk($('[data-test=status]').length, 'status should be hidden');

  $('[data-test=show-status-btn]').click();

  assert.ok($('[data-test=status]').length, 'status should be visible');
});
```

As this helper is supposed to be used only for components, you don't need to pass the type of the subject, just the name of the component is sufficient.

- `TestModuleForModel` (`moduleForModel`) - a specialized version of `TestModule` (`moduleFor`), which provides extra setup and methods making testing models easier. It registers Application Adapter, allows you to access `store` via `this.store()` and overrides subject instantiation - by default it uses `store` to create new instances of the model with `store.createRecord()` passing the test subject as a model name:

```
// tests/unit/models/user-test.js
import { moduleForModel, test } from 'ember-qunit';

moduleForModel('user', 'Unit | Model | user', {
  needs: []
});

test('it exists', function(assert) {
  const model = this.subject();
  const store = this.store();

  assert.ok(model, 'model should exist');
  assert.ok(store, 'store should exist');
});
```

- `hasEmberVersion` - a nifty little helper for checking if you the current Ember version is equal to the specified one or higher concerning major and minor version. If you are developing some addon and want to use a feature introduced in Ember 2.10 and provide a fallback for the older versions, you can do it in the following way:

```
import hasEmberVersion from 'ember-test-helpers/lib/ember-test-helpers/has-ember-version';

if (hasEmberVersion(2, 10)) {
  executeLogicUsingFeatureAvailableFromThatEmberVersion();
} else {
  executeFallback();
}
```

- `wait` - a helper making it possible to wait until all the asynchronous operations, such as timers or HTTP requests, have been executed and only then doing the assertions. It is particularly useful when you don't want to stub out the behavior, but just test the behavior in more real-world circumstances. You will most likely need to use this helper along with QUnit's `async()` helper.

Imagine a simple use case where you need to hide some button after clicking on it after a particular period. This is how we could approach testing it:

```
// tests/integrations/components/hide-button-test.js
```

```
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import wait from 'ember-test-helpers/wait';

moduleForComponent('hide-button', 'Integration | Component | hide button', {
  integration: true
});

test('button is hidden after clicking on it', function(assert) {
  assert.expect(2);

  const done = assert.async();

  const {
    $,
  } = this;

  this.render(hbs`{{hide-btn}}`);

  assert.ok($('[data-test=hide-btn]').length, 'button should be visible');

  $('[data-test=hide-btn]').click();

  wait().then(() => {
    assert.notOk($('[data-test=hide-btn]').length, 'button should not be visible');
    done();
  });
});
```

And here is the component:

```
<!-- app/templates/components/hide-btn.hbs -->
{{#unless hideBtn}}
  <button data-test="hide-btn" {{action "hide"}}>
    Hide me!
  </button>
{{/unless}}
```

```
// app/components/hide-btn.js
import Ember from 'ember';

const {
  run,
  set,
} = Ember;

export default Ember.Component.extend({
  hideBtn: false,

  actions: {
    hide() {
      run.later(this, function() {
        set(this, 'hideBtn', true);
      }, 2000);
    }
  }
});
```

And that's it! We don't need to stub any behavior in out tests or use `Ember.run.later()` (please, never ever do it) in our tests to make sure all the events have been processed.

The only concern here would be that this test takes more than 2 seconds to run. But we could quite easily make it faster by making delay time configurable from the outside:

```
// tests/integrations/components/hide-button-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import wait from 'ember-test-helpers/wait';

moduleForComponent('hide-button', 'Integration | Component | hide button', {
  integration: true
});

test('button is hidden after clicking on it', function(assert) {
  assert.expect(2);

  const done = assert.async();

  const {
    $,
  } = this;

  // let's change the delay time!
  this.render(hbs`{{hide-btn delayTime=0}}`);

  assert.ok($('[data-test=hide-btn]').length, 'button should be visible');

  $('[data-test=hide-btn]').click();

  wait().then(() => {
    assert.notOk($('[data-test=hide-btn]').length, 'button should not be visible');
    done();
  });
});
```

```
// app/components/hide-btn.js
import Ember from 'ember';

const {
  run,
  set,
  get,
} = Ember;

export default Ember.Component.extend({
  hideBtn: false,
  delayTime: 2000,

  actions: {
    hide() {
      const delayTime = get(this, 'delayTime');
      run.later(this, function() {
        set(this, 'hideBtn', true);
      }, delayTime);
    }
  }
});
```

By default `delayTime` is `2000` which is the same as it was, but this time this property is configurable, and we can just pass `0` to make the test much faster. As a nice side effect, we achieved a slightly better design in the component by extracting this delay time to a named property.

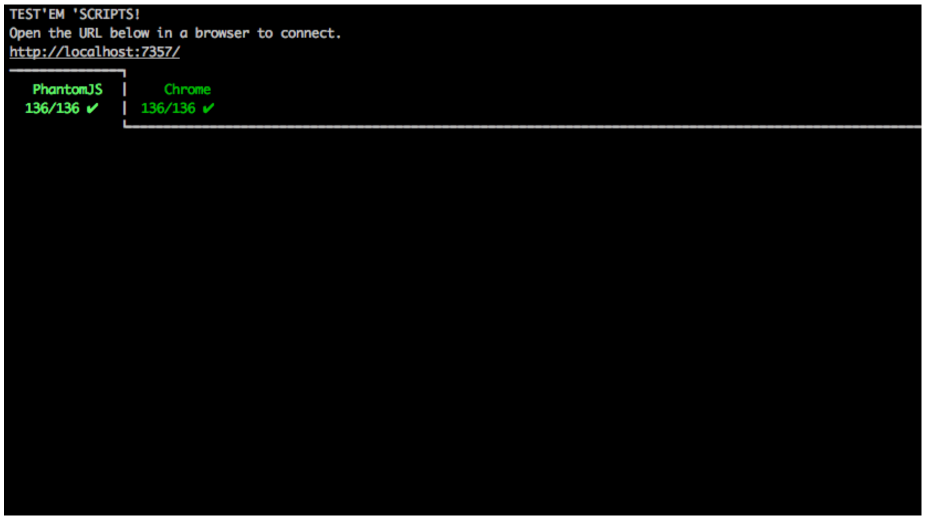
testem

Testem is a test runner used under the hood when running `ember test` and `ember test --server`. However, there's quite a big difference between those two.

The former runs the tests in CI mode which just runs entire test suite and shows how many tests were run, how many passed, how many were skipped and how many failed. As the name suggests, it's supposed to be used for CI. For everyday development, you should use the latter, which will run a particular subset of tests when the file changes which is very convenient for rapid TDD.

Here's an example config for Testem taken from `testem.json` config file in Ember app:

```
<!-- testem.json -->
```



```
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/

  PhantomJS  |     Chrome
 136/136 ✔   |  136/136 ✔
```

Figure 5: Testem

```

{
  "framework": "qunit",
  "test_page": "tests/index.html?hidepassed",
  "disable_watching": true,
  "launch_in_ci": [
    "PhantomJS"
  ],
  "launch_in_dev": [
    "PhantomJS",
    "Chrome"
  ]
}
```

The list of options is quite self-explanatory:

- framework - pass the name of test framework you are using. It could be `qunit`, `jasmine`, `mocha` and `buster`.
- test_page - a path to a customized test page to run tests.
- disable_watching - whether the file watching should be disabled or not. `ember-cli` already handles it, so no need to have it enabled.
- lanuch_in_ci - a list of launchers (such as PhantomJS, Chrome, Firefox) to be used when running tests in CI mode.
- launch_in_dev - a list of launchers to be used when running tests in development mode.

There are plenty of other config options that could be used if necessary; you can learn more about them here.

ember-cli-htmlbars-inline-precompile

An awesome addon which makes it possible to precompile HTMLBars template strings making components testing much easier. That way we can write fewer acceptance tests covering different scenarios of the same feature in favor of components' integration tests, which results ultimately in faster tests.

Its API is pretty limited - you just pass a component's code as if you were rendering it in a real-world handlebar templates:

```
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';

const {
  set,
} = Ember;

moduleForComponent('my-awesome-component', 'Integration | Component | my awesome
  component', {
  integration: true
});

test('button is hidden after clicking on it', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;

  const onUpdateAction = () => {
    assert.ok(true, 'onUpdate has been executed');
  };
  const user = Ember.Object.create();

  set(this, 'onUpdateAction', onUpdateAction);
  set(this, 'user', user);

  this.render(hbs`{{my-awesome-component user=user onUpdate=(action onUpdateAction)}}`);

  $('[data-test=update-btn]').click();
});
```

As you can see it is quite easy to pass bith the properties to the component and the actions! We just need to define them in the current context under the same name that we pass when rendering the component.

pretender / ember-cli-pretender

Pretender is a mock server library which makes it pretty straight-forward to define how mocked endpoints should behave, what kind of payload they should return and verify that the requests were performed to the given endpoints.

The API is limited but powerful - you just need to create a new instance of `Pretender` and mock the endpoints in the callback argument. You can define how the endpoints should behave using `get`, `post`, `put`, `patch`, `delete` and `head` methods.

Each of those methods takes three arguments: a path pattern, a callback handling the logic for given endpoint and an optional timing parameter. By default, all requests are asynchronous, but you can force a synchronous behavior, a response after the specified amount of time or making the endpoint not responding automatically at all which requires a manual resolving.

Each endpoint must return an array of 3 elements: HTTP status code, headers, and body.

Here's an example how you could use it:

```

const booking_1 = {
  {
    type: 'bookings',
    id: '1',
    attributes: {
      'start-at': '2017-01-01T12:00:00Z',
      'end-at': '2017-01-10T08:00:00Z',
    }
  }
};

const booking_2 = {
  {
    type: 'bookings',
    id: '2',
    attributes: {
      'start-at': '2017-02-01T12:00:00Z',
      'end-at': '2017-02-10T08:00:00Z',
    }
  }
};

const bookings = {
  data: [booking_1, bookings_2]
};

const server = new Pretender(function() {
  this.get('/api/bookings',(request) => {
    return [200, {'Content-Type': 'application/json'},
      JSON.stringify(bookings)];
  });

  this.get('/api/bookings/:id', (request) => {
    const idsBookingsMapping = {
      '1': booking_1,
      '2': booking_2,
    };
    const id = request.params.id;

    return [200, {"Content-Type": "application/json"}, JSON.stringify({
      data: idsBookingsMapping[id]
    })];
  }, 5000); // respond after 5 seconds
});
```

And what is this `request` argument passed to a handler callback? It's a `FakeRequest` object which offers a pretty convenient API giving you access to `params`, `queryParams`, `requestBody` or `requestParams` properties making it easy to do proper assertions and stub endpoints with some extra features like filtering which will be much closer to the behavior of a real backend server.

`Pretender` has also some other interesting features like `handledRequest(verb, path, request)` hook which makes it easy to do some assertions about specific endpoints being called or `prepareHeaders(headers)` and `prepareBody(body)` hooks for some extra transformations of headers and body for each request. For a full reference to all features check the docs.

What about `ember-cli-pretender`? It's just an extra layer for the integration of `pretender` with `ember-cli` apps; it doesn't introduce any new features.

`Pretender` is a handy tool, but adding all these handlers and preparing payloads for every endpoint can get quite cumbersome, especially when dealing with a more complex format such as JSONAPI. It would be good to have something that would make it much easier and DRY and add some factories on top to have a mocked backend DB. Fortunately, there's an exact solution for that, which is a wrapper for `pretender` plus tons of other excellent features that make dealing with HTTP requests pretty simple. Time to meet ember-cli-mirage.

ember-cli-mirage

Overview

`ember-cli-mirage` is a powerful tool which provides both mock backend server and factories/fixtures, which makes not only a testing much easier but also the development of the application is much faster - we no longer need real backend server, as we can rely just on a mock backend server. And the remarkable thing is that `ember-cli-mirage` handler JSONAPI format out-of-box! Even if we need any extra customization, we can quickly adjust the specific endpoints to do something more. Let's take a look at some examples.

Getting started

After installing the addon via `ember install ember-cli-mirage` we will see that `mirage` directory was added to the main directory of the app which consists of:

- `config.js` file for defining route handlers,
- `scenarios` directory with `default.js` file which is supposed to be used for seeding mock database
- `serializers` directory with `application.js` serializer which by default handles JSONAPI format.

ember-cli-mirage 101

Let's define some example routes to see ember-cli-mirage in action. As this layer is a wrapper for `pretender`, we may expect that it will work similarly. And that's indeed the case! We can define route handlers for given path pattern for `get`, `post`, `put`, `patch` and `del` methods. Here's a simple example how to define a handler returning all users:

```
// mirage/config.js
this.get('/api/users', () => {
  data: [
    {
      id: '1',
      type: 'users',
      attributes: {
        email: 'ember-cli-mirage@is-awesome.com'
      }
    }
  ]
});
```

It's still a bit simpler than defining handlers in `pretender`; nevertheless, it doesn't offer much of an improvement so far. Let's add some real models and play with `ember-cli-mirage` ORM and mock database and see how powerful it is!

ember-cli-mirage models and route handlers

If we want to interact with a mock in-memory database, we need to define models first. For now, let's focus on the minimum thing that will work and start with generating `user` model. We can use a built-in generator for that:

```
ember g mirage-model user
```

It should generate `user.js` file under `mirage/models` directory with the following body:

```
// mirage/models/user.js
import { Model } from 'ember-cli-mirage';

export default Model;
```

And for now, this is enough to define virtual `users` "table" in the mock database.

Before `0.3.2` version of it was necessary to generate those separate models for `ember-cl-mirage`, but fortunately, we are now able to reuse just Ember Data model for this purpose by setting `discoverEmberDataModels` to true for the ENV:

```
ENV['ember-cli-mirage'] = {
  discoverEmberDataModels: true
};
```

Later in the book, we will use the exact config to avoid generating models manually.

Let's get back to the routes and define route handlers for all the CRUD actions.

Here's the simplest form of a route handler for getting all users:

```
// mirage/config.js
this.get('/api/users', (schema) => {
  return schema.users.all();
});
```

The first argument passed to the handler callback is `schema` which exposes models and database giving us all the tools we need to interact with a mock backend, available under `schema.db` attribute.

One of these model collections is `schema.users` which exposes various methods for manipulating and accessing the specified collection. `all()` method is one of those that return all records for given model, but there are also few more:

- `find(id_or_ids)` - finds record(s) with given id(s). Note that you can also pass the array of ids. You can use it either as `schema.users.find(10)` or `schema.users.find([10, 20, 30])`
- `where(conditions)` - return records matching provided conditions specified as key-value pairs, e.g. `schema.users.where({ isAdmin: true })`
- `first()` - returns first item from collection: `schema.users.first()`
- `new(attributes)` - creates an unpersisted (i.e. without `id`) record with specified attributes: `schema.users.new({ isAdmin: true, full_name: 'Rich Piana' })`
- `create()` - similar to `new()`, but creates a persisted record having `id`, e.g. `schema.users.create({ isAdmin: true, fullName: 'Rich Piana' })`

The important thing is that these methods don't return just some sets of attributes, but real object with a very useful API:

- `attrs()` - returns attributes of given model record:

```
const user = schema.users.find(1);
user.attrs()
  // => { id: 1, fullName: 'Rich Piana' }
```

- `save()` - persists records with applied changes to the attributes' values:

```
const user = schema.users.new({ name: 'Rich' })
user.id
  // => null, record is not persisted

user.save()
user.id
  // => 1, record got persisted

user.name = 'Lazar'
  // the attribute has been assigned, but the changes are
  // not persisted yet
user.save()
  // the changes to the attributes have been persisted
```

- `update(attribute, value)` - updates specified attributes for given records and // persists changes to the database:

```
const user = schema.users.find(1);

user.update('name', 'Rich'})
  // `name` attribute has been updated and the update has been persisted to the database
```

- `destroy()` - removes given record from the database:

```
const user = schema.users.find(1);

user.destroy();
  // record has been removed from the database
```

- `isNew()` - returns `true` if record has not been persisted to the database:

```
const user = schema.users.new({ fullName: 'Lazar Angelov' });

user.isNew(); // true

user.save();
  // user has been persisted
user.isNew(); // false
```

- `isSaved()` - returns true if the record is persisted in the database:

```
const user = schema.users.new({ fullName: 'Lazar Angelov' });

user.isSaved(); // false

user.save();
  // user has been persisted
user.isSaved(); // true
```

What about the database itself, available under `schema.db` attribute?

In most cases you won't probably need to access it directly as accessing the model collection is going to be enough; nevertheless, it might be worth knowing the available API for manipulating it in case it is necessary.

- `collection` - it's not an attribute, but a particular model collection name, e.g. `users`, returning the array with the records of given type. The important thing to keep in mind is that it doesn't return real models, only the database representation of them:

```
schema.db.users[0];
  // { id: '1', fullName: 'Rich Piana' }, just the attributes
```

- `insert(attributes_or_array_with_attributes)` - inserts record or records or with given attributes to the database. It returs the newly inserted record or records with added `id`:

```
schema.db.users.insert({ fullName: 'Rich Piana' });
  // { fullName: "Rich Piana", id: '2' }

schema.db.users.insert([
  { fullName: 'Rich Piana' },
  { fullName: 'Lazar Angelov' }
);
  // [{ fullName: 'Rich Piana', id: '3' }, { fullName: 'Lazar Angelov', id: '4' }]
```

- `find(id_or_ids)` - returns record or record with given id / ids:

```
schema.db.users.find('3');
  // { fullName: 'Rich Piana', id: '3' }

schema.db.users.find(['3', '4']);
  // [{ fullName: 'Rich Piana', id: '3' },
  // { fullName: 'Lazar Angelov', id: '4' }]
```

- `where(conditions)` - returns records matching specified conditions provided as key-value pairs:

```
schema.db.users.where({ fullName: 'Rich Piana });
  // [{ fullName: 'Rich Piana, id: '3' }];
```

- `update(attributes)` - updates all records with given attributes:

```
schema.db.users.update({ isAdmin: true }); // all users are updated with `isAdmin`
  // value set to `true`
```

- `update(record, attributes)` - updates a record identified by either `id` or records matching specified conditions provided as key-value pairs and updates their attributes:

```
schema.db.users.update('3', { isAdmin: true });
  // a user with `id` '3'
  // updated with `isAdmin` value set to `true`
schema.db.users.update({ fullname: 'Lazar Angelov' }, { isAdmin: true });
  // all users with `fullname` 'Lazar Angelov' are updated with `isAdmin` value set to `tru
```

- `remove()` - removes all records from the database:

```
schema.db.users.remove(); // schema.db.users => []
```

- `remove(record)` - removes a record identified by either `id` or records matching specified conditions provided as key-value pairs:

```
schema.db.users.remove('3'); // removes record with id '3'
schema.db.users.remove({ fullname: 'Rich Piana' });
  // removes records with `fullname` 'Rich Piana'
```

- `firstOrCreate(conditions, attributesForCreate)` - finds a record matching specified conditions provided as key-value pairs or creates a new one using specified attributes for create:

```
schema.db.users.firstOrCreate({ fullName: 'Rich Piana' })
  // finds a record with `fullName` 'Rich Piana' or creates a new one
schema.db.users.firstOrCreate({ fullName: 'Rich Piana' }, { isAdmin: true})
  // finds a record with `fullName` 'Rich Piana' or creates
  // a new one assigning also `isAdmin` attribute with `true` value
```

Now that we know quite a lot about `schema` and we can define route handler for getting all records, we need to answer one important question: how does `ember-cli-mirage` handle serialization of the response? The format of the result returned by `schema.users.all()` is far from, e.g. JSONAPI standard which is the default choice in Ember Data. Under the hood `ember-cli-mirage` uses a serializer for given model (which can be defined in `mirage/serializers` directory. By default it's `JSONAPISerializer` as well, but you can also pick `RESTSerializer` and `ActiveModelSerializer`), so we don't need really need to think about the format that Ember Data models expect, the serializer layer will do it for us.

Let's try to define a handler for getting a model with given id:

```
// mirage/config.js
this.get('/api/users/:id', ({ users }, request) => {
  return users.find(request.params.id);
});
```

Just like we define a route handler for given path pattern in `pretender`, we do the same in `ember-cli-mirage`. Also, we had access to `request` param, and so we do in this case! `request` is a second argument in the callback available in every route handler we define. Thanks to ES 6 destructuring feature, we can add some syntactic sugar for getting a specified collection from `schema`. Then, we are simply using `find()` method on collection passing an `id` from the `params`.

And how to implement an action for creating new records? There is `create` method available on collection and that's exactly what we need here:

```
// mirage/config.js
this.post('/api/users', ({ users }, request) => {
  const attributes = JSON.parse(request.requestBody);

  users.create(attributes);
});
```

It's pretty straightforward - we just take `requestBody` from `request`, parse it and create a new record. However, this is not an elegant solution, and there is a potential problem - `ember-cli-mirage` expects a normalized format of the data to be used in `create` method! If your Ember Data models are using JSONAPI format, the payload will be far from the expected format. Fortunately, there is `normalizedRequestAttrs()` helper method implemented for exactly this purpose:

```
// mirage/config.js
this.post('/api/users', function({ users }, request) {
  consts attributes = this.normalizedRequestAttrs();

  return users.create(attrs);
});
```

This method takes care of the normalization process, which is based on the model's serializer, so we don't need to give it much thought.

Note that we need a proper context inside the callback function (`this.normalizedRequestAttrs()`), so we cannot use arrow functions in such case.

Updating models is quite similar, we just need to find given model by `id` and call `update` method passing normalized attributes:

```
this.put('/api/users/:id', function({ users }, request) {
  const id = request.params.id;
  consts attributes = this.normalizedRequestAttrs();

  return users.find(id).update(attributes);
});
```

And for deleting models we just need to call `destroy()` on a model:

```
this.del('/api/users/:id', ({ users }, request) => {
  consts id = request.params.id;

  users.find(id).destroy();
});
```

That way we defined all the CRUD actions for `users` resource. But defining all these handlers is quite repetitive, and most handlers will look the same for other resources. Is it possible to DRY it up a bit? The answer is yes! And it's quite easy.

ember-cli-mirage: shorthands and `resource` helper

Defining a `shorthand` in `ember-cli-mirage` simply means adding a route action without a callback, which is optional. If not provided, the default handler will be used, which looks exactly the same as the callbacks we previously defined. That way entire CRUD for some resource could be defined the following:

```
// mirage/config.js
this.get('/api/users')
this.get('/api/users/:id')
this.post('/api/users')
this.patch('/api/users/:id')
this.delete('/api/users/:id')
```

If we don't want to prefix every path pattern with `api`, we can provide a `namespace` to make it even shorter:

```
// mirage/config.js
this.namespace = '/api';

this.get('/users')
this.get('/users/:id')
this.post('/users')
this.patch('/users/:id')
this.delete('/users/:id')
```

But that's not everything; we can DRY it up even more! Rails-like `resource` helper is available for exactly this purpose which allows defining CRUD actions for given resource:

```
// mirage/config.js
this.namespace = '/api';

this.resource('users');
```

We can also whitelist or blacklist actions using `only` and `except` options:

```
// mirage/config.js
this.namespace = '/api';

this.resource('users', { only: ['index', 'show', 'create'] });
```

```
// mirage/config.js
this.namespace = '/api';

this.resource('users', { except: ['update', 'delete'] });
```

This is the exact mapping between actions and route handlers:

```
| action       | route handler           |
|--------------|-------------------------|
|  index       | this.get('/users');     |
|----------------------------------------|
|  show        | this.get('/users/:id');|
|----------------------------------------|
|  create      | this.post('/users');    |
|----------------------------------------|
|  update      | this.patch('/users');   |
|              | this.put('/users');     |
|----------------------------------------|
|  delete      | this.del('/users/:id');|
```

ember-cli-mirage: models and associations

Associations are a big part of modeling the domain layer. As you may expect, `ember-cli-mirage` has a great API for defining those as well. For this purpose there are two helpers: `belongsTo` and `hasMany`. Let's see them in action:

```
// mirage/models/user.js
import { Model, belongsTo } from 'ember-cli-mirage';

export default Model.extend({
  organization: belongsTo(),
  articles: hasMany(),
});
```

We've just defined a to-one relationship between Users and Organizations and to-many between Users and Articles. If you follow the conventions of the naming, you won't need to provide the literal model name for a given relationship as it will be inferred from the attribute name. However, if the model name differs from attribute name, you will need to provide the proper name as the first argument:

```
// mirage/models/user.js
import { Model, belongsTo } from 'ember-cli-mirage';

export default Model.extend({
  organization: belongsTo('company'),
  articles: hasMany(),
});
```

By declaring those relationships, we gain some dynamically defined methods for manipulating them. We get readers/writers for id/ids and methods for building the associated models.

This is how we could manipulate `organization` relationship:

```
const user = schema.users.find(1);

user.organizationId; // 2
user.organization; // returns organization with id equal to 2
user.organizationId; // 1;
user.organization;
  // returns organization with id equal to 1
user.newOrganization({ name: '5% Nutrition' });
  // builds in-memory organization instance associated to the user
user.createOrganization({ name: '5% Nutrition' });
  // creates persisted organization instance associated to the user
```

And here are the set of methods for manipulating articles:

```
const user = schema.users.find(1);

user.articleIds; // [10, 12, 100]
user.articleIds = [2, 3];
  // replaces current articles with the new set
user.articles;
  // returns array of associated articles
user.articles = [article1, article2];
  // replaces current articles with the new set
user.newArticle({ name: 'Pumping biceps to the max' });
  // builds in-memory article instance associated to the user
```

```
user.createArticle(({ name: 'Pumping biceps to the max' }));
  // creates persisted article instance associated to the user
```

Keep in mind that there is no need to define those relationships if the Ember Data models discovery feature is enabled.

ember-cli-mirage: factories

Attributes

Route handlers and mock database aren't the only things that `ember-cli-mirage` is responsible for. Another layer that makes our life as developers much easier are **factories**.

Factories are blueprints for model records with certain set of attributes and relationships which are used for seeding the database. You can either generate factory files by built-in generator:

`ember g mirage-factory user`

or create them manually inside `mirage/factories` directory.

Let's define a basic factory for creating users:

```
// mirage/factories/user.js
import { Factory } from 'ember-cli-mirage';

export default Factory.extend({
  fullName: 'Rich Piana',
  companyName: '5% Nutrition',
});
```

To define a new factory we need to extend `Factory` and provide the set of attributes. We are not limited to only static attributes like in the example above, but we can also provide dynamic ones taking a sequence number as an argument:

```
// mirage/factories/user.js
import { Factory } from 'ember-cli-mirage';

export default Factory.extend({
  fullName: 'Rich Piana',
  companyName: '5% Nutrition',
  birthDate() {
    return new Date();
  },
  email(i) {
    return `${this.fullName}_${i}@example.com`;
  },
});
```

Note that inside the definition of a dynamic attribute we have access to the current context (`this`) so we can reference other attributes.

We often need to use some random data, but with more meaningful values, especially if we want to reuse such data outside of tests. Fortunately, we can take advantage of `faker` which is included in `ember-cli-mirage` for exactly this purpose:

```
// mirage/factories/user.js
import { Factory, faker } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },
  companyName: '5% Nutrition',
  birthDate() {
    return new Date();
  },
  email(i) {
    return `email_${i}@example.com`;
  },
});
```

You can learn more about `faker` form the docs.

Building And Creating Records From Factories

To instantiate persisted or non-persisted records we can use `create()`/`createList()` for creating one record or multiple records or `build()`/`buildList()` for building one or multiple records. All these methods are available on the `server` object, which is a global injected to every acceptance test, but you can also make it available in unit or integration tests using `startMirage()` initializer:

```
// tests/integration/awesome-integration-test-with-mirage.js
import { startMirage } from 'my-app/initializers/ember-cli-mirage';

moduleForComponent('awesome-integration-test-with-mirage', 'Integration | Component |
  awesome integration test with mirage', {
  integration: true,
  beforeEach() {
    this.server = startMirage();
  },
  afterEach() {
    this.server.shutdown();
  }
});
```

Let's reuse previous factory for `users`:

```
// mirage/factories/user.js
import { Factory, faker } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },
  companyName: '5% Nutrition',
  birthDate() {
    return new Date();
  },
  email(i) {
    return `email_${i}@example.com`;
  },
});
```

By calling `server.create('user')` we would create a persisted `user` record. We could also provide the attributes' overrides if we want to set some custom value instead of relying on attributes defined in the factory:

```
server.create('user', { firstName: 'Rich' });
```

`createList()` method is very similar, we just need to provide the `count` argument for indicating the amount of records we want to create:

```
server.createList('user');
server.createList('user', { firstName: 'Rich' });
```

Both `build` and `buildList` methods work the same as their `create` equivalents, they just create unpersisted records instead.

Setting relationships: `association` helper & `afterCreate` callback

For setting to-one relationships we can use either **association** helper or **afterCreate** callback. **association** sounds much more suitable, and it's easier to use, so let's see it in action. We could reuse the previous example with `users` belonging to some `organization`:

```
// mirage/models/user.js
import { Model, belongsTo } from 'ember-cli-mirage';
```

```

export default Model.extend({
  organization: belongsTo(),
});
```

And here’s our factory:

```
// mirage/factories/user.js
import { Factory, association } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },
  organization: association(),
});
```

Super simple! What if the model name would not follow convention and we called it, e.g., `company`?

```
// mirage/models/user.js
import { Model, belongsTo } from 'ember-cli-mirage';

export default Model.extend({
  organization: belongsTo('company'),
});
```

It would be still the same - `association` helper is “smart” enough to infer the proper model name from the associations defined in the model.

If you don’t like this approach you could also use more generic tool - `afterCreate` callback which takes two arguments: a record which is being created and the `server` instance:

```
// mirage/factories/user.js
import { Factory } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },

  afterCreate(user, server) {
    server.create('organization', { user });
  }
});
```

For setting up to-many relationships we can use `afterCreate` callback as well:

```
// mirage/models/user.js
import { Model, belongsTo } from 'ember-cli-mirage';

export default Model.extend({
  organization: belongsTo('company'),
  articles: hasMany(),
});
```

```
// mirage/factories/user.js
import { Factory } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },

  afterCreate(user, server) {
    server.create('organization', { user });
    server.createList('article', 5, { user });
  }
});
```

Traits

It quite often happens that there are multiple contexts of some model, e.g., a user can be an admin user or not admin-user, or an article can be either published or not published (i.e., a draft). Instead of duplicating such setup logic in the multiple tests, we can use **traits** which are supposed to solve exactly this problem. Let's assume that we need to create admin and non-admin users, users that belong to some organization or do not and also the user having some articles. Here is how to do that using traits:

```
// mirage/factories/user.js
import { Factory, trait, association } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },

  adminUser: trait({
    isAdmin: true,
  }),

  withOrganization: trait({
    organization: association(),
  }),

  withComments: trait({
    afterCreate(user, server) {
      server.createList('article', 5, { user });
    }),
  },
});
```

Notice that we can use both `afterCreate` callback and `association` helper inside traits.

To create records with given traits, we just need to pass them as arguments to `create()`/`createList()` and `build()`/`buildList()` methods:

```
server.create('user', 'adminUser');
server.createList('user', 5, 'adminUser', 'withOrganization');
server.build('user', 'withOrganization', `withComments`);
server.buildList('user', 10, 'withOrganization', `withComments`,
  { firstName: 'Rich' });
```

A great thing (shown in the last example) is that we can also pass the attributes' overrides as the last argument which will take precedence over the attributes from the traits.

ember-cli-mirage: serializers

Serializers layer is responsible for normalizing incoming data (POST and PUT) and serializing data to a right format. There are three types of serializers available in `ember-cli-mirage` out of the box:

- JSONAPISerializer (a default one)
- ActiveModelSerializer
- RestSerializer

It's not the layer that you do a lot of customization, nevertheless, knowing the available API may be extremely valuable when you need to do something extra. You can either customize the global serializer (the one from `mirage/serializers/application.js`) or provide a model-specific serializer. Here's the list of the methods that you will most likely want to customize:

- `serialize(object, request)` - override this method if you need to return different data in the route handlers than the default one. Particularly useful when you want to add, e.g., some meta data to the response:

```
// mirage/serializers/user.js
import BaseSerializer from './application';

export defauls: BaseSerializer.extend({
  serialize(object, request) {
    const originalResponse = BaseSerializer.prototype.serialize.apply(this,
      arguments);

    originalResponse.meta = {
      timestamp: new Date().toString(),
    }
    return originalResponse;
  },
});
```

- `normalize(json)` - customize the payload coming from POST and PUT shorthands. Keep in mind that the format must be JSONAPI-compliant.

- `attrs` - you can use this method to whitelist attributes that should be returned in a serialized response:

```
// mirage/serializers/user.js
import BaseSerializer from './application';

export default BaseSerializer.extend({
  attrs: ['id', 'firstName', 'lastName']
});
```

- `include` - if you need to return sideloaded associations, that's the method you should use. You can provide an array of associations or a function:

```
// mirage/models/user.js
import { Model, hasMany } from 'ember-cli-mirage';

export default Model.extend({
  articles: hasMany(),
});
```

```
// mirage/serializers/user.js
import BaseSerializer from './application';
```

```

export default BaseSerializer.extend({
  include: ['articles'],
});
```

or

```
// mirage/serializers/user.js
import BaseSerializer from './application';

export default BaseSerializer.extend({
  include: function(request) {
    const queryParams = request.queryParams
    if (queryParams && queryParms.include && queryParams.indexOf('articles')) {
      return ['articles']
    } else {
      return [];
    }
  },
});
```

As the second use case is a pretty standard feature, it works out of the box for JSONAPI serializer: just specify the relationships that should be included using `include` query param, and you won't need to customize any serializer.

Check the official docs if you want to learn more.

ember-cli-mirage: seeding database

When it comes to testing it's pretty straightforward - we have `server` object available in acceptance tests or we can manually instantiate it when needed. How about using factories for seeding database for development?

In that case, we need to take advantage of `mirage/scenarios/default.js` file and define the entire setup there inside one function accepting `server` argument:

```
// mirage/scenarios/default.js
export default function(server) {
  server.createList('user', 10, 'admin');
  server.createList('user', 10, 'nonAdmin', 'withArticle');
}
```

No need to depend on the data coming from a backend app when doing the development. Just create the right setup for the data and focus on the important parts.

ember-test-selectors

A fundamental issue when writing acceptance or integration tests is deciding how you should identify elements when accessing them from the test. Should you use some special classes? Or maybe use some particular `data` attributes?

These solutions will surely work, but there are some problems with them. They add some extra stuff to DOM which is not needed besides tests, using special classes might be misleading, and you can't easily pass `data` attributes to components (e.g. `data-test='user-form'`) just like classes as it requires adding some additional attribute bindings in the component. Keeping the conventions consistent between the projects also gets more challenging.

Fortunately, there is a great solution to this problem: ember-test-selectors addon

Thanks to this addon, we can use `data-test-*` attributes in DOM elements, and they will be removed from the production builds! Another awesome feature is that you can pass them to the components and these attributes will be automatically bound. Rendering the following component:

```
{{article-comments comments=comments data-test-article-comments=article.id}}
```

will result in the following DOM:

```
<div id="ember100" data-test-article-comments="1000">
</div>
```

What is more, `ember-test-selectors` comes with a `testSelector` helper which can be used in both acceptance and component integration tests. If we wanted to find the `div` wrapping the component in the previous example, we could write the following acceptance test:

```
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { test } from 'qunit';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance: Finding components');

test('it finds components', function(assert) {
  const element = find(testSelector('article-comments', 1000));

  assert.ok(element);
});
```

```
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { test } from 'qunit';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance: Finding components');

```

```
test('it finds components', function(assert) {
  const element = find(testSelector('article-comments', 1000));

  assert.ok(element);
});
```

and in a component integration tests:

```
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import testSelector from 'ember-test-selectors';

moduleForComponent('my-awesome-component', 'Integration | Component |
  my awesome component', {
  integration: true
});

test('it finds components', function(assert) {
  const element = this.$(testSelector('article-comments', 1000));

  assert.ok(element);
});
```

ember-cli-page-object

In large test suites, it's quite easy to find a lot of repetitions with filling the same forms for testing different scenarios, querying the same elements and making similar assertions. Not only is it not that DRY, but it adds some unnecessary noise to the tests - instead of focusing on the testing scenario you see a bunch of query selectors which are pretty meaningless. Encapsulating querying logic, filling forms, visiting pages and assertions in a separate object that is easily reusable sounds like a good idea. And guess what! There is already a solution for this problem: ember-cli-page-object.

Imagine you are testing a user signup scenario. In such case we would probably want to visit some **signup** page, provide an email, a password, a password confirmation, click a button and then we should see some notification that a user has successfully signed up. An acceptance test for this use case could look like this:

```
// tests/acceptance/user-signup-test.js
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { test } from 'qunit';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance: User SignUp');

```

```
test('user can sign up with valid data', function(assert) {
  assert.expect(1);

  visit('/signup');
  fillIn('[data-test-user-email]', 'email@example.com');
  fillIn('[data-test-user-password]', 'supersecretpassword123');
  fillIn('[data-test-user-password-confirmation]', 'supersecretpassword123');
  click('[data-test-sign-up]');

  andThen(() => {
     const $notification = find(testSelector('success-notification'));
     assert.equal(
        $notification.text().trim(),
        'You have successufully signed up!',
        'success notification should be displayed'
     );
  });
});
```

It doesn't look bad so far. What if we wanted to test another scenario, e.g., that some error notification is displayed when a user provides invalid data? We would have a very similar test with a lot of duplication like how to access given input. It would be much better to have it encapsulated in one place. Let's create our first page object.

We can use a generator provided by the addon for creating new page objects:

```
ember generate page-object singup
```

We should see a new file in `tests/pages` directory. Let's provide an interface for testing signup scenario:

```
// tests/pages/signup.js
import PageObject, {
  clickable,
  fillable,
  text,
  visitable
} from 'my-awesome-app/tests/page-object';

export default PageObject.create({
  visit: visitable('/signup'),

  email: fillable('[data-test-user-email]'),
  password: fillable('[data-test-user-password]'),
  passwordConfirmation: fillable('[data-test-user-password-confirmation]'),
  signUp: clickable('[data-test-sign-up]'),
  successNotification: text('[data-test-success-notification]'),
});
```

There is quite a lot of things going on, so let's break them down. To create a page object, we call, well, `create` method on `PageObject` and specify the steps for a given scenario. We are using some interesting helpers here, so let's break them down:

- `visitable` - a helper for visiting a page under given path.
- `fillable` - fills an input identified by a provided selector.
- `clickable` - clicks element identified by a given selector.
- `text` - finds an element with given selector and extracts the text from it. By default it will normalize the returned text so that we don't need to call `trim()` on it, but if that's not the desired behaviour, we can provide `normalize` option and set it to `false`: `text('[data-test-success-notification]', { normalize: false })`

Most of these helpers accept extra `options` argument where you can provide such options as `scope` (a parent element in which the given element is nested) and some more. You can learn more about them from the official docs.

And this is how we can refactor our previous test with a page object:

```
// tests/acceptance/user-signup-test.js
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { test } from 'qunit';
import page from 'my-awesome-app/tests/pages/singup';

moduleForAcceptance('Acceptance: User SignUp');

test('user can sign up with valid data', function(assert) {
  assert.expect(1);

  page
    .visit()
    .email('email@example.com')
    .password('supersecretpassword123')
    .passwordConfirmation('supersecretpassword123')
    .signUp();

  andThen(() => {
     assert.equal(
       page.successNotification,
        'You have successufully signed up!',
        'success notification should be displayed'
     );
  });
});
```

Looks much better! What if we wanted to add a scenario where the sign up fails because the password confirmation doesn't match the password? We just need to add a step for extracting text for some `errorNotification` and we can reuse the same flow:

```
// tests/pages/signup.js
import PageObject, {
  clickable,
  fillable,
  text,
  visitable
} from 'my-awesome-app/tests/page-object';

export default PageObject.create({
  visit: visitable('/signup'),

  email: fillable('[data-test-user-email]'),
  password: fillable('[data-test-user-password]'),
  passwordConfirmation: fillable('[data-test-user-password-confirmation]'),
  signUp: clickable('[data-test-sign-up]'),
  successNotification: text('[data-test-success-notification]'),
  errorNotification: text('[data-test-error-notification]'),
});
```

And here's another test scenario:

```
// tests/acceptance/user-signup-test.js
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { test } from 'qunit';
import page from 'my-awesome-app/tests/pages/singup';

moduleForAcceptance('Acceptance: User SignUp');

test('user can sign up with valid data', function(assert) {
  assert.expect(1);

  page
    .visit()
    .email('email@example.com')
    .password('supersecretpassword123')
    .passwordConfirmation('supersecretpassword')
    .signUp();

  andThen(() => {
     assert.equal(
        page.errorNotification,
        'Password and password confirmation do not match',
        'error notification should be displayed'
      );
  });
});
```

And that’s it! Very elegant and DRY.

You can even use page objects in components’ integration tests with some extra setup and write much more complex scenarios using plenty of available helpers/ I would highly recommend checking the docs, just to get the idea what kind of helpers are available.

Test-Driving Our Application

Our Example Application - Book.me

Now that we already know the essential tools in **Ember** ecosystem and are fully aware of the value of writing tests and ideally sticking to TDD approach if possible, we can start developing our application.

The problem with many example applications is that there are either too simple to show more complex use cases that are common in real-world applications or they grow huge to the extent that many of the features don't bring much value regarding learning new concepts and only a small part of the application is valuable.

Nevertheless, it is still possible to provide an exciting feature(s) that will be complicated enough with a lot of interesting use cases but won't be too big for the book example.

For the last few years, I've been mostly involved in developing software for vacation rental industry, which makes it quite natural to provide the example within that domain. This industry comes with a lot of complex problems to solve and obviously, I won't attempt to provide any practical examples how to solve those issues as they might not necessarily contain that much concentrated educational value, but rather focus on some CRUD with cool additions.

As the vacation rental industry revolves mostly around reservations for, well, rentals generally speaking (villas, hotels, apartments, etc.), the primary concern of our example application, let's call it "Book.me", will be rentals and bookings management. Imagine you are the owner of multiple properties (rentals). To manage those properly a robust software is surely needed as we expect a lot of reservations and inquiries coming from everywhere. In a real-world scenario creating of the majority of the bookings would be automated by integration with applications like Booking.com, Airbnb or HomeAway, but this is certainly beyond the scope of this book, so let's focus on this particular one use case of creating the reservations manually.

Besides rentals management (simple CRUD), we will need to implement a nice calendar view where we will be able to select some dates and create a booking for given rental. Except for dates, we also need to specify a traveler for the booking, most likely identified by an email and/or a full name. To keep it simple, we will just ask for an email.

In the real-world scenario, we could expect some extra fees that would be added to the reservation like a cleaning fee, airport transfer, breakfast, etc., but adding this extra domain complexity would have little educational value compared to just creating the booking itself, so we can skip that part.

Starting The Development - Generating New App

Now that we know what we are going to develop, let's move to the most interesting part: the application itself.

Surprise, surprise, we will start with generating the new app:

```
ember new book-me
```

To have a more aesthetically pleasing experience when developing our application, let's add some HTML/CSS framework - `bootstrap-bookingsync-sass`, which is based on Bootstrap, is used extensively inside BookingSync universe and looks very nice. We can simply install it as an addon:

```
ember install ember-cli-bootstrap-bookingsync-sass
```

You will be asked for overwriting `app/styles/app.scss` and `app/templates/application.hbs`, just accept the changes, we will modify them a bit later anyway.

The last step is editing `config/environment.js` file and adjusting `contentSecurityPolicy` for handling Google Fonts used by the addon:

```
// config/environment.js
ENV.contentSecurityPolicy = {
  'default-src': "'none'",
  'script-src': "'self' 'unsafe-inline'",
  'style-src': "'self' 'unsafe-inline' https://fonts.googleapis.com",
  'font-src': "'self' fonts.gstatic.com",
  'connect-src': "'self'",
  'img-src': "'self' data:",
  'media-src': "'self'"
}
```

And that's enough for having some a pleasing design in the app. Now we can just start the server:

```
ember s
```

And you should see something like this:

Adding The First Feature - Sign Up And Sign In

Signing up and signing in are not the most exciting features out there as they get pretty repetitive in every app and don't deal much with the core domain of the application. Nevertheless, it's certainly useful to have one and do it from the very beginning - we need to scope models by account (I we don't want our calendar to be public, right?), so it is a good idea to start exactly with this feature. Another benefit is that we can get quickly warmed up by something moderately easy.

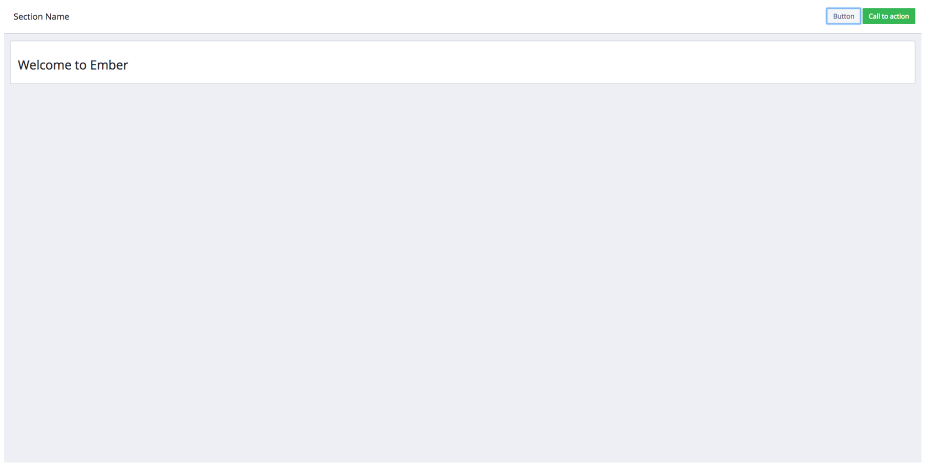


Figure 6: Initial layout

There is no excuse for this feature to not practice TDD, so let's start with a test.

As we will need some extra selectors that are meaningful in the acceptance tests, we need to add ember-test-selectors to our application:

```
ember install ember-test-selectors
```

We will also need ember-cli-mirage, so let' install it now:

```
ember install ember-cli-mirage
```

Now we can generate a new acceptance test:

```
ember g acceptance-test sign-in-sign-up
```

Let's open `book-me/tests/acceptance/sign-in-sign-up-test.js` and write our first test now!

Our first feature will be signing up and ensuring that we are logged in afterward. No need for any extra things like confirmations etc., we want to keep it simple here.

```
// tests/acceptance/sign-in-sign-up-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance | sign in sign up');
test('user can successfully sign up', function(assert) {
  assert.expect(1);

  server.post('/users', function(schema)  {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = {
      email: 'example@email.com',
      password: 'password123',
      passwordConfirmation: 'password123',
    };

    assert.deepEqual(attributes, expectedAttributes, "attributes don't match
      the expected ones");

    return schema.users.create(attributes);
  });

  click(testSelector('signup-link'));

  andThen(() => {
    fillIn(testSelector('signup-email-field'), 'example@email.com');
    fillIn(testSelector('signup-password-field'), 'password123');
    fillIn(testSelector('signup-password-confirmation-field'), 'password123');

    click(testSelector('signup-submit-btn'));
  });
});
```

What we want to do here is filling email, password, password confirmation fields and click some signup button. After clicking that button, we expect to hit `users` endpoint and verify that the proper payload has been sent. We also fall back to a default behavior expected by such endpoint, which is creating a user with the given attributes.

Let's make this test green. We will start with a little customization in `templates/application.hbs`. Locate the navbar which currently should look like this:

```
<!-- book-me/app/templates/application.hbs -->
<div class="collapse navbar-collapse navbar-top-collapse">
  <div class="navbar-right">
    <button class="btn btn-secondary navbar-btn" type="button">Button</button>
    <button class="btn btn-primary navbar-btn" type="button">Call to action</button>
  </div>
</div>
```

Remove all the buttons and replace them with the following link:

```
<!-- book-me/app/templates/application.hbs -->
{{link-to "Sign up" "signup" class="btn btn-primary navbar-btn" data-test-signup-link}}
```

As an extra bonus we may change few more things in the layout. In the following section, remove `Welcome to Ember` header:

```
<!-- book-me/app/templates/application.hbs -->
<section class="main-content">
  <div class="sheet">
    <h1>Welcome to Ember</h1>

    {{outlet}}
  </div>
</section>
```

and replace this part:

```
<!-- book-me/app/templates/application.hbs -->
<div class="navbar-brand-container">
  <span class="navbar-brand">
    <h1><i class="fa fa-star"></i> Section Name</h1>
  </span>
</div>
```

with the following one:

```
<!-- book-me/app/templates/application.hbs -->
<div class="navbar-brand-container">
  <span class="navbar-brand">
    <h1><i class="fa fa-star"></i> Book Me!</h1>
  </span>
</div>
```

Let's get back to making our test happy: `signup` route doesn't exist yet, so let's add it in the router:

```
// book-me/app/router.js
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup');
});

export default Router;
```

And let's generate the route:

```
ember g route signup
```

For handling the registration action, we are going to provide `registerUser` function. To make this action available in the template let's install ember-route-action-helper:

```
ember install ember-route-action-helper
```

Besides adding the action for registration, we also need to create a new `User` record in `beforeModel` hook.

Generating a new model sounds like a reasonable thing to do now:

```
ember g model User
```

We can now add two attributes that we will need for registration: `email` and `password` (we can forget for now about `passwordConfirmation`):

```
// book-me/app/models/user.js
import DS from 'ember-data';

const {
  Model,
  attr,
} = DS;

export default Model.extend({
  email: attr('string'),
  password: attr('string'),
});
```

Before `0.3.2` version of `ember-cli-mirage` it was necessary to generate a separate set of models just for `ember-cl-mirage`, but fortunately, we are now able to reuse just Ember Data model for this purpose. We just need to enable models' discovery feature by adding the following line to the ENV file:

```
// book-me/config/environment.js
ENV['ember-cli-mirage'] = {
  discoverEmberDataModels: true
};
```

Let's implement the logic for `signup` route and template:

```
<!-- book-me/app/templates/signup.hbs -->
<form {{action (route-action "registerUser" user) on="submit"}}>
  <div class="form-group">
    <label for="signup-email">Email address</label>
    {{input
      data-test-signup-email-field
      id="signup-email"
      value=(mut user.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password</label>
    {{input
      data-test-signup-password-field
      id="signup-password"
      value=(mut user.password)
      class="form-control"
      type="password"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password Confirmation</label>
    {{input
      data-test-signup-password-confirmation-field
      id="signup-passwordConfirmation"
      value=(mut user.passwordConfirmation)
      class="form-control"
      type="password"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-signup-submit-btn>Submit</button>
</form>
```

```
// book-me/app/routes/signup.js
import Ember from 'ember';

const {
  set,
} = Ember;

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('user');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'user', model);
  },

  actions: {
    registerUser(user) {
      user.save();
    },
  },
});
```

Nothing fancy going here - we just set up some necessary form fields and create a simple route with `registerUser` action, which is executed when submitting the form.

Note that we haven't added any unit tests for the route. Why is that?

The main reason is that these methods don't need to be tested. `model` and `setupController` hooks may be considered as implementation details, and they are indirectly tested via the acceptance test. The decision whether the actions should be unit-tested or not is more complex though. In this case, the logic is so simple that it doesn't require any other tests, the passing acceptance test makes me confident enough about the code that it will work. The rule of thumb would be to not add any tests unless you see the benefits of them.

Our Mirage backend doesn't know so far how to handle POST requests for `users` endpoint, we can solve that problem with the following addition:

```
// book-me/mirage/config.js
export default function() {
  this.post('/users');
};
```

In most cases the API URLs will be namespaced by `api` or similar a segment, to make it more real-world, we could do the same. Firstly, in the Mirage config file:

```
// book-me/mirage/config.js
export default function() {
  this.namespace = 'api';

  this.post('/users');
};
```

And secondly, in the application adapter which needs to be generated in the first place:

```
ember g adapter application
```

```
// book-me/app/adapters/application.js
import DS from 'ember-data';

export default DS.JSONAPIAdapter.extend({
  namespace: 'api',
});
```

Now our test is passing!

In the TDD cycle, besides writing tests and the actual implementation for satisfying the requirements specified in tests, there is one more phase: refactoring.

At this point, I'm pretty happy with the overall design and the code, so this time we may skip this part.

How about the failure path? What happens if the passwords don't match or we don't fill any input at all and submit the form? It sounds like we need to add some validations.

Just like before, let's start with the test. At this level we don't need to test every possible failure scenario, just testing that some validation works will be good enough for acceptance tests. The details of the validation might be tested on unit-test level or integration-level later.

Here's our test:

```
// tests/acceptance/sign-in-sign-up-test.js
test('user cannot signup if there is an error', function(assert) {
  assert.expect(1);

  server.post('/users', () => {
    assert.notOk(true, 'request should not be performed');
  });

  visit('/');

  click(testSelector('signup-link'));

  andThen(() => {
    fillIn(testSelector('signup-email-field'), 'example@email.com');
    fillIn(testSelector('signup-password-field'), 'password123');

    click(testSelector('signup-submit-btn'));
  });

  andThen(() => {
    assert.ok(find(testSelector('signup-errors')).length,
      'errors should be displayed');
  });
});
```

Quite similar to the previous test, but here we want to make sure that the error is displayed and that no request is performed to `/users` endpoint.

This failure case brings a new challenge: implementing validations. Where should be put it: in the model? In the controller? Or maybe we should generate a new component?

Arguably, the common approach would be adding some validations in a model, especially if you have some experience in Ruby on Rails. However, adding validations to models may create some serious issues if you have multiple contexts of the validations. And the idea of having “invalid object” sounds a bit uncomfortable to me, the params might not be valid in given context, but the model object itself shouldn’t be the subject of validation. That’s why I’ve been a fan of form objects for a long time.

In Ember apps, there are few ways to implement form objects. One way would be simply adding some computed properties in a controller or a component which would act as a form object, validate the values and if everything went fine, it would just assign the values to the model. Another way would be using some model proxy - like ember-changeset or ember-buffered-proxy, so that we don’t operate directly on models, but on something that stands in front of it. I usually choose ember-changeset and its close friend ember-changeset-validations, which provides validation layer for changesets.

Let’s install these addons:

```
ember install ember-changeset
ember install ember-changeset-validations
```

But where are we going to use the changeset? It can be either a controller or a new component. In almost all the cases I choose to go with the components and keep controllers only for some particular use cases where the components are not enough, like query params. Components are easier to test, and they decouple concepts from the routes making them more reusable.

Let’s generate a new `signup` component:

```
ember g component user-signup
```

Let’s start by moving template content from `signup.hbs` to the component’s template:

```
<!-- book-me/app/templates/signup.hbs -->
{{user-signup user=user registerUser=(route-action "registerUser")}}
```

```
<!--  book-me/app/templates/components/user-signup.hbs -->
<form {{action "registerUser" on="submit"}}>
  <div class="form-group">
    <label for="signup-email">Email address</label>
    {{input
      data-test-signup-email-field
      id="signup-email"
      value=(mut user.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password</label>
    {{input
      data-test-signup-password-field
      id="signup-password"
      value=(mut user.password)
      class="form-control"
      type="password"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password Confirmation</label>
    {{input
      data-test-signup-password-confirmation-field
      id="signup-passwordConfirmation"
      value=(mut user.passwordConfirmation)
      class="form-control"
      type="password"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-signup-submit-btn>Submit</button>
</form>
```

Notice that the way the `registerUser` action is invoked has changed as we are no longer invoking a route action, but we are invoking component's action now. Let's add the last changes to the component to make the acceptance test for the successful path happy. However, that will require adding some new code to the component, so let's start with a test as usual:

```
// book-me/tests/integration/components/user-signup-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';

const {
  set,
} = Ember;

moduleForComponent('user-signup', 'Integration | Component | user signup', {
  integration: true
});

test('it invokes passed `registerUser` action when clicking on signup
  button', function(assert) {
  const {
    $,
  } = this;

  assert.expect(1);

  const user = Ember.Object.create();
  const registerUser = (userArgument) => {
    assert.deepEqual(userArgument, user,
      'action should be invoked with proper user argument');
  };

  set(this, 'user', user);
  set(this, 'registerUser', registerUser);

  this.render(hbs`{{user-signup user=user registerUser=registerUser}}`);

  $(testSelector('signup-submit-btn')).click();
});
```

It's just a simple component integration test where we verify that the `registerUser` action was invoked with correct arguments after clicking the signup button.

And here's the implementation:

```
// book-me/app/components/user-signup.js
import Ember from 'ember';

const {
  get,
} = Ember

export default Ember.Component.extend({
  actions: {
    registerUser() {
      const user = get(this, 'user');
      get(this, 'registerUser')(user);
    },
  },
});
```

After all those changes we are back to green - all tests but the one for the failure path are passing, which means we did some refactoring of the code (without changing the behavior).

Now, let's add an integration test for the failure ensuring that the error messages are displayed and that the action is never called:

```
// book-me/app/components/user-signup-test.js
test('it does not invoke passed `registerUser` action when there is a
  validation error and displays the error messages', function(assert) {
  const {
    $,
  } = this;

  assert.expect(1);

  const user = Ember.Object.create();
  const registerUser = () => {
    assert.notOk(true, 'action should not be called');
  };

  set(this, 'user', user);
  set(this, 'registerUser', registerUser);

  this.render(hbs`{{user-signup user=user registerUser=registerUser}}`);

  $(testSelector('signup-submit-btn')).click();

  assert.ok($(testSelector('signup-errors').length), 'errors should be displayed');
});
```

Let's get back to the idea of using changesets, which we will need in a moment for adding validations. However, we will start with refactoring the current behavior: we will just use changesets for syncing properties to model instead of directly using the model. First, we need to change the template:

```
<!-- book-me/app/templates/components/user-signup.hbs -->
<form {{action "registerUser" on="submit"}}>
  <div class="form-group">
    <label for="signup-email">Email address</label>
    {{input
      data-test-signup-email-field
      id="signup-email"
      value=(mut changeset.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password</label>
    {{input
      data-test-signup-password-field
      id="signup-password"
      value=(mut changeset.password)
      class="form-control"
      type="password"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password Confirmation</label>
    {{input
      data-test-signup-password-confirmation-field
      id="signup-passwordConfirmation"
      value=(mut changeset.passwordConfirmation)
      class="form-control"
      type="password"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-signup-submit-btn>Submit</button>
</form>
```

And the component itself:

```
// book-me/app/components/user-signup.js
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const user = get(this, 'user');
    const changeset = new Changeset(user);

    set(this, 'changeset', changeset);
  },

  actions: {
    registerUser() {
      const changeset = get(this, 'changeset');
      get(this, 'registerUser')(changeset);
    },
  },
});
```

There are not that many changes - we merely introduced a changeset, which acts as a proxy for a model and we use it interchangeably in `registerUser` action - the cool thing is that saving the changes in the changesets requires calling the `save` method, just like for models.

However, we need to adjust one testing scenario for the integration test, which is not aware that we use `changeset`:

```
// book-me/tests/integration/componenets/user-signup-test.js
test('it invokes passed `registerUser` action when clicking on
  signup button', function(assert) {
  const {
    $,
  } = this;

  assert.expect(1);

  const user = Ember.Object.create();
  const registerUser = (userArgument) => {
    // a change to make it changeset-aware: userArgument => userArgument._content
    assert.deepEqual(userArgument._content, user,
      'action should be invoked with proper user argument');
  };

  set(this, 'user', user);
  set(this, 'registerUser', registerUser);

  this.render(hbs`{{user-signup user=user registerUser=registerUser}}`);

  $(testSelector('signup-submit-btn')).click();
});
```

Now let's add some validations:

```
ember generate validator user-signup
```

As usual, we are going to start with the tests. The problem with unit-testing validators is that we couple tests to the interface of validators, which sounds a bit like an implementation detail and ideally it should be hidden behind changeset's interface, but doing the integration tests of validators seems to be an overkill, so let's accept the issue of coupling to implementation details and move on.

Changeset validators are higher-order functions which return the validator function. Here is one example:

```
export default function validateCustom(options) {
  return (key, newValue, oldValue, changes, content) => {
    // return true if valid or error message if invalid
  }
}
```

By keeping in mind that the validator function returns `true` for valid results and error message for invalid results and that the validators are simply key-value pairs with attributes names as keys and validator functions as values, we may add some basic tests for `email` format, `password` length and `confirmation` validation for passwords:

```
// book-me/tests/unit/validators/user-signup-test.js
import { module, test } from 'qunit';
import validateUserSignup from 'book-me/validators/user-signup';

module('Unit | Validator | user-signup');

test('it validates email format', function(assert) {
  assert.equal(validateUserSignup.email('email', 'invalid'),
    'Email must be a valid email address');
  assert.ok(validateUserSignup.email('email', 'example@gmail.com'));
});

test('it validates password length', function(assert) {
  assert.equal(validateUserSignup.password('password', 'invalid'),
    'Password is too short (minimum is 8 characters)');
  assert.ok(validateUserSignup.password('password', 'password123'));
});

test('it validates password confirmation', function(assert) {
  assert.equal(validateUserSignup.passwordConfirmation('passwordConfirmation',
    'invalid', '', { password: 'password123' }),
    "Password confirmation doesn't match password");
  assert.ok(validateUserSignup.passwordConfirmation('passwordConfirmation',
    'password123', '', { password: 'password123' }));
});
```

Indeed, we are tightly coupled to the implementation details, but it's good enough, we don't need to strive for the perfect tests suite. Writing those specs require knowing the signature of the validator functions, what kind of arguments do they accept, etc., so it is a good idea to check the docs and get familiar with all these concepts.

Here's the implementation that satisfies the tests:

```
// book-me/app/validators/user-signup.js
import {
  validateLength,
  validateConfirmation,
  validateFormat
} from 'ember-changeset-validations/validators';

export default {
  email: validateFormat({ type: 'email' }),
  password: validateLength({ min: 8 }),
  passwordConfirmation: validateConfirmation({ on: 'password' }),
};
```

Now let's do the actual validation in the components and display the error messages if he changeset happens to be invalid:

```js
// book-me/app/components/user-signup.js
import Ember from 'ember';
import Changeset from 'ember-changeset';
import lookupValidator from 'ember-changeset-validations';
import UserSignupValidators from 'book-me/validators/user-signup';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const user = get(this, 'user');
    const changeset = new Changeset(user, lookupValidator(UserSignupValidators),
      UserSignupValidators);

    set(this, 'changeset', changeset);
  },

  actions: {
    registerUser() {
      const changeset = get(this, 'changeset');

      changeset.validate().then(() => {
        if (get(changeset, 'isValid')) {
          get(this, 'registerUser')(changeset);
        }
      });
    },
  },
});
```

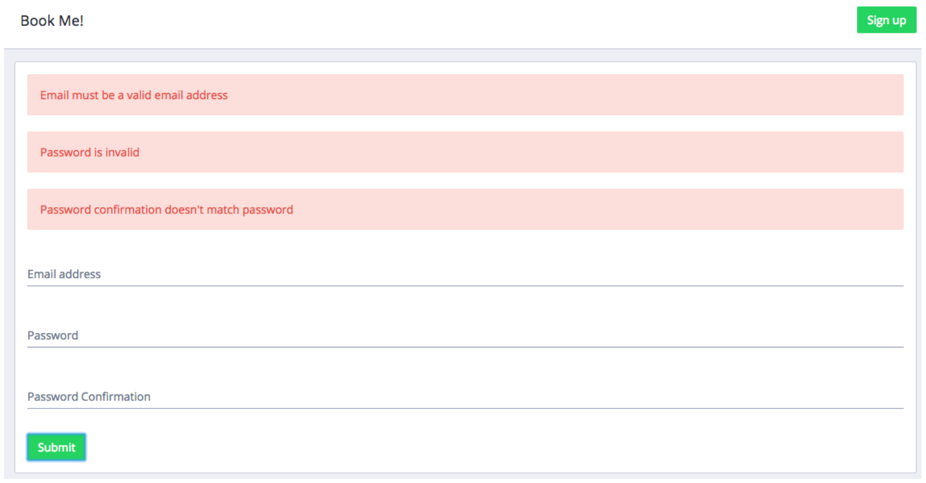
```hbs
<!-- book-me/app/templates/components/user-signup.hbs -->
{{#if changeset.isInvalid}}
  <section data-test-signup-errors>
    {{#each changeset.errors as |error|}}
      <div class="alert alert-danger" role="alert">
        {{error.validation}}
      </div>
    {{/each}}
  </section>
{{/if}}
```

```

<form {{action "registerUser" on="submit"}}>
  <div class="form-group">
    <label for="signup-email">Email address</label>
    {{input
      data-test-signup-email-field
      id="signup-email"
      value=(mut changeset.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password</label>
    {{input
      data-test-signup-password-field
      id="signup-password"
      value=(mut changeset.password)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="signup-password">Password Confirmation</label>
    {{input
      data-test-signup-password-confirmation-field
      id="signup-passwordConfirmation"
      value=(mut changeset.passwordConfirmation)
      class="form-control"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-signup-submit-btn>Submit</button>
</form>
```

To give you an idea how it should look like, here's a screenshot:

We could expect that all tests will be green now, but it turns out we have one failure! The following scenario `'it invokes passed`registerUser`action when clicking on signup button'` in the `user-signup-test.js` fails because the form is invalid. To make it green, we need to fill the inputs with proper data:

```
// book-me/tests/integration/componenets/user-signup-test.js
test('it invokes passed `registerUser` action when clicking on
  signup button', function(assert) {
  const {
    $,
  } = this;

```


Book Me!

Sign up

Email must be a valid email address

Password is invalid

Password confirmation doesn't match password

Email address

Password

Password Confirmation

Submit

Figure 7: Form with errors

```
  assert.expect(1);

  const user = Ember.Object.create();
  const registerUser = (userArgument) => {
    assert.deepEqual(userArgument._content, user,
      'action should be invoked with proper user argument');
  };

  set(this, 'user', user);
  set(this, 'registerUser', registerUser);

  this.render(hbs`{{user-signup user=user registerUser=registerUser}}`);

  $(testSelector('signup-email-field')).val('example@email.com').change();
  $(testSelector('signup-password-field')).val('password123').change();
  $(testSelector('signup-password-confirmation-field')).val('password123').change();

  $(testSelector('signup-submit-btn')).click();
});
```

And that's it! All tests are passing now!

It looks like we have now a working version of sign-up, but it's far from the truth - we've only confirmed so far that our form and validations work. Implementing the proper sign-up will require dealing with tokens and sessions.

When dealing with the authentication process, ember-simple-auth should cover most of our needs. And if not, it is easily extendible so we can either add our authentication strategy or tweak the existing ones.

Let's install the addon:

```
ember install ember-simple-auth
```

Explaining the details how `ember-simple-auth` works and how different authentication flows differ from each other and which one is the most suitable choice is beyond the scope of this book, I highly recommend to read the docs to learn more.

For the sake of simplicity, we will use `OAuth2PasswordGrantAuthenticator` which implements `Resource Owner Password Credentials Grant Type` without a refresh token and `OAuth2BearerAuthorizer` which uses Bearer tokens. The authenticators are the objects responsible for authenticating the session and authorizers use the data acquired by authenticators to handle authorization data that is required when performing the requests.

Let's add the necessary layers to our application. The first thing will be adding authenticators under `authenticators` directory:

```
// book-me/app/authenticators/oauth2.js
import OAuth2PasswordGrant from 'ember-simple-auth/authenticators/oauth2-password-grant';

export default OAuth2PasswordGrant.extend({
  serverTokenEndpoint: '/api/oauth/token',
  serverTokenRevocationEndpoint: '/api/oauth/destroy',
  refreshAccessTokens: false,
});
```

We are extending here `OAuth2PasswordGrant` from `ember-simple-auth` and we are doing some extra customization to specify the endpoint for acquiring tokens and revoking them. We also don't care this time about refresh tokens, so we set `refreshAccessTokens` to `false`.

The next thing will be adding `authorizer` under `authorizers` directory:

```
// book-me/app/authorizers/oauth2.js
import OAuth2Bearer from 'ember-simple-auth/authorizers/oauth2-bearer';

export default OAuth2Bearer.extend();
```

Now let's extend our `ApplicationAdapter` with `DataAdapterMixin` which will be used for properly handling the authorization process:

```
// book-me/app/adapters/application.js
import DS from 'ember-data';
import DataAdapterMixin from 'ember-simple-auth/mixins/data-adapter-mixin';

export default DS.JSONAPIAdapter.extend(DataAdapterMixin, {
  namespace: 'api',
  authorizer: 'authorizer:oauth2',
});
```

At this point, we need to update the unit test for the `ApplicationAdapter` which will be failing now because of the `service:session` dependency. Let's fix it now:

```
// book-me/tests/unit/adapters/application.js
import { moduleFor, test } from 'ember-qunit';

moduleFor('adapter:application', 'Unit | Adapter | application', {
  // added the dependency
  needs: ['service:session']
});
```

Let's focus now on simulating the backend part when it comes to tokens and authentication process in `ember-cli-mirage` config. We need two endpoints: one for handling the login and one for the logout processes. For logout it is pretty simple: we will assume that the response is always successful, so we will just return a response with 204 HTTP code and no body. For login it's a bit more complex: we need somehow to implement the authentication process. The simplest way to do it (which would also be close to what happens on backend server) would be to find the user by provided email and compare the provided password with user's password. In that case, we will return the data in the expected format. Otherwise, we will return 401 status to indicate that the request is not authenticated with some error message. Here's how we can approach this problem:

```
// book-me/mirage/config.js
import Mirage from 'ember-cli-mirage';

const {
  Response,
} = Mirage;

export default function() {
  this.namespace = 'api';

  this.post('/users');

  this.post('/oauth/token', (schema, request) => {
    const potentialPasswordMatch = request.requestBody.match(/password=([^&]*)/);
    const potentialEmailMatch = request.requestBody.match(/username=([^&]*)/);
    // example: [
    //   "password=password123",
    //   "password123",
    //   index: 50,
    //   input: "grant_type=password&username=example%40gmail.com&password=password123"
    // ]
    const password = potentialPasswordMatch && potentialPasswordMatch[1];
    const email = potentialEmailMatch && decodeURIComponent(potentialEmailMatch[1]);

    const user = schema.users.findBy({ email });

    if (!user || user.password !== password) {
      return new Response(401, {}, { message: 'invalid credentials' });
    } else {
      return {
        access_token: '123456789',
        token_type: 'bearer',
        user_id: user.id,
      };
    }
  });

  this.post('/oauth/destroy', () => {
    return new Response(204);
  });
}
```

At this point we've already made a proper setup for authentication and authorization process, so we can add another scenario. What we want to achieve is to make sure that the `token` endpoint is indeed reached and that we transition to some authentication-protected route, let's call it an `admin` route, after a successful signup.

As `ember-cli-mirage` uses `pretender` internally, we could take advantage of a great feature provided by `pretender` - recording of handled requests. Thanks to this feature, we can check all the requests that have been performed with the URLs of the endpoints, request bodies, etc. Let's modify our `user can successfully sign up` scenario:

```
// book-me/tests/acceptance/sign-in-sign-up-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance | sign in sign up', {
  beforeEach() {
    this.email = 'example@email.com';
    this.password = 'password123';
  }
});
test('user can successfully sign up', function(assert) {
  assert.expect(3);

  const { email, password } = this;

  server.post('/users', function(schema)  {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = {
      email: email,
      password: password,
    };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.users.create(attributes);
  });

  visit('/');

  click(testSelector('signup-link'));

  andThen(() => {
    fillIn(testSelector('signup-email-field'), email);
    fillIn(testSelector('signup-password-field'), password);
    fillIn(testSelector('signup-password-confirmation-field'), password);

    click(testSelector('signup-submit-btn'));
  });

  // new scenario: make sure that the request to `tokens` endpoint is performed
  andThen(() => {
    const tokenUrl = '/api/oauth/token';
    const tokenRequest = server.pretender.handledRequests.find((request) => {
      return request.url === tokenUrl;
    });

    assert.ok(tokenRequest, 'tokenRequest should be performed');
    assert.equal(currentURL(), '/admin');
  });
});
```

Let’s make our test suite happy again. We will start with some adjustments in `signup` route. The simplest way to solve our problem will be using `session` service from `ember-simple-auth` and authenticating the user after it gets created:

```
// book-me/app/routes/signup.js
import Ember from 'ember';
```

```

const {
  set,
  get,
  getProperties,
  inject: {
    service,
  }
} = Ember;

export default Ember.Route.extend({
  session: service(),

  model() {
    return this.store.createRecord('user');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'user', model);
  },

  actions: {
    registerUser(user) {
      user.save().then(() => {
        const { email, password } = getProperties(user, 'email', 'password');

        get(this, 'session').authenticate('authenticator:oauth2', email, password);
      });
    },
  },
});
```

We also need to update the unit test for the route and inject `service:session` dependency:

```
// book-me/tests/unit/routes/sign-up-test.js
import { moduleFor, test } from 'ember-qunit';

moduleFor('route:signup', 'Unit | Route | signup', {
  needs: ['service:session']
});
```

We are almost there; now we are only missing the implementation for the final step: the transition to `admin` route. To handle this step, we can just transition to that route after the successful authentication:

```
// book-me/app/routes/signup.js
export default Ember.Route.extend({

  // the rest of the logic
  actions: {
    registerUser(user) {
      user.save().then(() => {
        const { email, password } = getProperties(user, 'email', 'password');

        get(this, 'session').authenticate('authenticator:oauth2', email,
          password).then(() => {
            this.transitionTo('admin');
        });
      });
    },
  },
});
```

We also need to set up the routing and the template for the new route:

```
// book-me/app/router.js
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup');
  this.route('admin'); // new route
});

export default Router;
```

Here is a new route:

```
import Ember from 'ember';

export default Ember.Route.extend();
```

Let’s skip the authentication-protection for this route to not add too much code at once, especially if this part is not covered with any tests.

And the final step, the template:

```
<!-- book-me/app/templates/admin.hbs -->
<h2>Admin</h2>
```

And all the tests are passing again! However, there is one scenario we are missing: what if the error comes not from the client-side validation, but from the server? So far we've only covered the error case when the validation fails in Ember app. Let's add another test for the scenario we've just discovered:

```
// tests/acceptance/sign-in-sign-up-test.js
test('user cannot signup if there is an error on server', function(assert) {
  assert.expect(1);

  const { email, password } = this;

  server.post('/users', () => {
    const errors = {
      errors: [
        {
          detail: 'is already taken',
          source: {
            pointer: 'data/attributes/email'
          }
        }
      ]
    };
    return new Response(422, {}, errors);
  });

  visit('/');

  click(testSelector('signup-link'));

  andThen(() => {
    fillIn(testSelector('signup-email-field'), email);
    fillIn(testSelector('signup-password-field'), password);
    fillIn(testSelector('signup-password-confirmation-field'), password);

    click(testSelector('signup-submit-btn'));
  });

  andThen(() => {
    assert.ok(find(testSelector('signup-errors')).length, 'errors should be displayed');
  });
});
```

To make the new test pass, we just need to handle the error scenario in `registerUser` action in `signup` route:

```
// book-me/app/routes/signup.js
  actions: {
    registerUser(user) {
      user.save().then(() => {
        const { email, password } = getProperties(user, 'email', 'password');

        get(this, 'session').authenticate('authenticator:oauth2', email,
          password).then(() => {
            this.transitionTo('admin');
          }).catch(() => { // handle error scenario
              get(user._content, 'errors').forEach(({ attribute, message }) => {
                  user.pushErrors(attribute, message);
              });
        });
      });
    },
  },
```

As this is a scenario for handling server-side errors, our `user` changeset won't be automatically populated with model errors; we need to do it manually. Fortunately, populating the errors is handled by Ember Data and we can just loop over all the errors and add them to the changeset using `pushErrors` method.

Now, we can proceed to the next feature: making `admin` route protected by the authentication. Again, let's start with the test, the acceptance one:

```
ember g acceptance-test access-admin
```

And here's our test:

```
// book-me/tests/acceptance/access-admin-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';

moduleForAcceptance('Acceptance | access admin');

test('it is not possible to visit `admin` without authentication', function(assert) {
  assert.expect(1);

  visit('/admin');

  andThen(() => {
    assert.equal(currentPath(), 'login',
      'should not be an admin route for not authenticated users');
  });
});

test('it is possible to visit `admin` when user is authenticated', function(assert) {
  assert.expect(1);

  const user = server.create('user');
  authenticateSession(this.application, { user_id: user.id });

  visit('/admin');

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'should be an admin route');
  });
});
```

We are taking advantage of `authenticateSession` provided by `ember-simple-auth` which greatly simplifies authentication process for tests' setup. We want to verify two scenarios: one is that without authentication we can't access the `admin` route and the other one is that after being authenticated we can access that route.

To make the `admin` route, protected we need to include `AuthenticatedRouteMixin` from `ember-simple-auth`:

```
// book-me/app/routes/admin.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

export default Ember.Route.extend(AuthenticatedRouteMixin, {
});
```

By default `ember-simple-auth` performs a transition to `login` route if the user is not authenticated, so it would be a good idea to add this route in the first place:

```
// book-me/app/router.js
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup');
  this.route('login'); // new route
  this.route('admin');
});

export default Router;
```

And now we are back to green! All tests are passing.

If we already added the `login` route, it would be a good idea to implement the login process itself.

Again, let's start with the acceptance test:

```
ember g acceptance-test user-login
```

We want to cover here three scenarios: First, that after providing the valid email and password combo the user will be logged in and redirected to `admin` route. The other two would be failures for both client and server side issues. Here are the tests covering these scenarios:

```
// book-me/tests/acceptance/user-login-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';

moduleForAcceptance('Acceptance | user login', {
  beforeEach() {
    const email = 'example@email.com';
    const password = 'password123';

    this.email = email;
    this.password = password;
    this.user = server.create('user', { email, password, });
  }
});

test('user can successfully log in and is redirected to /admin route', function(assert) {
  assert.expect(1);

  const { email, password } = this;

  visit('/');

  click(testSelector('login-link'));

  andThen(() => {
    fillIn(testSelector('login-email-field'), email);
    fillIn(testSelector('login-password-field'), password);

    click(testSelector('login-submit-btn'));
  });

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'should be an admin route');
  });
});

test('user cannot log in with invalid credentials and sees the error messages from
  client', function(assert) {
  assert.expect(2);

  visit('/');

  click(testSelector('login-link'));

  andThen(() => {
    fillIn(testSelector('login-email-field'), '');
    fillIn(testSelector('login-password-field'), '');

    click(testSelector('login-submit-btn'));
  });

  andThen(() => {
    assert.equal(currentPath(), 'login', 'should still be a login route');
    assert.ok(find(testSelector('login-errors')).length, 'errors should be displayed');
  });
});

test('user cannot log in with invalid credentials and sees the error messages
  from server', function(assert) {
  assert.expect(2);

  visit('/');

  click(testSelector('login-link'));

  andThen(() => {
    fillIn(testSelector('login-email-field'), this.email);
    fillIn(testSelector('login-password-field'), 'invalidPassword');

    click(testSelector('login-submit-btn'));
  });

  andThen(() => {
    assert.equal(currentPath(), 'login', 'should still be a login route');
    assert.ok(find(testSelector('login-errors')).length, 'errors should be displayed');
  });
});
```

Let's make these tests green now. We need to start with generating a `login` route:

```
ember g route login
```

Let's do something similar as we did for signup and generate `user-login` component. It may sound a bit like a Big Design Upfront. However, we already did something very similar for the signup process and we can easily predict that having a separate component will be useful in this use case as well. Let's generate it then:

```
ember g component user-login
```

Before adding anything new to the `login` or `user-login` templates, let's add the actual link to the login page in the layout template (`application.hbs`):

```
<!-- book-me/app/templates/application.hbs -->
<nav class="navbar navbar-default navbar-fixed-top" role="navigation">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle navbar-toggle-context"
              data-toggle="collapse" data-target=".navbar-top-collapse">
        <span class="sr-only">Toggle Navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <div class="navbar-brand-container">
        <span class="navbar-brand">
          <h1><i class="fa fa-star"></i> Book Me!</h1>
        </span>
      </div>
    </div>
    <div class="collapse navbar-collapse navbar-top-collapse">
      <div class="navbar-right">
        {{link-to "Sign up" "signup" data-test-signup-link
          class="btn btn-primary navbar-btn"}}
        {{link-to "Login" "login" data-test-login-link
          class="btn btn-primary navbar-btn"}}
      </div>
    </div>
  </div>
</nav>
<section class="main-content">
  <div class="sheet">
    {{outlet}}
  </div>
</section>
```

And let's render the component in `login` template:

```
<!-- book-me/app/templates/login.js -->
{{user-login}}
```

Just like for the **signup** use case, we want the **login** component to encapsulate data aggregation for the process, handle validation and if the data is valid, call some action that would handle the actual login. That action should probably come from the route. But for now, let's focus exclusively on the component. Just like before, we are going to start with the tests.

```
// app/tests/integration/components/user-login-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';

const {
  set,
} = Ember;

moduleForComponent('user-login', 'Integration | Component | user login', {
  integration: true
});

test('it invokes passed `loginUser` action when clicking on login button',
  function(assert) {
  const {
    $,
  } = this;

  assert.expect(1);

  const loginModel = Ember.Object.create();
  const loginUser = (loginArgument) => {
    assert.deepEqual(loginArgument._content,
      loginModel, 'action should be invoked with proper user argument');
  };

  set(this, 'loginUser', loginUser);

  this.render(hbs`{{user-login loginUser=loginUser}}`);

  $(testSelector('login-email-field')).val('example@email.com').change();
  $(testSelector('login-password-field')).val('password').change();

  $(testSelector('login-submit-btn')).click();
});

test('it does not invoke passed `loginUser` action when there is a
  validation error and displays the error messages', function(assert) {
  const {
    $,
  } = this;

  assert.expect(1);

  const loginUser = () => {
    assert.notOk(true, 'action should not be called');
  };

  set(this, 'loginUser', loginUser);

  this.render(hbs`{{user-login loginUser=loginUser}}`);

  $(testSelector('login-submit-btn')).click();

  assert.ok($(testSelector('login-errors').length), 'errors should be displayed');
});
```

These tests are quite similar to the ones for the **signup** - we want to test both success and failure scenarios. For the success one, we want to make sure that the action passed to the component is called with the proper arguments and for the failure case, we want to ensure that the action is not called and the error messages from validation are displayed.

Time to make these tests happy. Here is the simple component to make the success scenario pass:

```
// book-me/app/components/user-login.js
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const loginModel = Ember.Object.create();
    const changeset = new Changeset(loginModel);

    set(this, 'changeset', changeset);
  },

  actions: {
    loginUser() {
      const changeset = get(this, 'changeset');

      get(this, 'loginUser')(changeset);
    },
  },
});
```

And here's the template:

```
<!-- book-me/app/templates/components/user-login.hbs -->
<h2>Log in</h2>

<form {{action "loginUser" on="submit"}}>
  <div class="form-group">
    <label for="login-email">Email address</label>
    {{input
      data-test-login-email-field
      id="login-email"
      value=(mut changeset.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="login-password">Password</label>
    {{input
      data-test-login-password-field
      id="login-password"
      value=(mut changeset.password)
      class="form-control"
      type="password"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-login-submit-btn>Log in</button>
</form>
```

Not many surprises here: we just set up the proper `changeset` and provide input fields for `email` and `password` and obviously the submit button. We are using a virtual `loginModel` - we don't want to use any Ember Data model here as it doesn't make much sense - we just need some something that can merely aggregate the data, so we are simply creating an Ember Object to be used as such attributes' aggregate.

With this code, we managed to get the integration tests pass. Let's deal with the validation messages now. First, we are going to decide on the actual validations we need and write tests for them.

We don't need to make it overly complicated, so let's just validate the format of the `email` address and make sure that the password has at least eight characters, which is also the requirement for signup process.

Here are the tests:

```
// book-me/tests/unit/validators/user-login-test.js
import { module, test } from 'qunit';
import validateUserLogin from 'book-me/validators/user-login';

module('Unit | Validator | user-login');

test('it validates email format', function(assert) {
  assert.equal(validateUserLogin.email('email', 'invalid'),
    'Email must be a valid email address');
  assert.ok(validateUserLogin.email('email', 'example@gmail.com'));
});

test('it validates password length', function(assert) {
  assert.equal(validateUserLogin.password('password', 'invalid'),
    'Password is too short (minimum is 8 characters)');
  assert.ok(validateUserLogin.password('password', 'password123'));
});
```

And the implementation that will make these tests pass:

```
// book-me/app/validators/user-login.js
import {
  validateLength,
  validateFormat
} from 'ember-changeset-validations/validators';

export default {
  email: validateFormat({ type: 'email' }),
  password: validateLength({ min: 8 }),
};
```

Let's get back to the login component. It's almost the same as the signup component, so we may handle it with the same flow:

```
// book-me/app/components/user-login.js
import Ember from 'ember';
import Changeset from 'ember-changeset';
import lookupValidator from 'ember-changeset-validations';
import UserLoginValidators from 'book-me/validators/user-login';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const loginModel = Ember.Object.create();
    const changeset = new Changeset(loginModel, lookupValidator(UserLoginValidators),
      UserLoginValidators);

    set(this, 'changeset', changeset);
  },

  actions: {
    loginUser() {
      const changeset = get(this, 'changeset');

      changeset.validate().then(() => {
        if (get(changeset, 'isValid')) {
          get(this, 'loginUser')(changeset);
        }
      });
    },
  },
});
```

And, as the last step, let's display the error messages if the changeset is invalid:

```
<!-- book-me/app/templates/components/user-login.js -->
<h2>Log in</h2>

{{#if changeset.isInvalid}}
  <section data-test-login-errors>
    {{#each changeset.errors as |error|}}
      <div class="alert alert-danger" role="alert">
        {{error.validation}}
      </div>
    {{/each}}
  </section>
{{/if}}

<form {{action "loginUser" on="submit"}}>
```

```
  <div class="form-group">
    <label for="login-email">Email address</label>
    {{input
      data-test-login-email-field
      id="login-email"
      value=(mut changeset.email)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="login-password">Password</label>
    {{input
      data-test-login-password-field
      id="login-password"
      value=(mut changeset.password)
      class="form-control"
      type="password"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-login-submit-btn>Log in</button>
</form>
```

We've managed to make all the integration tests pass; however, we still have the acceptance ones failing. Apparently, we haven't defined `loginUser` function yet that will be responsible for the actual logging in, so let's add it now. Just like for the signup process, we need a route action:

```
// book-me/app/routes/login.js
import Ember from 'ember';

const {
  get,
  getProperties,
  inject: {
    service,
  }
} = Ember;

export default Ember.Route.extend({
  session: service(),

  actions: {
    loginUser(loginModel) {
      const { email, password } = getProperties(loginModel, 'email', 'password');

      get(this, 'session').authenticate('authenticator:oauth2', email,
        password).then(() => {
          this.transitionTo('admin');
        }).catch((error) => {
          loginModel.addError('login', error.message);
        });
    },
  },
});
```

Now we just need a final adjustment in login route unit test as we injected the `session` service:

```
// book-me/tests/unit/routes/login-test.js
import { moduleFor, test } from 'ember-qunit';

moduleFor('route:login', 'Unit | Route | login', {
  needs: ['service:session']
});

test('it exists', function(assert) {
  let route = this.subject();
  assert.ok(route);
});
```

All there tests are passing now! It seems like we've just finished our first feature.

However, there are some duplications here and there. Some of the tests have quite a similar setup, especially the ones for the signup - they require filling some input and submitting the form. That's a perfect use case to DRY up with page objects! Let's install `ember-cli-page-object` addon:

```
ember install ember-cli-page-object
```

and generate a page object for the signup process:

```
ember generate page-object signup
```

Interacting with the signup page consists of visiting the page, filling the fields with the proper values and submitting the form. For such use case, this is how our page object may look like:

```
// book-me/tests/pages/signup.js
import {
  create,
  visitable,
  clickable,
  fillable,
} from 'ember-cli-page-object';
import testSelector from 'ember-test-selectors';

```

```
export default create({
  visit: visitable('/'),
  goToSignup: clickable(testSelector('signup-link')),
  email: fillable(testSelector('signup-email-field')),
  password: fillable(testSelector('signup-password-field')),
  passwordConfirmation: fillable(testSelector('signup-password-confirmation-field')),
  submit: clickable(testSelector('signup-submit-btn')),
});
```

And here are the acceptance `sign in sign up` tests after the refactoring to page objects:

```
// book-me/tests/acceptance/sign-in-sign-up-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import Mirage from 'ember-cli-mirage';
import signupPage from 'book-me/tests/pages/signup';

const {
  Response,
} = Mirage;

moduleForAcceptance('Acceptance | sign in sign up', {
  beforeEach() {
    this.email = 'example@email.com';
    this.password = 'password123';
  }
});
test('user can successfully sign up', function(assert) {
  assert.expect(3);

  const { email, password } = this;

  server.post('/users', function(schema)  {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = {
      email: email,
      password: password,
    };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.users.create(attributes);
  });

  andThen(() => {
    signupPage
      .visit()
      .goToSignup()
      .email(email)
      .password(password)
      .passwordConfirmation(password)
      .submit();
  });

  andThen(() => {
    const tokenUrl = '/api/oauth/token';
    const tokenRequest = server.pretender.handledRequests.find((request) => {
      return request.url === tokenUrl;
    });

    assert.ok(tokenRequest, 'tokenRequest should be performed');
    assert.equal(currentURL(), '/admin');
  });
});

test('user cannot signup if there is an error', function(assert) {
  assert.expect(1);

  const { email, password } = this;

  server.post('/users', () => {
    assert.notOk(true, 'request should not be performed');
  });

  andThen(() => {
    signupPage
      .visit()
      .goToSignup()
      .email(email)
      .password(password)
      .submit();
  });

  andThen(() => {
    assert.ok(find(testSelector('signup-errors')).length, 'errors should be displayed');
  });
});

test('user cannot signup if there is an error on server when fetching a token',
  function(assert) {
  assert.expect(1);

  const { email, password } = this;

  server.post('/oauth/token', () => {
    return new Response(401, {}, { message: 'invalid credentials' });
  });

  andThen(() => {
    signupPage
      .visit()
      .goToSignup()
      .email(email)
      .password(password)
      .passwordConfirmation(password)
      .submit();
  });

  andThen(() => {
    assert.ok(find(testSelector('signup-errors')).length, 'errors should be displayed');
  });
});

test('user cannot signup if there is an error on server when creating a user',
  function(assert) {
  assert.expect(1);

  const { email, password } = this;

  server.post('/users', () => {
    const errors = {
      errors: [
        {
          detail: 'is already taken',
          source: {
            pointer: 'data/attributes/email'
          }
        }
      ]
    };
    return new Response(422, {}, errors);
  });

  andThen(() => {
    signupPage
```

```
      .visit()
      .goToSignup()
      .email(email)
      .password(password)
      .passwordConfirmation(password)
      .submit();
  });

  andThen(() => {
    assert.ok(find(testSelector('signup-errors')).length, 'errors should be displayed');
  });
});
```

Looks much better! The code is more readable, more reusable and the implementation details are hidden behind expressive methods. Let's do the same thing for the tests for `user login` scenario:

```
ember generate page-object login
```

Here are the steps for interacting with `login` page:

```
// book-me/tests/pages/login.js
import {
  create,
  visitable,
  clickable,
  fillable,
} from 'ember-cli-page-object';
import testSelector from 'ember-test-selectors';

export default create({
  visit: visitable('/'),
  goTologin: clickable(testSelector('login-link')),
  email: fillable(testSelector('login-email-field')),
  password: fillable(testSelector('login-password-field')),
  submit: clickable(testSelector('login-submit-btn')),
});
```

And the tests after the refactoring:

```
// book-me/tests/acceptance/user-login-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import loginPage from 'book-me/tests/pages/login';

moduleForAcceptance('Acceptance | user login', {
  beforeEach() {
    const email = 'example@email.com';
    const password = 'password123';

    this.email = email;
    this.password = password;
    this.user = server.create('user', { email, password, });
  }
});

test('user can successfully log in and is redirected to /admin route', function(assert) {
  assert.expect(1);

  const { email, password } = this;

  andThen(() => {
    loginPage
      .visit()
      .goTologin()
      .email(email)
      .password(password)
      .submit();
  });

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'should be an admin route');
  });
});

test('user cannot log in with invalid credentials and sees the error messages
  from client', function(assert) {
  assert.expect(2);

  andThen(() => {
    loginPage
      .visit()
      .goTologin()
      .email('')
      .password('')
      .submit();
  });

  andThen(() => {
    assert.equal(currentPath(), 'login', 'should still be a login route');
    assert.ok(find(testSelector('login-errors')).length, 'errors should be displayed');
  });
});
```

```

test('user cannot log in with invalid credentials and sees the error messages
  from server', function(assert) {
  assert.expect(2);

  andThen(() => {
    loginPage
      .visit()
      .goTologin()
      .email(this.email)
      .password('invalidPassword')
      .submit();
  });

  andThen(() => {
    assert.equal(currentPath(), 'login', 'should still be a login route');
    assert.ok(find(testSelector('login-errors')).length, 'errors should be displayed');
  });
});
```

Again, much cleaner!

To put the cherry on top, let's make some improvements when it comes to sign in / sign up process - we can do both of these things, but so far we are not able to log out! The other problem is that `Sign up` and `Login` buttons are displayed even when a user is authenticated. We need to change that. Obviously, we will start with a spec - for that purpose, we will extend `User Login` and `user can successfully log in and is redirected to /admin route` scenarios a bit:

```
// book-me/tests/acceptance/user-login-test.js
test('user can successfully log in and is redirected to /admin route and is
  able to logout afterward', function(assert) {
  assert.expect(4);

  const { email, password } = this;

  andThen(() => {
    loginPage
      .visit()
      .goTologin()
      .email(email)
      .password(password)
      .submit();
  });

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'should be an admin route');
    assert.notOk(find(testSelector('signup-link')).length,
      'signup button should not be displayed');
    assert.notOk(find(testSelector('login-link')).length,
      'login button should not be displayed');
  });

  click(testSelector('logout-link'));

  andThen(() => {
    assert.equal(currentPath(), 'index', 'should be an application route');
  });
});
```

To make the new scenario pass, we just need to add an action for handling logging out and make the proper adjustments in the templates. We can easily reuse the example that is documented in `ember-simple-auth` docs. Let's start with the injection of `session` service to `application` route and defining `logOut` action:

```
// book-me/app/routes/application.js
import Ember from 'ember';

const {
  inject: { service },
  get,
  set,
} = Ember;

export default Ember.Route.extend({
  session: service(),

  setupController(controller) {
    this._super();

    set(controller, 'session', get(this, 'session'));
  },

  actions: {
    logOut() {
      get(this, 'session').invalidate().then(() => {
        this.transitionTo('application');
      });
    },
  },
});
```

And update `application` template:

```
<!-- book-me/app/templates/application.hbs -->
<div class="collapse navbar-collapse navbar-top-collapse">
  <div class="navbar-right">
    {{#if session.isAuthenticated}}
      <a {{action (route-action 'logOut')}}
        data-test-logout-link class="btn btn-primary navbar-btn">Logout</a>
    {{else}}
      {{link-to "Sign up" "signup" data-test-signup-link
        class="btn btn-primary navbar-btn"}}
      {{link-to "Login" "login" data-test-login-link
        class="btn btn-primary navbar-btn"}}
    {{/if}}
  </div>
</div>
```

That would mean we've managed to finish our first feature - sign in/sign up process following the entire TDD cycle - red/green/ refactor.

Now we can start implementing the core domain of our application - rentals management and a calendar for creating bookings.

Adding The Core Feature - Rentals' CRUD And The Calendar

Now we are getting to the most exciting part of the application. Let's break it down into some basic points to fully understand what we want to achieve here:

- Rentals (properties) can be booked for a particular period (bookings). To keep it realistic, let's assume that the minimum stay is one night.
- Rentals must have some **daily rate** defined so that it is possible to calculate the price for the booking for a given period.
- Obviously, bookings for a given rental cannot overlap with each other.
- Bookings must be associated with some client. To keep it simple, we won't introduce any new model, like `Client`, we will just have `clientEmail` attribute instead of a client relationship, which should be just enough in our case.
- Also, bookings must have **price** attribute stored on themselves - we can't safely rely on rental's **daily rate** as it might change and obviously, the price of already existing booking should stay the same.

So what we want to do here is to implement full CRUD for rentals and CRUD for bookings with a bit more complex way of creating bookings - we could just add two date fields backed by datepickers, but that would not be the best UX. A proper calendar sounds like a much better choice.

However, building a calendar sounds like an awful amount of work. How are we going to handle it?

Fortunately, there is already an addon for that! ember-power-calendar is robust and flexible and easily saves us hours of work!

That's one of the most amazing things about developing applications in Ember - not only is it an awesome framework that makes you very productive, but also the community has already created so many addons that solve quite complex problems in a generic way.

However, before adding a calendar, we need to implement a full CRUD for rentals.

Rentals' CRUD

Fortunately, with Ember, it is a pretty straight-forward task. Just like before, let's start with an acceptance test:

```
ember g acceptance-test rentals-crud
```

The first thing we will test will be "C" and "R" parts of CRUD which is creating and reading accordingly. What we initially expect to see here is empty admin page when there are no rentals created yet. The next step will be visiting some `create` page, filling forms, creating a rental and making sure the new rental that has just been created is displayed there. A good thing to do would also be to ensure the POST request is performed to `/rentals` endpoint - otherwise, we may get a false-positive and see the in-memory rental on the admin page, not the one that has been created and persisted on the server.

Before writing test let's take advantage of the awesome `resource` helper from `ember-cli-mirage` which defines all routes for CRUD actions:

```
// book-me/mirage/config.js
export default function() {
    // existing code

    this.resource('rentals');
};
```

And here's out initial test:

```
// book-me/tests/acceptance/rentals-crud-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import PageObject, {
  clickable,
  fillable,
  visitable
} from 'book-me/tests/page-object';

moduleForAcceptance('Acceptance | rentals crud');

test('it is possible to read, create, edit and delete rentals', function(assert) {
  assert.expect(3);

  const page = PageObject.create({
    visitAdmin: visitable('/admin'),
    goToNewRental: clickable(testSelector('add-rental')),
    rentalName: fillable(testSelector('rental-name')),
    rentalDailyRate: fillable(testSelector('rental-daily-rate')),
    createRental: clickable(testSelector('create-rental'))
  });

  const user = server.create('user');
  authenticateSession(this.application, { user_id: user.id });

  page.visitAdmin();

  andThen(() => {
    assert.notOk(find(testSelector('rental-row')).length, 'no rentals should be visible');
  });

  const name = 'Rental 1';
  const dailyRate = 100;

  server.post('/rentals', function (schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { name, dailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .goToNewRental()
    .rentalName(name)
    .rentalDailyRate(dailyRate)
    .createRental();

  andThen(() => {
    assert.ok(find(testSelector('rental-row')).length, 'a new rental should be visible');
  });
});
```

Not many surprises here - we are taking advantage of **test selectors** and **page objects** to make the test more expressive and simpler for both writing and reading. Not only do we interact with UI but we are also verifying if the expected request has been performed, which might not be the case if there is some validation error.

You might be wondering if we haven't written too many tests at once - shouldn't we maybe write one test, make it pass and only then write another one? That is a good question, but the answer is: it depends. In case of such simple scenario like here, there is not much risk in doing that, and I can't recall any single false-positives in similar cases in acceptance tests, so I'm confident enough to bend some TDD rules. However, writing a minimum amount of tests and then writing a minimum implementation to make those tests pass is the right default way of TDD and unless you really know what you are doing, I wouldn't recommend going against it.

To make this scenario pass let's generate the model first:

```
ember g model Rental name dailyRate
```

and provide the types of the attributes:

```
// book-me/app/models/rental.js
import DS from 'ember-data';

const {
  Model,
  attr,
} = DS;

export default Model.extend({
  name: attr('string'),
  dailyRate: attr('number')
});
```

The next step will be adding some table to `admin.hbs` template where we are going to display all the rentals and also the button for adding a new rental:

```
<!-- book-me/app/templates/admin.hbs -->
<h2>Admin</h2>

{{#link-to 'rentals.new' data-test-add-rental class='btn btn-primary'}}
  Add rental
{{/link-to}}

<table class='table table-border'>
  <thead>
    <tr>
      <th>Name</th>
      <th>Daily Rate</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {{#each rentals as |rental|}}
      <tr data-test-rental-row>
        <td>{{rental.name}}</td>
        <td>{{rental.dailyRate}}</td>
        <td></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

As we need `rentals` to be available under `rentals` property, not generic `model` property, let's do the proper adjustments in `admin` route:

```
// book-me/app/routes/admin.js
```

```
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model() {
    return this.store.findAll('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rentals', model);
  },
});
```

Another step will be generating `rentals/new` route where the actual creation of the rental is going to happen:

```
ember g route rentals/new
```

What we need to put in this route is the logic responsible for creating a new rental and some **route action** that is going to persist the rental and then transition back to `admin` route on success. Here is the route:

```
// book-me/app/routes/rentals/new.js
import Ember from 'ember';

const {
  set,
} = Ember

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('rental')
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },

  actions: {
    createRental(rental) {
      rental.save().then(() => {
        this.transitionTo('admin');
      });
    },
  },
});
```

And the last missing piece - the template:

```
<!-- book-me/app/templates/rentals/new.hbs -->
<h2>Create a new rental</h2>

<form {{action (route-action "createRental" rental) on="submit"}}>
  <div class="form-group">
    <label for="rental-name">Name</label>
    {{input
      data-test-rental-name
      id="rental-name"
      value=(mut rental.name)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="rental-dailyRate">Daily Rate</label>
    {{input
      data-test-rental-daily-rate
      id="rental-daily-rate"
      value=(mut rental.dailyRate)
      class="form-control"
      type="integer"
    }}
  </div>
  <button type="submit" class="btn btn-primary" data-test-create-rental>Create rental</butto
</form>
```

Back to green tests again!

Let's cover the "U" part of the CRUD now, which is updating the rentals. This should be quite straightforward - we just need to add some `edit` route, where the updating will happen. Again, we are going to start with a test by extending the last scenario:

```
// book-me/tests/acceptance/rentals-crud-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import PageObject, {
  clickable,
  fillable,
  visitable
} from 'book-me/tests/page-object';

moduleForAcceptance('Acceptance | rentals crud');

test('it is possible to read, create, edit and delete rentals', function(assert) {
  assert.expect(5);

  const page = PageObject.create({
    visitAdmin: visitable('/admin'),
    goToNewRental: clickable(testSelector('add-rental')),
    rentalName: fillable(testSelector('rental-name')),
    rentalDailyRate: fillable(testSelector('rental-daily-rate')),
    createRental: clickable(testSelector('create-rental')),
    goToEditRental: clickable(testSelector('edit-rental')),
    updateRental: clickable(testSelector('update-rental')),
  });

  const user = server.create('user');
  authenticateSession(this.application, { user_id: user.id });

  page.visitAdmin();

  andThen(() => {
    assert.notOk(find(testSelector('rental-row')).length,
      'no rentals should be visible');
  });

  const name = 'Rental 1';
  const dailyRate = 100;

  server.post('/rentals', function(schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { name, dailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .goToNewRental()
    .rentalName(name)
    .rentalDailyRate(dailyRate)
    .createRental();

  andThen(() => {
    assert.ok(find(testSelector('rental-row')).length, 'a new rental should be visible');
  });

  const updatedDailyRate = 200;

  server.patch('/rentals/:id', function({ rentals }, request) {
    const id = request.params.id;
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { id, name, dailyRate: updatedDailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return rentals.find(id).update(attributes);
  });

  page
    .goToEditRental()
    .rentalDailyRate(updatedDailyRate)
    .updateRental();

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'user should be redirected to admin page');
  });
});
```

Let's make the tests green again. We will start with generating new routes:

```
ember g route rental
ember g route rental/edit
```

And do the proper adjustments in the router:

```
// book-me/app/router.js
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup')
  this.route('login');
  this.route('admin');

  this.route('rentals', function() {
    this.route('new');

    this.route('rental', { path: '/rentals/:rental_id' }, function() {
      this.route('edit');
    });
  });
});

export default Router;
```

Let's add the edit link in `admin` template:

```
<!-- book-me/app/templates/admin.hbs -->
<h2>Admin</h2>

{{#link-to 'rentals.new' data-test-add-rental class='btn btn-primary'}}
  Add rental
{{/link-to}}

<table class='table table-border'>
  <thead>
    <tr>
      <th>Name</th>
      <th>Daily Rate</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {{#each rentals as |rental|}}
      <tr data-test-rental-row>
        <td>{{rental.name}}</td>
        <td>{{rental.dailyRate}}</td>
        <td>
          {{#link-to 'rental.edit' rental data-test-edit-rental class='btn btn-primary'}}
            Edit
          {{/link-to}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Now need to handle two routes: `rental` and `edit`. The former will be responsible for finding a proper rental by id, and the latter will contain the logic related to editing rentals. Here's the code for both the `rental` route and the template:

```
// book-me/app/routes/rental.js
import Ember from 'ember';

export default Ember.Route.extend({
  model(params) {
    return this.store.findRecord('rental', params.rental_id);
  },
});
```

```
<!-- book-me/app/templates/rental.hbs -->
{{outlet}}
```

And here is for `edit` route:

```
// book-me/app/routes/rental/edit.js
import Ember from 'ember';

const {
  set,
} = Ember

export default Ember.Route.extend({
  model(params) {
    return this.modelFor('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },

  actions: {
    updateRental(rental) {
      rental.save().then(() => {
        this.transitionTo('admin');
      });
    },
  },
});
```

```
<!-- book-me/app/templates/rental/edit.hbs -->
<h2>Edit rental</h2>

```

```
<form {{action (route-action "updateRental" rental) on="submit"}}>
  <div class="form-group">
    <label for="rental-name">Name</label>
    {{input
      data-test-rental-name
      id="rental-name"
      value=(mut rental.name)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="rental-dailyRate">Daily Rate</label>
    {{input
      data-test-rental-daily-rate
      id="rental-daily-rate"
      value=(mut rental.dailyRate)
      class="form-control"
      type="integer"
    }}
  </div>
  <button type="submit" class="btn btn-primary"
    data-test-update-rental>Edit Rental</button>
</form>
```

Now we are back to green again! All tests are passing.

Before moving to deleting rentals to finally finish the CRUD for rentals, we need to handle few more things.

One is that we have some duplications for handling creating and updating rentals as the logic and templates for both cases is pretty much the same. Also, it would be a good idea to add some validation, which makes it even harder argument for DRYing some code up here.

Another thing is that the new routes are not protected by the authentication requirement.

Let's handle those issues step by step.

The first step will be introducing changesets for both `create` and `update` actions. To avoid duplications, let's try first to unify `new.hbs` and `edit.hbs` templates under a new component - `rental-persistence-form`:

```
ember g component rental-persistence-form
```

The only difference between the `new` and `edit` templates are headers (which are not the parts of the form itself) and submit button. To keep things simple we can make a button configurable part from the outside - for that purpose we will take advantage of `yield` helper inside a component. To handle the actions on submit, we will generalize both `createRental` and `updateRental` actions to `persistRental` action.

The role of this action will be quite simple - it will just call the action that was passed to the component. Let's start with the component integration test:

```
// book-me/tests/integration/components/rental-persistence-form-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';

const {
  set,
} = Ember;

moduleForComponent('rental-persistence-form', 'Integration | Component |
  rental persistence form', {
  integration: true
});

test('it calls persistRental action when submitting form', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const rental = Ember.Object.create({
    id: 1,
  });

  set(this, 'rental', rental);
  set(this, 'persistRental', (rentalArgument) => {
    assert.deepEqual(rentalArgument, rental,
      'persistRental action should be called with rental argument');
  });

  this.render(hbs`
    {{#rental-persistence-form rental=rental persistRental=persistRental}}
      "<button type='submit'>Submit</button>"
    {{/rental-persistence-form}}
  `);

  $('button').click();
});
```

The test is pretty simple - we just want to make sure that after submitting the form (which in this case is triggered by clicking the button that is configurable from the outside), the proper action will be called with the right arguments.

We can now make this new test pass. Here's the component's template:

```
<!-- book-me/app/templates/components/rental-persistence-form.hbs -->
<form {{action "persistRental" on="submit"}}>
  <div class="form-group">
    <label for="rental-name">Name</label>
    {{input
      data-test-rental-name
      id="rental-name"
      value=(mut rental.name)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="rental-dailyRate">Daily Rate</label>
    {{input
      data-test-rental-daily-rate
      id="rental-daily-rate"
      value=(mut rental.dailyRate)
      class="form-control"
      type="integer"
    }}
  </div>

  {{yield}}

</form>
```

And here's the component's body where we just handle `persistRental` action:

```
// book-me/app/components/rental-persistence-form.js
import Ember from 'ember';

const {
  get,
} = Ember

export default Ember.Component.extend({
  actions: {
    persistRental() {
      const rental = get(this, 'rental');

      get(this, 'persistRental')(rental);
    },
  },
});
```

And the test is passing! Now we can easily refactor `edit.hbs` and `new.hbs`. Here is the former template after refactoring and using `rental-persistence-form` component:

```
<!-- book-me/app/templates/rental/edit.hbs -->
<h2>Edit rental</h2>

{{#rental-persistence-form persistRental=(route-action "updateRental") rental=rental}}
  <button type="submit" class="btn btn-primary"
    data-test-update-rental>Edit Rental</button>
{{/rental-persistence-form}}
```

and here is the latter:

```
<!-- book-me/app/templates/rental/edit.hbs -->
<h2>Create a new rental</h2>

{{#rental-persistence-form persistRental=(route-action "createRental") rental=rental}}
  <button type="submit" class="btn btn-primary"
    data-test-create-rental>Create rental</button>
{{/rental-persistence-form}}
```

Awesome, looks like we didn't break any tests.

Now that we've managed to unify both actions under one template, let's introduce changeset in the component:

```
// book-me/app/components/rental-persistence-form.js
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const rental = get(this, 'rental');
    const changeset = new Changeset(rental);

    set(this, 'changeset', changeset);
  },

  actions: {
    persistRental() {
      const changeset = get(this, 'changeset');

      get(this, 'persistRental')(changeset);
    },
  },
});
```

```
<!-- book-me/app/templates/components/rental-persistence-form.hbs -->
<form {{action "persistRental" on="submit"}}>
  <div class="form-group">
    <label for="rental-name">Name</label>
    {{input
      data-test-rental-name
      id="rental-name"
      value=(mut changeset.name)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="rental-dailyRate">Daily Rate</label>
    {{input
      data-test-rental-daily-rate
      id="rental-daily-rate"
      value=(mut changeset.dailyRate)
      class="form-control"
      type="integer"
    }}
  </div>

  {{yield}}

</form>
```

Introducing changeset was simple, but there is one problem though: our unit test for the component failed. But that's not a surprise - we are no longer passing there a “raw” rental but a changeset instead, so let's make the proper adjustments in the test:

```
// book-me/tests/integration/components/rental-persistence-form-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';

const {
  set,
} = Ember;

moduleForComponent('rental-persistence-form', 'Integration | Component |
  rental persistence form', {
  integration: true
});

test('it calls persistRental action when submitting form', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const rental = Ember.Object.create({
    id: 1,
  });

  set(this, 'rental', rental);
  set(this, 'persistRental', (changeset) => {
    assert.deepEqual(changeset._content, rental,
      'persistRental action should be called with rental changset');
  });

  this.render(hbs`
    {{#rental-persistence-form rental=rental persistRental=persistRental}}
      "<button type='submit'>Submit</button>"
    {{/rental-persistence-form}}
  `);

  $('button').click();
});
```

And we are back to green again.

Another step will be introducing validations which will be the same for both `create` and `update` actions. We are going to take advantage of `ember-changeset-validations`, just like for the signup process:

```
ember generate validator rental
```

We want to make sure that `name` is a required attribute and that `dailyRate` is an integer and is greater than 0. Here are out tests for such requirements:

```
// book-me/tests/unit/validators/rental-test.js
import { module, test } from 'qunit';
import validateRental from 'book-me/validators/rental';

module('Unit | Validator | rental');

test('it validates presence of name', function(assert) {
  assert.equal(validateRental.name('name', ''), "Name can't be blank");
  assert.ok(validateRental.name('name', 'Rental 1'));
});

test('it validates if dailyRate is an integer greater than 0', function(assert) {
  assert.equal(validateRental.dailyRate('dailyRate', null),
    'Daily rate must be a number');
  assert.equal(validateRental.dailyRate('dailyRate', 123.12),
    'Daily rate must be an integer');
  assert.ok(validateRental.dailyRate('dailyRate', 100));
});
```

The implementation is quite simple:

```
// book-me/app/validators/rental.js
import {
  validatePresence,
  validateNumber
} from 'ember-changeset-validations/validators';

export default {
  name: validatePresence(true),
  dailyRate: validateNumber({ integer: true, gt: 0 }),
};
```

Now we need to integrate these validators with the changeset and the rest of the component. As you may have guessed already, we will start with the test checking that the errors are displayed when the form is submitted with invalid data and that the original `persistRental` action is not called. Here it is:

```
// book-me/tests/integration/components/rental-persistence-form-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';

const {
  set,
} = Ember;

moduleForComponent('rental-persistence-form', 'Integration | Component |
  rental persistence form', {
  integration: true
});

test('it calls persistRental action when submitting form', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const rental = Ember.Object.create({
    id: 1,
  });

  set(this, 'rental', rental);
  set(this, 'persistRental', (changeset) => {
    assert.deepEqual(changeset._content, rental,
      'persistRental action should be called with rental changset');
  });

  this.render(hbs`
    {{#rental-persistence-form rental=rental persistRental=persistRental}}
      "<button type='submit'>Submit</button>"
    {{/rental-persistence-form}}
  `);

  $('button').click();
});

// new test
test('it displays validation error when the data is invalid', function(assert) {
  assert.expect(2);

  const {
    $,
  } = this;
  const rental = Ember.Object.create();

  set(this, 'rental', rental);
  set(this, 'persistRental', () => {
    throw new Error('action should not be called');
  });

  this.render(hbs`
    {{#rental-persistence-form rental=rental persistRental=persistRental}}
      "<button type='submit' data-test-submit-btn>Submit</button>"
    {{/rental-persistence-form}}
  `);

  assert.notOk($(testSelector('rental-errors')).length,
    'errors should not initially be visible')

  $(testSelector('submit-btn')).click();

  assert.ok($(testSelector('rental-errors')).length,
    'errors should be visible when submitting form with invalid data');
});
```

To make it pass need to do the proper adjustments in the component:

```
// book-me/app/components/rental-persistence-form.js
import Ember from 'ember';
import Changeset from 'ember-changeset';
import lookupValidator from 'ember-changeset-validations';
import RentalValidators from 'book-me/validators/rental';

const {
  get,
  set,
} = Ember

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const rental = get(this, 'rental');
    const changeset = new Changeset(rental, lookupValidator(RentalValidators),
      RentalValidators);

    set(this, 'changeset', changeset);
  },

  actions: {
    persistRental() {
      const changeset = get(this, 'changeset');

      changeset.validate().then(() => {
        if (get(changeset, 'isValid')) {
          get(this, 'persistRental')(changeset);
        }
      });
    },
  },
});
```

and in the template:

```
<!-- book-me/app/templates/components/rental-persistence-form.hbs -->
{{#if changeset.isInvalid}}
  <section data-test-rental-errors>
    {{#each changeset.errors as |error|}}
      <div class="alert alert-danger" role="alert">
      {{error.validation}}
    </div>
  {{/each}}
</section>
{{/if}}

<form {{action "persistRental" on="submit"}}>
  <div class="form-group">
    <label for="rental-name">Name</label>
    {{input
      data-test-rental-name
      id="rental-name"
      value=(mut changeset.name)
      class="form-control"
    }}
  </div>
  <div class="form-group">
    <label for="rental-dailyRate">Daily Rate</label>
    {{input
      data-test-rental-daily-rate
      id="rental-daily-rate"
      value=(mut changeset.dailyRate)
      class="form-control"
      type="integer"
    }}
  </div>

  {{yield}}

</form>
```

So now we should be back to green, right?

Well, not exactly. Our new test is passing, but the previous test is not! But that makes sense - we don't currently fill any input with any data there, so the fact that it's failing is just a sign of a good test suite. Let's do the adjustments there and make this test pass:

```
// book-me/tests/integration/components/rental-persistence-form-test.js
test('it calls persistRental action when submitting form when the data
  is valid', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const rental = Ember.Object.create({
    id: 1,
  });

  set(this, 'rental', rental);
  set(this, 'persistRental', (changeset) => {
    assert.deepEqual(changeset._content, rental,
      'persistRental action should be called with rental changset');
  });

  this.render(hbs`
    {{#rental-persistence-form rental=rental persistRental=persistRental}}
      "<button type='submit' data-test-submit-btn>Submit</button>"
    {{/rental-persistence-form}}
  `);

  $(testSelector('rental-name')).val('Rental 1').change();
  $(testSelector('rental-daily-rate')).val(100).change();

  $(testSelector('submit-btn')).click();
});
```

And we are back to green!

What about server-side errors? Ideally, the UI validations would cover all the possible cases, but sometimes it is not feasible to do it perfectly (e.g., uniqueness validation) or some use case might be just overlooked, so it's always a good idea to handle error messages coming from the server.

The natural way to handle it is to start with the tests. But the questions is - on what level? In this case it would be ideally acceptance test as multiple layers are going to be involved and checking if the errors are displayed in the UI is the safest way to verify it, but on the other hand those tests are much slower than unit tests, which would be the alternative here to acceptance tests as we could just handle it by testing route actions. The extra benefit of the unit tests here is that we could test for details in isolation.

To make it simpler let's write both unit and acceptance tests to get an idea how it may look like and later decide which way is better. We will start with acceptance tests.

Before writing new tests as the part of `Rentals CRUD` scenario, let's move the authentication logic to `beforeEach` hook and extract **page object** so that we make the tests more DRY and page object reusable in all scenarios.

Let's generate `rentals` page:

```
ember generate page-object rentals
```

And move the logic there from `rentals-crud-test.js` test:

```
// book-me/tests/pages/rentals.js
import {
  create,
  clickable,
  fillable,
  visitable,
} from 'ember-cli-page-object';
import testSelector from 'ember-test-selectors';

export default create({
  visitAdmin: visitable('/admin'),
  goToNewRental: clickable(testSelector('add-rental')),
  rentalName: fillable(testSelector('rental-name')),
  rentalDailyRate: fillable(testSelector('rental-daily-rate')),
  createRental: clickable(testSelector('create-rental')),
  goToEditRental: clickable(testSelector('edit-rental')),
  updateRental: clickable(testSelector('update-rental')),
});
```

And here is our test file after refactoring:

```
// book-me/tests/acceptance/rentals-crud-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import page from 'book-me/tests/pages/rentals';

moduleForAcceptance('Acceptance | rentals crud', {
  beforeEach() {
    const user = server.create('user');
    authenticateSession(this.application, { user_id: user.id });
  },
});

test('it is possible to read, create, edit and delete rentals', function(assert) {
  assert.expect(5);

  page.visitAdmin();

  andThen(() => {
    assert.notOk(find(testSelector('rental-row')).length,
      'no rentals should be visible');
  });

  const name = 'Rental 1';
  const dailyRate = 100;

  server.post('/rentals', function(schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { name, dailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .goToNewRental()
    .rentalName(name)
    .rentalDailyRate(dailyRate)
    .createRental();

  andThen(() => {
    assert.ok(find(testSelector('rental-row')).length,
      'a new rental should be visible');
  });

  const updatedDailyRate = 200;

  server.patch('/rentals/:id', function({ rentals }, request) {
    const id = request.params.id;
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { id, name, dailyRate: updatedDailyRate };

    assert.deepEqual(attributes, expectedAttributes, "attributes don't match the expected o

    return rentals.find(id).update(attributes);
  });

  page
    .goToEditRental()
    .rentalDailyRate(updatedDailyRate)
    .updateRental();

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'user should be redirected to admin page');
  });
});
```

All tests are still passing, so we managed to not break anything. Let's write two new tests: one for handling server-side error messages when creating a new rental and one for updating a rental. The idea is simple - we want to make sure the error messages are displayed, even when passing the client-side validations. Here are our two tests:

```
// book-me/tests/acceptance/rentals-crud-test.js
// new import
import Mirage from 'ember-cli-mirage';

const {
  Response,
} = Mirage;

// 2 new tests:
test('it displays server-side validation errors when creating new rental',
  function(assert) {
  assert.expect(2);

  server.post('/rentals', () => {
    const errors = {
      errors: [
        {
          detail: 'is already taken',
          source: {
            pointer: 'data/attributes/name'
          }
        }
      ]
    };
    return new Response(422, {}, errors);
  });

  page
    .visitAdmin()
    .goToNewRental()
    .rentalName('name')
    .rentalDailyRate(100)
    .createRental();

  andThen(() => {
    assert.equal(currentPath(), 'rentals.new', 'user should stay on new rental page');
    assert.ok(find(testSelector('rental-errors')).length,
      'errors should be visible when submitting form with invalid data');
  });
});

```

```
test('it displays server-side validation errors when updating rental',
  function(assert) {
  assert.expect(2);

  server.create('rental', { name: 'name', dailyRate: 20 });

  server.patch('/rentals/:id', () => {
    const errors = {
      errors: [
        {
          detail: 'is already taken',
          source: {
            pointer: 'data/attributes/name'
          }
        }
      ]
    };
    return new Response(422, {}, errors);
  });

  page
    .visitAdmin()
    .goToEditRental()
    .rentalName('updated name')
    .rentalDailyRate(100)
    .updateRental();

  andThen(() => {
    assert.equal(currentPath(), 'rental.edit',
      'user should stay on edit rental page');
    assert.ok(find(testSelector('rental-errors')).length,
      'errors should be visible when submitting form with invalid data');
  });
});
```

Although first part of our tests is fine - we verified that we won't be redirected to `admin` route, the second part, checking if validation errors are displayed, is failing which is a very good thing as seeing this part fail is the way to verify that we are not confusing client-side errors with server-side errors.

To make these tests pass we just need to copy validation errors from model's errors to changeset in `edit` and `new` routes. We need to keep in mind that the formatted validation message that we display in a component comes from error's `validation` property, so it's essential to populate this attribute:

```
// book-me/app/routes/rentals/new.js
import Ember from 'ember';
```

```

const {
  get,
  set,
} = Ember

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('rental')
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },

  actions: {
    createRental(changeset) {
      changeset.save().then(() => {
        this.transitionTo('admin');
      }).catch(() => {
        const errors = get(changeset._content, 'errors')

        errors.forEach(error => {
          const key = error.attribute;
          const message = error.message;

          changeset.addError(key, { validation: `${key} ${message}` });
        });
      });
    },
  },
});
```

```
// book-me/app/routes/rental/edit.js
import Ember from 'ember';

const {
  get,
  set,
} = Ember

export default Ember.Route.extend({
  model() {
    return this.modelFor('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },

  actions: {
    updateRental(changeset) {
      changeset.save().then(() => {
        this.transitionTo('admin');
      }).catch(() => {
        const errors = get(changeset._content, 'errors')

        errors.forEach(error => {
          const key = error.attribute;
          const message = error.message;

          changeset.addError(key, { validation: `${key} ${message}` });
        });
      });
    },
  },
});
```

And we are back to green tests again! Let's cover the same use case with unit tests for route actions. As both `edit` and `new` routes are almost the same, the tests won't differ that much. The idea is simple: we just need to check if the changeset errors are indeed populated with model errors when the `save` fails (i.e., when the promise is rejected).

Here's how we can approach it:

```
// book-me/tests/unit/routes/rentals/new-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  RSVP,
  run,
} = Ember

moduleFor('route:rentals/new', 'Unit | Route | rentals/new', {
});
```

```

test('changeset gets populated with model errors in `createRental` action
  when there is an error on backend', function(assert) {
  assert.expect(2)

  const route = this.subject();
  const createRental = route.actions.createRental;

  const errors = [
    {
      attribute: 'name',
      message: 'is invalid',
    }
  ];
  const rentalStub = Ember.Object.extend({
    save() {
      return RSVP.reject();
    },
  }).create({ errors });
  const changeset = new Changeset(rentalStub);

  run(createRental.bind(route, changeset));

  assert.ok(get(changeset, 'isInvalid'), 'changeset should be invalid');

  const expectedErrors = [
    {
      key: 'name',
      validation: 'name is invalid',
    }
  ];
  assert.deepEqual(get(changeset, 'errors'), expectedErrors,
    'changeset should be populated with errors');
});
```

```
// book-me/tests/unit/routes/rental/edit-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  RSVP,
  run,
} = Ember

```

```
moduleFor('route:rental/edit', 'Unit | Route | rental/edit', {
});

test('changeset gets populated with model errors in `updateRental`
  action when there is an error on backend', function(assert) {
  assert.expect(2)

  const route = this.subject();
  const updateRental = route.actions.updateRental;

  const errors = [
    {
      attribute: 'name',
      message: 'is invalid',
    }
  ];
  const rentalStub = Ember.Object.extend({
    save() {
      return RSVP.reject();
    },
  }).create({ errors });
  const changeset = new Changeset(rentalStub);

  run(updateRental.bind(route, changeset));

  assert.ok(get(changeset, 'isInvalid'), 'changeset should be invalid');

  const expectedErrors = [
    {
      key: 'name',
      validation: 'name is invalid',
    }
  ];
  assert.deepEqual(get(changeset, 'errors'), expectedErrors,
    'changeset should be populated with errors');
});
```

There are some interesting patterns used in those two examples. To grab a specific action from a route, we can access its `actions` property and then fetch the action we need. Another step is creating a rental stub with a `save` method that merely returns rejected promise, which is exactly what we need to handle the error flow. We also populate rental stub with some `errors` messages. Then, we run the actual action in `Ember.run`, which is quite important here as we are dealing with promises and that way we will make sure that the assertions are not being run before the promise is fulfilled. To make sure `this` will be the expected context in the action when running tests, we need to take advantage of `bind` and pass the actual `route` as the first argument; otherwise, everything that calls `this` in the route action will fail. The last thing is the assertions - we obviously want to make sure the changeset is invalid and that it contains the proper validation messages.

Before moving to deleting rentals, we have one more thing to handle - making sure `edit` and `new` routes require authentication.

Adding acceptance tests to cover those cases might be too heavy. Ideally, we would cover this case with unit tests.

The best way to handle this case will be to check if `AuthenticatedRouteMixin` is included. After a quick look at the mixin, we can see that `beforeModel` hook uses `session` service and its `isAuthenticated` property to check if we are authenticated or not. To make sure the routes are protected, we can just make sure that this property is called when executing `beforeModel` hook. Here are the tests:

```
// book-me/tests/unit/routes/rentals/new-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  RSVP,
  run,
  computed,
} = Ember

moduleFor('route:rentals/new', 'Unit | Route | rentals/new', {
});

test('it requires authentication', function(assert) {
  assert.expect(1);

  const sessionStub = Ember.Service.extend({
    isAuthenticated: computed(() => {
      assert.ok(true, 'isAuthenticated has to be used for checking authentication');

      return true;
    }),
  });

  this.register('service:session', sessionStub);
  this.inject.service('session');

  const route = this.subject();

  route.beforeModel();
});
```

```
// book-me/tests/unit/routes/rental/edit-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';
import Changeset from 'ember-changeset';

const {
  get,
  RSVP,
  run,
  computed,
} = Ember

moduleFor('route:rental/edit', 'Unit | Route | rental/edit', {
});

test('it requires authentication', function(assert) {
  assert.expect(1);

  const sessionStub = Ember.Service.extend({
    isAuthenticated: computed(() => {
      assert.ok(true, 'isAuthenticated has to be used for checking authentication');

      return true;
    }),
  });

  this.register('service:session', sessionStub);
  this.inject.service('session');

  const route = this.subject();

  route.beforeModel();
});
```

The interesting thing here is that we are providing a session stub instead of using a real service with `isAuthenticated` computed property where we are making assertion just to make sure this property is called. Next, we register the stub as `service:session` and inject the service. In the end, we are just calling `beforeModel()` method. Thanks to `assert.expect(1)`, such test is enough here - if `isAuthenticated` property doesn't get called, it will fail.

And here is the implementation to make those tests pass:

```
// book-me/app/routes/new.js
```

```
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  get,
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
    // the rest of the code
});
```

```
// book-me/app/routes/new.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  get,
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
    // the rest of the code
});
```

```
// book-me/app/routes/edit.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  get,
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
    // the rest of the code
});
```

But not the all tests are green! Apparently, we've just broken the tests for route actions, and we are getting `Attempting to inject an unknown injection: 'service:session'` error. To fix this, we just need to take advantage of `needs` property and provide the missing dependencies:

```
// book-me/tests/unit/routes/rental/edit-test.js
moduleFor('route:rental/edit', 'Unit | Route | rental/edit', {
  needs: ['service:session'],
});
```

```
// book-me/tests/unit/routes/rentals/new-test.js
moduleFor('route:rentals/new', 'Unit | Route | rentals/new', {
  needs: ['service:session'],
});
```

Now we are back to green; all tests are passing.

Now, it's time for the final part of Rentals' CRUD - adding a possibility to delete rentals.

When adding a delete button, it's worth keeping in mind some usability aspects and providing good UX - it's quite easy to accidentally click on the button and remove something that was not supposed to be removed.

One easy solution is to possibly display a confirmation alert and make user users confirm the action before deleting anything. That would certainly get the job done, but it's not that pretty from the UX perspective. Fortunately, we can do much better than this, and it is quite simple to implement - we could just make the user hold a button for a particular period, e.g., 3 seconds and only after 3 seconds will the rental be deleted. Holding it for a shorter period would not have any effect.

Apparently, we are quite lucky since there is already some addon that provides such solution - ember-hold-button.

We can now write an acceptance tests. What we want to test is that after holding the button, rental will no longer be in the UI and that `DELETE` request will be performed to `rentals/:id` endpoint. Here's the test:

```
// book-me/tests/acceptance/rentals-crud-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import page from 'book-me/tests/pages/rentals';
import Mirage from 'ember-cli-mirage';

const {
  Response,
} = Mirage;

moduleForAcceptance('Acceptance | rentals crud', {
  beforeEach() {
    const user = server.create('user');
    authenticateSession(this.application, { user_id: user.id });
  },
});

```

```
test('it is possible to read, create, edit and delete rentals', function(assert) {
  assert.expect(7);

  page.visitAdmin();

  andThen(() => {
    assert.notOk(find(testSelector('rental-row')).length,
      'no rentals should be visible');
  });

  const name = 'Rental 1';
  const dailyRate = 100;

  server.post('/rentals', function(schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { name, dailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .goToNewRental()
    .rentalName(name)
    .rentalDailyRate(dailyRate)
    .createRental();

  andThen(() => {
    assert.ok(find(testSelector('rental-row')).length,
      'a new rental should be visible');
  });

  const updatedDailyRate = 200;

  server.patch('/rentals/:id', function({ rentals }, request) {
    const id = request.params.id;
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = { id, name, dailyRate: updatedDailyRate };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return rentals.find(id).update(attributes);
  });

  page
    .goToEditRental()
    .rentalDailyRate(updatedDailyRate)
    .updateRental();

  andThen(() => {
    assert.equal(currentPath(), 'admin', 'user should be redirected to admin page');
  });

  // new stuff for testing deleting the rentals

  server.del('/rentals/:id', function({ rentals }, request) {
    const id = request.params.id;

    assert.ok(true, 'rental should be destroyed')

    rentals.find(id).destroy();
  });

  page.deleteRental(); // not implemented yet

  andThen(() => {
    assert.notOk(find(testSelector('rental-row')).length,
      'no rentals should be visible');
  });
});
```

The next step will be implementing `deleteRental` on Rentals page object. But how should we handle the interaction with the button? Obviously, it's not just a simple click.

To solve that problem we should break the problem down into the single events. On a non-mobile device, pressing a button means triggering **mouseDown** event and releasing means triggering **mouseUp** event. On mobile that would be **touchStart** and **touchEnd** events accordingly.

Based on how **hold-button** component works, we may suspect that there is some internal timer which starts counting time after triggering **mouseDown** (**touchStart**) event or a scheduler which executes the action if it was held for required amount of time and cancels it if it was released before that period, which would mean cancelling timer on **mouseUp** event.

After checking the internals of the addon, it turns out this is exactly the case! We don't have to care about **mouseUp** part; we just need to make sure that `mouseDown` event is triggered on the button. Fortunately, it is quite easy to achieve with `ember-cli-page-object` - we just need to take advantage of `triggerable` helper. Here's how our page object is going to look like now:

```js
// book-me/tests/pages/rentals.js
import {
  create,
  clickable,
  fillable,
  visitable,
  triggerable,
} from 'ember-cli-page-object';
import testSelector from 'ember-test-selectors';

export default create({
  visitAdmin: visitable('/admin'),
  goToNewRental: clickable(testSelector('add-rental')),
  rentalName: fillable(testSelector('rental-name')),
  rentalDailyRate: fillable(testSelector('rental-daily-rate')),
  createRental: clickable(testSelector('create-rental')),
  goToEditRental: clickable(testSelector('edit-rental')),
  updateRental: clickable(testSelector('update-rental')),
  deleteRental: triggerable('mousedown', testSelector('delete-rental')),
});
```

Now that we see the new scenario failing let's move to the actual implementation. We will start with installing the addon:

```
ember install ember-hold-button
```

We need to add the button to `admin.hbs` template and implement a route action, let's call it `delete-rental`, that will be responsible for deleting rentals. Here's the template:

```hbs
<!-- book-me/app/templates/admin.hbs -->
<h2>Admin</h2>

{{#link-to 'rentals.new' data-test-add-rental class='btn btn-primary'}}
  Add rental
{{/link-to}}

<table class='table table-border'>
  <thead>
    <tr>
      <th>Name</th>
      <th>Daily Rate</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {{#each rentals as |rental|}}
      <tr data-test-rental-row>
```

```
        <td>{{rental.name}}</td>
        <td>{{rental.dailyRate}}</td>
        <td>
          {{#link-to 'rental.edit' rental data-test-edit-rental class='btn btn-primary'}}
            Edit
          {{/link-to}}

          {{#hold-button action=(route-action 'deleteRental' rental) delay=3000
            class='btn' data-test-delete-rental=rental.id}}
            Delete
          {{/hold-button}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Holding the button for 3 seconds will trigger `deleteRental` action. Here is its implementation in `admin` route:

```
// book-me/app/routes/admin.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model() {
    return this.store.findAll('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rentals', model);
  },

  actions: {
    deleteRental(rental) {
      rental.destroyRecord();
    },
  },
});
```

And again, all tests are passing! We've managed to implement the full CRUD

TDD-style. However, there are a couple of things that be improved.

The first one is that we may want to delete rental in multiple places later, not only in `admin`. In such case, we may want to have the entire process of deleting rental encapsulated in a component that would be exclusively responsible for deleting rentals. Also, that would be a good use case to learn how to test such logic in a component integration test ;).

Another thing is that holding a button for 3 seconds in test suite makes it last 3 seconds longer, which is quite long. Ideally, we would have some kind of helper that in the non-test environment would return the “real” delay time and in the test environment, it would return `0` to make it possibly fast.

We will certainly need to get back to that issue but for now, let’s focus now on a component that we will call `delete-rental-button`:

```
ember g component delete-rental-button
```

As you may have already guessed, we will start with the integration test. The idea is simple: we need to make sure that after holding a button for a certain time the rental will get deleted. The test itself is not that obvious, however:

```
// book-me/tests/integration/components/delete-rental-button-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';
import wait from 'ember-test-helpers/wait';

const {
  set,
  RSVP,
} = Ember;

moduleForComponent('delete-rental-button', 'Integration | Component |
  delete rental button', {
  integration: true
});

test('it deletes rental by holding a button', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;

  const rentalStub = Ember.Object.extend({
    destroyRecord() {
      assert.ok(true, 'item should be destroyed');

      return RSVP.resolve(this);
    },
  }).create({ id: 1 });

  set(this, 'rental', rentalStub);

  this.render(hbs`{{delete-rental-button rental=rental}}`);

  const $deleteBtn = $(testSelector('delete-rental'));
  const done = assert.async();

  $deleteBtn.mousedown();

  wait().then(() => {
    done();
  });
});
```

Even though there is only one assertion, there are few interesting things going on in the test. The beginning is quite simple - we create a rental stub and define `destroyRecord` method where we do the actual assertion that this method was called and we render the component and find `deleteBtn`, that is simple. But then we are using `async() / done()` functions, triggering `mouseDown` event on the button and using `wait()` helper. What are the purposes of those functions?

- `async()/done()` – To make sure QUnit will wait for an asynchronous operation to be finished, we need to use `async()` function. That way QUnit will wait until `done()` is called.
- `wait()` – it forces run loop to process all the pending events. That way we ensure that the asynchronous operation has been executed (like calling `deleteRental` action after 3 seconds. When all the events have been processed, we can call `done()`.

It might sound a bit complex, but after writing several of such tests, such flow becomes quite simple and obvious.

Here’s the implementation of the component:

```
// book-me/app/components/delete-rental-button.js
import Ember from 'ember';

const {
  get,
} = Ember;

export default Ember.Component.extend({
  delay: 3000,

  actions: {
    deleteRental() {
      const rental = get(this, 'rental');

      rental.destroyRecord();
    },
  },
});
```

and its body:

```
<!--book-me/app/templates/components/delete-rental-button.hbs -->
{{#hold-button action="deleteRental" delay=delay class='btn'
  data-test-delete-rental=rental.id}}
  Delete
{{/hold-button}}
```

And that way we managed to get the component's test to pass. Now, let's get back to the idea of making the tests faster:

We will implement a utility function called `resolveDelay`. We are going to start with a test checking that whatever value we pass, we will get `0` as a result. Since the behavior is going to be environment-dependent, there is not much we can do for testing it for other environments, so it might be worth testing in UI via the real interaction in such case.

Here's the test:

```
// book-me/tests/unit/utilities/resolve-delay-test.js
import resolveDelay from 'book-me/utilities/resolve-delay';
import { module, test } from 'qunit';

module('Unit | Utility | resolve delay');

test('it returns 0 for every value in test env', function(assert) {
  assert.equal(resolveDelay(42), 0);
  assert.equal(resolveDelay(0), 0);
  assert.equal(resolveDelay(3000), 0);
});
```

And here's the implementation:

```
// book-me/app/utilities/resolve-delay.js
import config from 'book-me/config/environment';

export default function resolveDelay(delay) {
  if (config.environment === 'test') {
    return 0;
  } else {
    return delay;
  }
}
```

We can apply it now in our `delete-rental-button` component:

```
// book-me/app/components/delete-rental-button.js
import Ember from 'ember';
import resolveDelay from 'book-me/utilities/resolve-delay';

const {
  get,
} = Ember;

export default Ember.Component.extend({
  delay: resolveDelay(3000),

  actions: {
    deleteRental() {
      const rental = get(this, 'rental');

      rental.destroyRecord();
    },
  },
});
```

All tests are still passing so nothing got broken. We can now modify `admin.hbs` template and take advantage of `delete-rental-button` component:

```
<!--book-me/app/templates/admin.hbs -->
<h2>Admin</h2>

{{#link-to 'rentals.new' data-test-add-rental class='btn btn-primary'}}
  Add rental
{{/link-to}}

<table class='table table-border'>
  <thead>
    <tr>
      <th>Name</th>
      <th>Daily Rate</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {{#each rentals as |rental|}}
      <tr data-test-rental-row>
        <td>{{rental.name}}</td>
        <td>{{rental.dailyRate}}</td>
        <td>
          {{#link-to 'rental.edit' rental data-test-edit-rental class='btn btn-primary'}}
            Edit
          {{/link-to}}

          {{delete-rental-button rental=rental}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

And our test suite has just become much faster ;). We can also remove the route action since it's not used anymore:

```
// book-me/app/routes/admin.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model() {
    return this.store.findAll('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rentals', model);
  },
});
```

As the last improvement, let's display some notification when the rental gets deleted, which would bring more pleasant UX. There is a great addon that will be very helpful here called **ember-notify**, let's install it:

```
ember install ember-notify
```

The setup is quite straightforward, we just need to display `ember-notify` component in `application.hbs` template:

```
<!-- book-me/app/templates/application.hbs -->
<nav class="navbar navbar-default navbar-fixed-top" role="navigation">
  <div class="container-fluid">
    <div class="navbar-header">
```

```
      <button type="button" class="navbar-toggle navbar-toggle-context"
              data-toggle="collapse" data-target=".navbar-top-collapse">
        <span class="sr-only">Toggle Navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <div class="navbar-brand-container">
        <span class="navbar-brand">
          <h1><i class="fa fa-star"></i> Book Me!</h1>
        </span>
      </div>
    </div>
    <div class="collapse navbar-collapse navbar-top-collapse">
      <div class="navbar-right">
        {{#if session.isAuthenticated}}
          <a {{action (route-action 'logOut')}} data-test-logout-link
            class="btn btn-primary navbar-btn">Logout</a>
        {{else}}
          {{link-to "Sign up" "signup" data-test-signup-link
            class="btn btn-primary navbar-btn"}}
          {{link-to "Login" "login" data-test-login-link
            class="btn btn-primary navbar-btn"}}
        {{/if}}
      </div>
    </div>
  </div>
</nav>
<section class="main-content">
  <div class="sheet">
    {{outlet}}
  </div>
</section>

{{ember-notify messageStyle='bootstrap'}}
```

Writing acceptance tests checking if the notification is displayed would be too much - displaying a notification after a rental gets deleted is not a business-critical feature. It's much better to just check in the UI if the notifications are displayed nicely and then, only write integration or unit tests.

Since we want to display some notification after a rental gets deleted, let's extend the integration test for `delete-rental-button` component:

```
// book-me/tests/integration/component/delete-rental-button-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';
import wait from 'ember-test-helpers/wait';

const {
  set,
  RSVP,
} = Ember;

moduleForComponent('delete-rental-button', 'Integration | Component |
  delete rental button', {
  integration: true
});

test('it deletes rental by holding a button', function(assert) {
  assert.expect(2);

  const {
    $,
  } = this;

  const notifyStub = Ember.Service.extend({
    info() {
      assert.ok(true, 'notification should be displayed');
    },
  });

  this.register('service:notify', notifyStub);
  this.inject.service('notify');

  const rentalStub = Ember.Object.extend({
    destroyRecord() {
      assert.ok(true, 'item should be destroyed');

      return RSVP.resolve(this);
    },
  }).create({ id: 1 });

  set(this, 'rental', rentalStub);

  this.render(hbs`{{delete-rental-button rental=rental}}`);

  const $deleteBtn = $(testSelector('delete-rental'));
  const done = assert.async();

  $deleteBtn.mousedown();

  wait().then(() => {
    done();
  });
});
```

We are creating a stub of the `notify` service where we do perform another assertion in `info` method to make sure it gets called; we are registering this service and injecting it. The implementation to make this test pass is going to be quite straightforward:

```
// book-me/app/components/delete-rental-button.js
import Ember from 'ember';
import resolveDelay from 'book-me/utilities/resolve-delay';

const {
  get,
  inject,
} = Ember;

export default Ember.Component.extend({
  delay: resolveDelay(3000),
  notify: inject.service(),

  actions: {
    deleteRental() {
      const rental = get(this, 'rental');

      rental.destroyRecord().then(() => {
        const notify = get(this, 'notify');

        notify.info('Rental is deleted');
      });
    },
  },
});
```

And all tests are again green!

That would be all for rentals' CRUD. There are obviously few more things that could make a nice improvement - we could, for example, disable submit buttons until the changeset is valid, we could implement rollback / reset buttons or delete a rental from memory in `rentals/new` route when exiting the route without persisting the rental, but at this point I'm pretty sure you will be able to TDD those features without any problems :).

If you are curious about the visual side of the features we delivered, here are some screenshots:

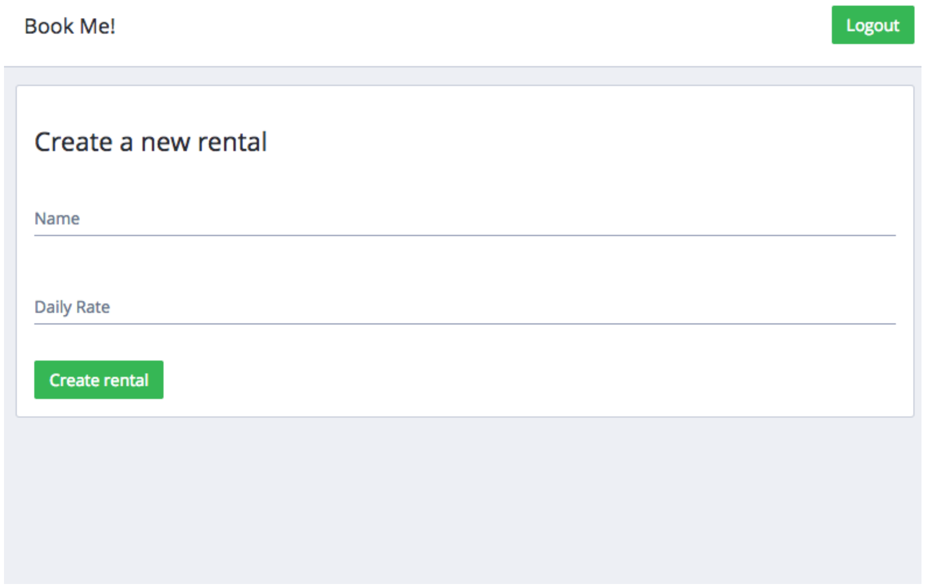
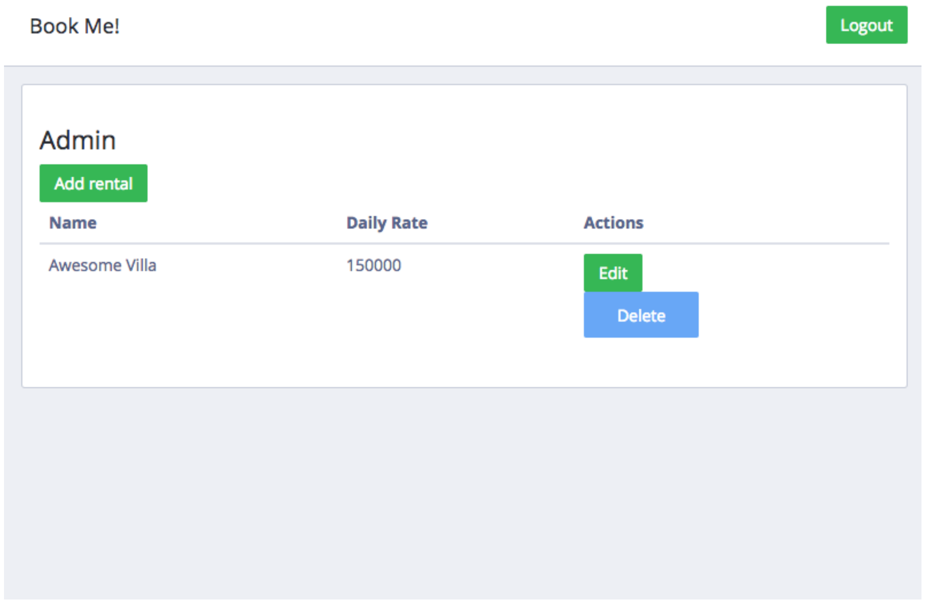


Figure 8: Create rental



Name	Daily Rate	Actions
Awesome Villa	150000	Edit Delete

Figure 9: Admin

As you can see the `Delete` button doesn't look perfect, but TDD-ing CSS is outside of the scope of this book ;).

Let's move to the final and the most interesting feature we are going to implement: the calendar.

The Calendar

As already discussed earlier, we are going to take advantage of `ember-power-calendar`. Let's install this powerful addon now:

```
ember install ember-power-calendar
```

The addon comes also with some stylesheets. To make the calendars look better we can include them as well:

```
/* book-me/app/styles/app.scss */
@import "ember-power-select";
@import "ember-modal-dialog/ember-modal-structure";
@import "bootstrap-bookingsync";
@import "ember-power-calendar";
```

However, before moving on to playing with a calendar, let's populate in-memory "database" for development to make the interaction with app better instead of creating all the rentals manually.

We will start with generating an ember-cli-mirage `rental factory`:

```
ember g mirage-factory rental
```

If we are already on generating factories, we may generate one for user model to be able to use the sign in right away when opening the app, without signing up in the first place to create a user:

```
ember g mirage-factory user
```

For rentals, we need to have some unique and random `name`s and some integer `daily rate`s. We will take advantage of sequence number which is an optional argument for attributes and use `faker` to generate random numbers for rates:

```
// book-me/mirage/factories/rental.js
import { Factory, faker } from 'ember-cli-mirage';

export default Factory.extend({
  name(i) {
    return `Rental ${i}`;
  },

  dailyRate() {
    return faker.random.number();
  }
});
```

We are going to have a pretty similar setup for `user`s with unique `email`s and hardcoded `password`s, just to make it easier to sign in:

```
// book-me/mirage/factories/user.js
import { Factory } from 'ember-cli-mirage';

export default Factory.extend({
  email(i) {
    return `example_${i}@gmail.com`;
  },

  password() {
    return `password123`;
  }
});
```

Since we want to have the rentals and users available right away during the development, let's seed the `database` with 10 `rentals` and one `user`:

```
// book-me/mirage/scenarios/default.js
export default function(server) {
  server.createList('rental', 10);

  server.create('user', { email: 'email@example.com', password: 'password123' });
}
```

Now that we've made a helpful setup, we can focus on the calendar itself and what we want to do with it. There is one particularly interesting section in the docs of the addon about range selection which is exactly what we need - the booking will have `beginsAt` and `finishesAt` attributes which can be easily populated from the selected range. To make it possible, we will just display a calendar in `rental/show` template where we will be able to select the dates for a new booking.

Another thing to consider is: what is going to happen next? We also need to provide input for `clientEmail` attribute. It would also be nice to display the actual length of stay for the booking and its price that is going to be calculated based on this length of stay and rental's `dailyRate` value.

To achieve all those things and to provide a nice UX, we are going to display a modal after selecting the range on a calendar with the form to fill other fields and display the required info.

An excellent choice for modals in Ember ecosystem is `ember-modal-dialog` which provides flexible API and is easy to use. And it's based on `ember-wormhole` addon, which has a pretty awesome name. Let's install the addon now:

```
ember install ember-modal-dialog
```

To make the UI nice we can also add some stylesheets provided by the addon:

```
/* book-me/app/styles/app.scss */
@import "ember-power-select";
```

```
@import "ember-modal-dialog/ember-modal-structure";
@import "bootstrap-bookingsync";
@import "ember-power-calendar";
@import "ember-modal-dialog/ember-modal-structure";
@import "ember-modal-dialog/ember-modal-appearance";
```

Just like with every feature we've implemented so far, we are going to start with an acceptance test. Initially, we will keep it pretty simple: what we want to test is that we can go to admin page, click on the link to go to `rental/show` route, select same dates range on calendar, fill in `clientEmail`, click some button and verify that a new booking gets displayed and that it actually gets created by checking the outgoing request and its payload. Let's generate a new acceptance test:

```
ember g acceptance-test create-booking
```

The next step would be writing a test cover the just mentioned scenario:

```
// book-me/tests/acceptance/create-booking-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import page from 'book-me/tests/pages/create-booking';
import moment from 'moment';

moduleForAcceptance('Acceptance | create booking', {
  beforeEach() {
    const user = server.create('user');
    authenticateSession(this.application, { user_id: user.id });
  },
});

test('creating a booking for rental', function(assert) {
  assert.expect(3);

  const dailyRate = 100;
  const rental = server.create('rental', { dailyRate });
  const clientEmail = 'client@email.com';
  const today = moment();
  const currentMonth = today.month() + 1; // month are indexed starting from 0
  const startDay = 10;
  const endDay = 20;
  const price = (endDay - startDay) * dailyRate;

  page
    .visitAdmin()
    .goToRentalPage();

  andThen(() => {
    assert.notOk(find(testSelector('booking-row')).length, 'no bookings should be visible')
  });

  server.post('/bookings', function(schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = {
      rentalId: rental.id,
      clientEmail,
      price,
      beginsAt: `2017-0${currentMonth}-${startDay}T14:00:00.000Z`,
      finishesAt: `2017-0${currentMonth}-${endDay}T10:00:00.000Z`,
    };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .selectStartDate(startDay)
    .selectEndDate(endDay)
    .fillInClientEmail(clientEmail)
    .createNewBooking();

  andThen(() => {
    assert.ok(find(testSelector('booking-row')).length,
      'bookings should be visible');
  });
});
```

There are some interesting things going on in this test. The beginning is pretty straight-forward - authentication of the user and variables' setup. However, there is one surprise when it comes to the months - in JavaScript, months are indexed from **0**! So, e.g., January will not be `1` but `0` instead! That's why we are adding +1, just to have some reasonable number standing for the current month. Then, we are taking advantage of page object (which is not implemented yet) - we are visiting admin page and going to a rental page. In next step, we are verifying that no booking has been created or loaded yet. Another step is quite similar to what we did in other acceptance tests - we are intercepting a request and verifying if the params sent to `bookings` endpoint match the expected ones and simply creating a booking, just like the original implementation of the route handler from `ember-cli-mirage` does. What we are doing next is selecting start and end dates, filling in client's email and creating a booking, which means simply submitting a form. The last step is checking that the new booking is displayed.

Now let's generate the page object that we are using here since it's not implemented yet:

```
ember generate page-object create-booking
```

Here is its implementation:

```
// book-me/tests/pages/create-booking.js
import {
  create,
  clickable,
  fillable,
  visitable,
  clickOnText,
} from 'ember-cli-page-object';
import testSelector from 'ember-test-selectors';

export default create({
  visitAdmin: visitable('/admin'),
  goToRentalPage: clickable(testSelector('show-rental')),
  selectStartDate: clickOnText('button',
    { scope: testSelector('new-booking-calendar') }),
  selectEndDate: clickOnText('button',
    { scope: testSelector('new-booking-calendar') }),
  fillInClientEmail: fillable(testSelector('booking-client-email')),
  createNewBooking: clickable(testSelector('create-booking')),
});
```

There is a new one thing that comes particularly in handy here: `clickOnText` - this property allows to specify an element containing text that is later provided as an argument. In case of the calendar, we need `button` containing the number of the day in the month (which is going to be 10 for start date and 20 for end date). This might not be obvious, but if you've ever used `ember-power-calendar`, you're probably familiar with the structure of the calendar. To make sure we are dealing with the right button in the proper scope, we are passing one explicitly as a `scope` option.

The next step would be generating some `rental/show` route, where we will display the rental info and its bookings. And we are also going to provide a calendar here for creating new bookings:

```
ember g route rental/show
```

The router should include a new `show` route:

```
// book-me/app/router.js
```

```
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup')
  this.route('login');
  this.route('admin');

  this.route('rentals', function() {
    this.route('new');
  });

  this.route('rental', { path: '/rentals/:rental_id' }, function() {
    this.route('edit');
    this.route('show');
  });
});

export default Router;
```

Here's the route's body that we will start with:

```
// book-me/ap/routes/rental/show.js
import Ember from 'ember';

const {
  set,
} = Ember

export default Ember.Route.extend({
  model() {
    return this.modelFor('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },
});
```

Nothing surprising here - we are just grabbing a `rental` from, well, `rental` route and aliasing it on the controller.

Here's the template's body:

```handlebars
<!-- book-me/app/templates/rental/show.hbs -->
<h2>{{rental.name}}</h2>

<table class='table table-border'>
  <thead>
    <tr>
      <th>Begins At</th>
      <th>Finishes At</th>
      <th>Length of Stay</th>
      <th>Client Email</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    {{#each rental.bookings as |booking|}}
      <tr data-test-booking-row>
        <td>{{booking.beginsAt}}</td>
        <td>{{booking.finishesAt}}</td>
        <td>{{booking.lengthOfStay}}</td>
        <td>{{booking.clientEmail}}</td>
        <td>{{booking.price}}</td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

We are just displaying rental's name here and some properties like dates, booking's price and client's email. We are also showing the length of stay of the booking in days to make it more clear how many days are booked. This is going to be a custom computed property that will depend on `booking.beginsAt` and `booking.finishesAt` properties.

Obviously, this template won't work so far as we don't even have a `Booking` model, so let's generate one:

```
ember g model Booking
```

Let's also add `bookings` resource to `ember-cli-mirage` config:

```javascript
// book-me/mirage/config.js
export default function() {
  // current logic

  this.resource('bookings');
}
```

The next step would be setting up relationships between `Rental` and `Booking` models and defining attributes. Since we will be dealing with dates here (`beginsAt` and `finishesAt` attributes), ideally we would wrap them with `moment`. Unfortunately, there are no out-of-box attribute transforms in Ember Data that would properly handle serialization and deserialization of the datetime attributes if we care about the timezones. But the good news is that we can easily add them by installing `ember-cli-moment-transform` addon:

```
ember install ember-cli-moment-transform
```

Thanks to that addon, we can use `moment-utc` transform in `Booking` model:

```
// book-me/app/models/booking.js
import DS from 'ember-data';

const {
  Model,
  attr,
  belongsTo,
} = DS;

export default Model.extend({
  beginsAt: attr('moment-utc'),
  finishesAt: attr('moment-utc'),
  clientEmail: attr('string'),
  price: attr('number'),

  rental: belongsTo('rental'),
});
```

Let's also add `hasMany` bookings relationship to `Rental` model:

```
// book-me/app/models/rental.js
import DS from 'ember-data';

const {
  Model,
  attr,
  hasMany,
} = DS;

export default Model.extend({
  name: attr('string'),
  dailyRate: attr('number'),

  bookings: hasMany('booking'),
});
```

If we are already dealing with models, let's implement `lengthOfStay` property for bookings. The idea behind it is simple: we want to calculate some booked days based on `beginsAt` and `finishesAt` attributes. However, we need to keep in mind that those attributes are datetimes, not only dates, so we can easily end up with the case where, e.g., for one day stay we will get potential `lengthOfStay` value of 0 if the total time of the booking is less than 24h. Maybe technically it is less than one day (i.e., 24h), but in this domain, it must be counted as a one day. The simplest way to deal with it would be just subtracting dates without considering the time part. We can do that by setting time to 0 for both datetimes.

As you may have already guessed, we will start with a test:

```
// book-me/app/tests/unit/models/booking-test.js
import { moduleForModel, test } from 'ember-qunit';
import Ember from 'ember';
import moment from 'moment';

const {
  get,
} = Ember;

moduleForModel('booking', 'Unit | Model | booking', {
});

test('lengthOfStay returns number of days of stay', function(assert) {
  const model = this.subject({
    beginsAt: moment.utc('2017-10-01 14:00:00'),
    finishesAt: moment.utc('2017-10-10 10:00:00')
  });

  assert.equal(get(model, 'lengthOfStay'), 9);
});
```

This example should be enough as it already illustrates the just mentioned edge case. Here is the implementation that will make this test happy:

```
// book-me/app/models/booking.js
import DS from 'ember-data';
import Ember from 'ember';

const {
  Model,
  attr,
  belongsTo,
} = DS;

const {
  get,
  computed,
} = Ember;

export default Model.extend({
  beginsAt: attr('moment-utc'),
  finishesAt: attr('moment-utc'),
  clientEmail: attr('string'),
  price: attr('number'),

  rental: belongsTo('rental'),

  lengthOfStay: computed('beginsAt', 'finishesAt', function() {
    const beginsAt = get(this, 'beginsAt').clone();
    const finishesAt = get(this, 'finishesAt').clone();

    return finishesAt.set({ hour: 0 }).diff(beginsAt.set({ hour: 0 }), 'days');
  }),
});
```

When dealing with `moment` objects and performing any modification to them (e.g., by using `set` method), it is highly recommended to `clone` any value before - operations such as `set` are mutable, so even when dealing with `lengthOfStay` which definitely looks like something that should be read-only operation, it would still modify `beginsAt` and `finishesAt` attributes and cause unexpected side effects and a real mess in your app.

We are actually missing a link to `rental/show` route, so let's fix that and add it in `admin.hbs` template:

```
<!-- book-me/app/templates/admin.hbs -->
<h2>Admin</h2>

{{#link-to 'rentals.new' data-test-add-rental class='btn btn-primary'}}
  Add rental
{{/link-to}}

<table class='table table-border'>
  <thead>
    <tr>
      <th>Name</th>
      <th>Daily Rate</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    {{#each rentals as |rental|}}
      <tr data-test-rental-row>
```

```
        <td>{{link-to rental.name "rental.show" rental}}</td>
        <td>{{rental.dailyRate}}</td>
        <td>
          {{#link-to 'rental.edit' rental data-test-edit-rental
            class='btn btn-primary'}}
            Edit
          {{/link-to}}

          {{#link-to 'rental.show' rental data-test-show-rental
            class='btn btn-primary'}}
            Show
          {{/link-to}}

          {{delete-rental-button rental=rental}}
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Now that we get easily navigate to **rental/show** route, we can think of the subsequent step: a calendar. The cool thing is that we don't need to do much here and we can almost reuse entire example from the docs:

```
<!-- book-me/app/templates/rental/show.hbs -->
<h2>{{rental.name}}</h2>

<table class='table table-border'>
  <thead>
    <tr>
      <th>Begins At</th>
      <th>Finishes At</th>
      <th>Length of Stay</th>
      <th>Client Email</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    {{#each rental.bookings as |booking|}}
      <tr data-test-booking-row>
        <td>{{booking.beginsAt}}</td>
        <td>{{booking.finishesAt}}</td>
        <td>{{booking.lengthOfStay}}</td>
        <td>{{booking.clientEmail}}</td>
        <td>{{booking.price}}</td>
      </tr>
    {{/each}}
  </tbody>
</table>

{{#power-calendar-range
  data-test-new-booking-calendar
  selected=range
  onSelect=(action 'selectRange' value='moment') as |calendar|}}
    {{calendar.nav}}
    {{calendar.days}}
{{/power-calendar-range}}

{{outlet}}
```

We are simply displaying a very basic calendar and providing `selectRange` action to be called when the selection changes. As we are dealing here with some controller variables, we are going to handle it with controller's action, not route's action. Even though controllers should be eventually replaced by routable components, it still seems more natural to use controllers instead of routes for such cases.

Also, `outlet` part should give you a hint what is going to happen next. The idea was to display a modal when the `beginsAt` and `finishesAt` dates are selected. Instead of dealing with some conditionals to resolve if we should show the modal or not, we are going to take advantage of Ember router - a modal will simply be placed in a new route: `rental/show/createBooking`. Not only is it simpler as it makes the state management easier, but it also provides a nice way to access this modal just by visiting a page with given URL. We will use query params here and keep the selected date range in the URL. We probably won't share the URLs for creating bookings with anyone, so maybe this use case is a bit far-fetched; nevertheless, it's a good practice in Ember to use its powerful router and URLs for such state management.

Let's generate a controller now for `rental/show` route to handle `selectRange` action and general another route: `rental/show/createBooking`. Since we are going to deal with query params, we will need a controller for that route as well. Query params is a perfectly legit use case for using controllers, so we don't have many alternatives here. Let's generate those controllers and routes now:

```
ember g controller rental/show
ember g controller rental/show/createBooking
ember g route rental/show/createBooking
```

What should we put in `rental/show` controller? The first thing would be explicitly defining `range` property that we are passing as `selected` value to the calendar component, just to make it obvious what kind of data we are dealing with. The second thing would be obviously `selectRange` action. How should it work?

To answer this question, we need to think about `range` property. It is a simple

JS object with two properties: `start` and `end`, which as you may have already guessed contain selected dates. In that case, we can check if both of those dates are selected. If they are, we can parse the dates with `moment` to make sure we are dealing with UTC time. Once we parse the data and have the formatted as something that would look reasonable as query params, we can transition to `rental.show.createBooking`.

However, before we do that, we need to add one more thing. Remember the acceptance test's part where we were comparing expected attributes to be sent to `bookings` endpoint via POST request and the actual ones? The values of `startsAt` and `finishesAt` properties contained the time part. With `ember-power-calendar` we are only selecting with dates, so what can we do about time?

Just to keep things simple for the sake of example, let's assume that one day the API will cover managing rentals' `defaultArrivalHour` and `defaultDepartureHour` attributes, but for now, we will hardcode them in `Rental` model. Even though those are going to be super simple computed properties returning just some hardcoded values, it is still worth starting with a test:

```
// book-me/tests/unit/models/rental-test.js
import { moduleForModel, test } from 'ember-qunit';
import Ember from 'ember';

const {
  get,
} = Ember;

moduleForModel('rental', 'Unit | Model | rental', {
});

test('defaultArrivalHour returns 14', function(assert) {
  const model = this.subject();

  assert.equal(get(model, 'defaultArrivalHour'), 14);
});

test('defaultDepartureHour returns 10', function(assert) {
  const model = this.subject();

  assert.equal(get(model, 'defaultDepartureHour'), 10);
});
```

And here is the implementation that will make the tests happy:

```
// book-me/app/models/rental.js
import DS from 'ember-data';
import Ember from 'ember';

const {
  Model,
  attr,
  hasMany,
} = DS;

const {
  computed,
} = Ember;

export default Model.extend({
  name: attr('string'),
  dailyRate: attr('number'),

  bookings: hasMany('booking'),

  defaultArrivalHour: computed(() => {
    return 14;
  }),

  defaultDepartureHour: computed(() => {
    return 10;
  }),
});
```

In `selectRange` action we will just take those values from the rental and set them as time parts on selected dates:

```
// book-me/app/contollers/rental/show.js
import Ember from 'ember';
import moment from 'moment';

const {
  get,
  set,
} = Ember;

export default Ember.Controller.extend({
  range: {
    start: null,
    end: null,
  },

  actions: {
    selectRange(range) {
      set(this, 'range', range);

      if (range.start && range.end) {
        const rental = get(this, 'rental');
        const start = toUTCDate(range.start).set({
          hour: get(rental, 'defaultArrivalHour')
        });
        const end = toUTCDate(range.end).set({
          hour: get(rental, 'defaultDepartureHour')
        });
        this.transitionToRoute('rental.show.createBooking', rental, {
          queryParams: {
            start: start.format(),
            end: end.format(),
          }
        });
      }
    },
  },
});

function toUTCDate(date) {
  return moment.utc(date.format('YYYY-MM-DD'));
}
```

To make sure that the time part is getting set on UTC date attribute and not a local date, we are introducing `toUTCDate` helper function - managing dates in JavaScript is pretty inconvenient and even `moment` doesn't necessarily solve all possible problems with dates.

You might be wondering if we shouldn't maybe start with the unit test before writing `selectRange` action and if that's compatible with TDD flow. The answer is not that simple. To me, such things are just implementation details, and the logic is already covered by the acceptance test. For more complex scenarios I would probably write a unit test to make sure every possible edge case is handled. It might still be considered as an implementation detail, but testing different scenarios on a unit level is simply faster and easier.

We can move on now to `createBooking` route and controller. The router should be updated by running the generator, but just to double check if it's right, this is how it should be looking like now:

```
// book-me/app/router.js
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('signup')
  this.route('login');
  this.route('admin');

  this.route('rentals', function() {
    this.route('new');
  });

  this.route('rental', { path: '/rentals/:rental_id' }, function() {
    this.route('edit');
    this.route('show', function() {
      this.route('createBooking');
    });
  });
});

export default Router;
```

Since the controller will be super simple as it is going to be responsible merely for managing `start` and `end` query params, let's start with the controller:

```
// book-me/app/controllers/rental/show/create-booking.js
import Ember from 'ember';

export default Ember.Controller.extend({
  queryParams: ['start', 'end'],
  start: null,
  end: null,
});
```

Again, we are not writing any new tests here as this functionality is indirectly covered by acceptance tests and having unit tests for such code wouldn't bring many benefits either - query params don't really have much sense in isolation, so it's very similar to testing, e.g. `model` or `setupController` hooks which are just minor implementation details of something much bigger.

Let's move to the route now. This part is going to be far more complex, which means it will also be quite fun. There a couple of things this route should be responsible for. The obvious one would be setting up a new booking model. This booking should belong to a rental (which we can easily grab from the `rental` route, thanks to the awesome routing in Ember). Since both `beginsAt` and `finishesAt` dates are available in the query params, we can just parse them with `moment` and populate in `booking` model. We also need two more things: `price` and `client email`. For the email we will provide a separate input but what about `price`?

The cool thing is that we already have `lengthOfStay` computed property available in the model and the price depends exclusively on the length of stay and rental's daily rate, which can be easily taken from the model. In this case, in `model` hook we can just set up the model with populated dates, calculate the price, assign it and return the model.

There are also some other things that will be going on here. The route itself is named `createBooking` so one may expect that it will be responsible for persistence. Since the route is clearly the data owner, it will implement some action that would persist the booking.

Remember the idea how the form for persisting bookings should look like? It was supposed to be displayed inside a modal. And this is exactly what we will do in the template. Also, modals usually have some button for closing them. We are going to implement something very similar, an action like `closeModal` which will call `deleteRecord` on `booking` model (to not leave any unpersisted leftovers) and then perform transition back to `rental.show` route. The same transition makes sense for the action persisting the `booking` so we will do the same upon the successful persistence request.

The transition part is not covered by our acceptance test, so we will add it now:

```
// book-me/tests/acceptance/create-booking-test.js
/* global server */
import { test } from 'qunit';
import moduleForAcceptance from 'book-me/tests/helpers/module-for-acceptance';
import testSelector from 'ember-test-selectors';
import { authenticateSession, } from 'book-me/tests/helpers/ember-simple-auth';
import page from 'book-me/tests/pages/create-booking';
import moment from 'moment';

moduleForAcceptance('Acceptance | create booking', {
  beforeEach() {
    const user = server.create('user');
    authenticateSession(this.application, { user_id: user.id });
  },
});

test('creating a booking for rental', function(assert) {
  assert.expect(4); // one more expectation

  const dailyRate = 100;
  const rental = server.create('rental', { dailyRate });
  const clientEmail = 'client@email.com';
  const today = moment();
  const currentMonth = today.month() + 1; // month are indexed starting from 0
  const startDay = 10;
  const endDay = 20;
  const price = (endDay - startDay) * dailyRate;

  page
    .visitAdmin()
    .goToRentalPage();

  andThen(() => {
    assert.notOk(find(testSelector('booking-row')).length,
      'no bookings should be visible');
  });

  server.post('/bookings', function(schema) {
    const attributes = this.normalizedRequestAttrs();
    const expectedAttributes = {
      rentalId: rental.id,
      clientEmail,
      price,
      beginsAt: `2017-0${currentMonth}-${startDay}T14:00:00.000Z`,
      finishesAt: `2017-0${currentMonth}-${endDay}T10:00:00.000Z`,
    };

    assert.deepEqual(attributes, expectedAttributes,
      "attributes don't match the expected ones");

    return schema.rentals.create(attributes);
  });

  page
    .selectStartDate(startDay)
    .selectEndDate(endDay)
    .fillInClientEmail(clientEmail)
    .createNewBooking();

  andThen(() => {
    assert.ok(find(testSelector('booking-row')).length,
      'bookings should be visible');
    assert.equal(currentURL(), `/rentals/${rental.id}/show`,
      'should transition back to rental/show route'); // new scenario
  });
});
```

Ok, let's implement the route itself now, write a proper template and… make the acceptance test pass!

```
// book-me/app/routes/rental/show/create-booking.js
```

```
import Ember from 'ember';
import moment from 'moment';

const {
  get,
  set,
} = Ember;

export default Ember.Route.extend({
  model(params) {
    const rental = this.modelFor('rental');
    const dailyRate = get(rental, 'dailyRate');
    const beginsAt = moment.utc(params.start);
    const finishesAt = moment.utc(params.end);
    const booking = this.store.createRecord('booking', {
      rental,
      beginsAt,
      finishesAt,
    });

    const price = get(booking, 'lengthOfStay') * dailyRate;
    set(booking, 'price',  price);

    return booking;
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'booking', model);
  },

  actions: {
    createBooking(booking) {
      booking.save().then(this._transitionToRentalRoute.bind(this));
    },
  },

  _transitionToRentalRoute() {
    const rental = this.modelFor('rental');

    this.transitionTo('rental.show', rental);
  },
});
```

```
<!-- book-me/app/templates/rental/show/create-booking.hbs -->
```

```
{{#modal-dialog targetAttachment='center' translucentOverlay=true}}
  <h3>Create booking</h3>

  <form {{action (route-action 'createBooking' booking) on='submit'}}>
    <div class="form-group">
      <label for="booking-client-email">Client Email</label>
      {{input
        data-test-booking-client-email
        id="client-email"
        value=(mut booking.clientEmail)
        class="form-control"
      }}
    </div>

    <button data-test-create-booking type="submit"
      class="btn btn-primary">Create booking</button>
  </form>
{{/modal-dialog}}
```

By saving this template, we made our final test acceptance test pass :).

The template is quite basic without much new stuff except `modal-dialog` component which provides the actual modal. We want modal to be positioned in the center and to have a translucent overlay, which we can easily configure. As you can see, model's API is simple yet powerful. That is everything we need to handle this modal.

In route we've taken advantage of a pretty cool pattern - instead of providing an anonymous function for handling the success part of persisting a booking like this:

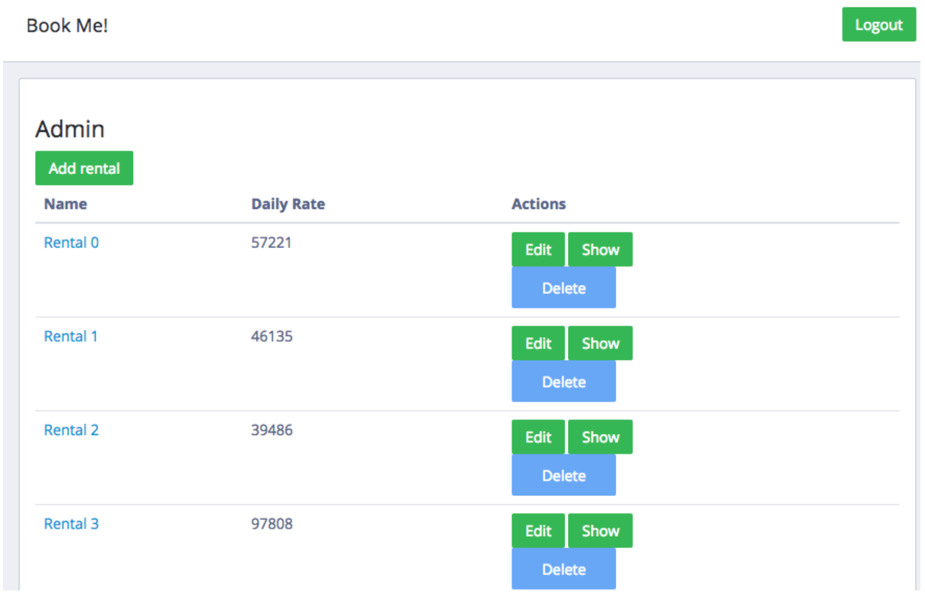
```
booking.save().then(() => {
  const rental = this.modelFor('rental');

  this.transitionTo('rental.show', rental);
});
```

we passed `_transitionToRentalRoute` function, which arguably looks more readable. We also need to keep in mind that we are taking advantage of `this` in that function which we expect to be a `route` object. In such case, we need to `bind` the proper context, that's why we are doing it as `this._transitionToRentalRoute.bind(this)` to make it work.

Here are the screenshots to illustrate what we've just achieve:

Again, there are some issues and the UI is not perfect, but this is not something that we want to devote much time in this book.

Nevertheless, we are not finished yet; there are some still few interesting things we need to add:


Book Me!

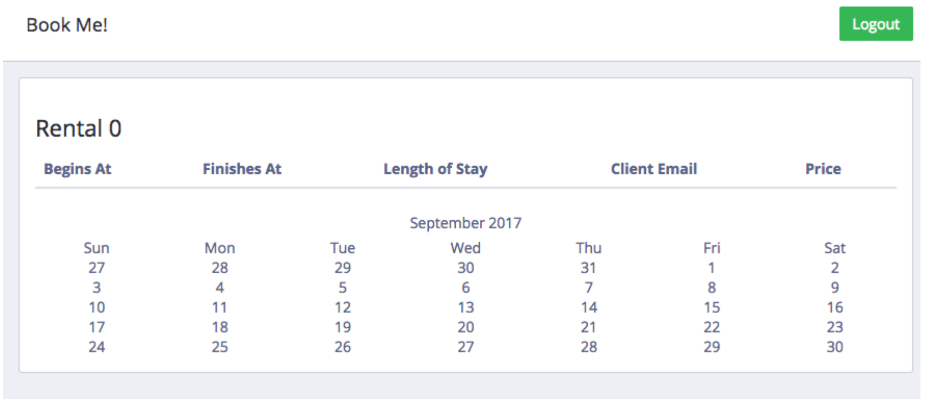
Logout

Admin

Add rental

Name	Daily Rate	Actions
Rental 0	57221	Edit Show Delete
Rental 1	46135	Edit Show Delete
Rental 2	39486	Edit Show Delete
Rental 3	97808	Edit Show Delete

Figure 10: Admin


Book Me!

Logout

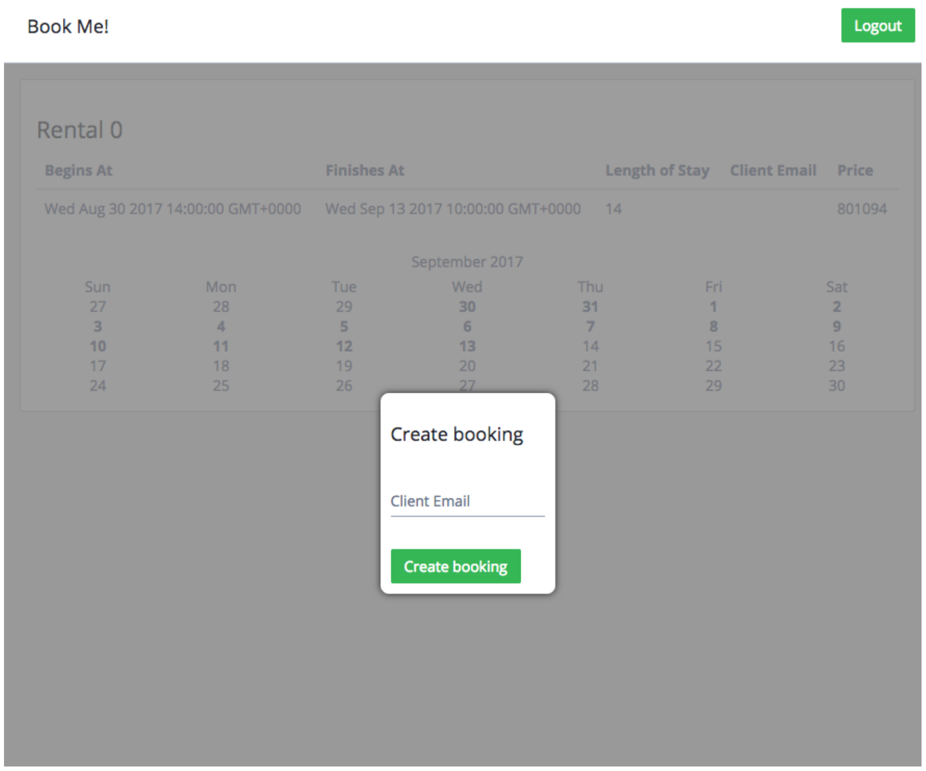
Rental 0

Begins At	Finishes At	Length of Stay	Client Email	Price

September 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Figure 11: Rental show with calendar



Rental 0

Begins At	Finishes At	Length of Stay	Client Email	Price
Wed Aug 30 2017 14:00:00 GMT+0000	Wed Sep 13 2017 10:00:00 GMT+0000	14		801094

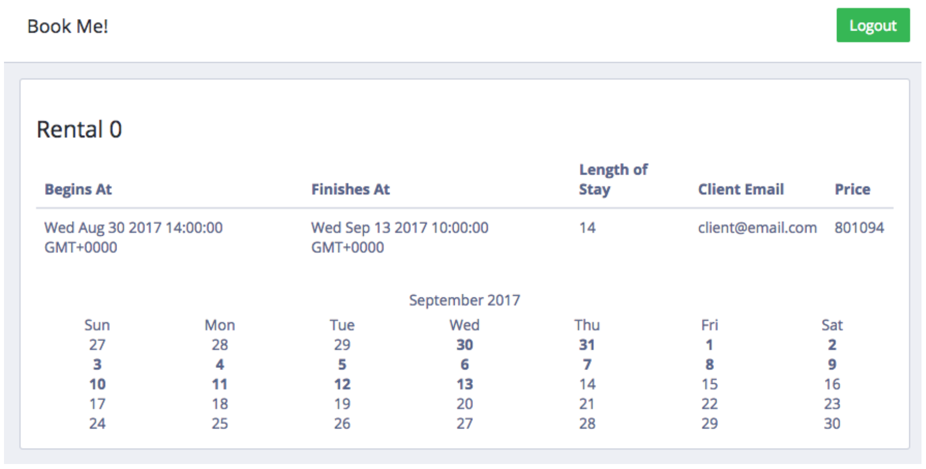
September 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	**30**	**31**	**1**	**2**
3	**4**	**5**	**6**	**7**	**8**	**9**
10	**11**	**12**	**13**	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Create booking

Client Email

Figure 12: Form in modal



Rental 0

Begins At	Finishes At	Length of Stay	Client Email	Price
Wed Aug 30 2017 14:00:00 GMT+0000	Wed Sep 13 2017 10:00:00 GMT+0000	14	client@email.com	801094

September 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	**30**	**31**	**1**	**2**
3	**4**	**5**	**6**	**7**	**8**	**9**
10	**11**	**12**	**13**	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Figure 13: Persisted booking

1. Closing the modal and transitioning to `rental/show` route.
2. Deselecting range selection in calendar upon the successful persistence of a booking. The same behavior would make sense as well in point 1.
3. Validations. The obvious ones would be validating the numericality and presence of `price` and presence and format of `clientEmail`. However, there is one not that obvious - validation of the dates and making sure that there is no overlapping of any dates.

For 3rd point, just like for previous use cases, we will generate a separate component to encapsulate the logic, so let's do it now:

```
ember g component create-booking-form
```

Here's the code for the component and its template to preserve the current behavior:

```js
// book-me/app/components/create-booking-form.js
import Ember from 'ember';

const {
  get,
} = Ember;

export default Ember.Component.extend({
  actions: {
    createBooking() {
      const booking = get(this, 'booking');

      get(this, 'onCreateBooking')(booking);
    },
  },
});
```

```hbs
<!-- book-me/app/templates/components/create-booking-form.hbs -->
<form {{action 'createBooking' on='submit'}}>
  <div class="form-group">
    <label for="booking-client-email">Client Email</label>
    {{input
      data-test-booking-client-email
      id="client-email"
      value=(mut booking.clientEmail)
      class="form-control"
    }}
  </div>

  <button data-test-create-booking type="submit"
    class="btn btn-primary">Create booking</button>
```

```
</form>
```

And the updated version of the template of `rental/show/createBooking` route which takes advantage of `create-booking-form` component:

```
<!-- book-me/app/templates/rental/show/create-booking.hbs -->
{{#modal-dialog targetAttachment='center' translucentOverlay=true}}
  <h3>Create booking</h3>

  {{create-booking-form
    booking=booking
    onCreateBooking=(route-action 'createBooking')
  }}
{{/modal-dialog}}
```

Let's start with point no. 1: "closing" the modal which will simply invoke a transition back to `rental/show` route and deselect calendar's range.

One way to start this feature would be writing an acceptance test. The other way would be covering this functionality with unit tests. Obviously, the acceptance test would be more accurate, but it would also be slower. Closing a modal is not a business-critical feature, so it might be ok make it a bit simpler and just cover it with unit or integration tests, which maybe don't provide 100% guarantee that the entire feature works (there is always a possibility that something in some layer may go wrong), but gives enough confidence that we might assume that as long as those tests pass, the feature works just fine.

Let's start with a route's action test. We will implement an action called `closeModal`. The idea behind it is simple: we want to `deleteRecord` to not leave any leftovers, reset calendar's `range` and transition back to `rental/show` route. Here's the test:

```
// book-me/tests/unit/routes/rental/show/create-booking-test.js
import { moduleFor, test } from 'ember-qunit';

moduleFor('route:rental/show/create-booking', 'Unit | Route |
  rental/show/create booking', {
});

test("closeModal action deletes booking record, resets calendar's range and
  peforms transition to `rental/show` route", function(assert) {
  assert.expect(4);

  const route = this.subject();

  const controllerStub = Ember.Object.extend({
    resetRange() {
      assert.ok(true, 'resetRange should be called');
    },
  }).create();
  const rentalStub = Ember.Object.create();
  const bookingStub = Ember.Object.extend({
    deleteRecord() {
      assert.ok(true, 'deleteRecord should be called');
    },
  }).create();

  route.controllerFor = (name) => {
    if (name === 'rental.show') {
      return controllerStub;
    }
  };
  route.modelFor = (name) => {
    if (name === 'rental') {
      return rentalStub;
    }
  };
  route.transitionTo = (routeName, rentalArgument) => {
    assert.equal(routeName, 'rental.show',
      'should transition to rental.show route');
    assert.deepEqual(rentalArgument, rentalStub,
      'should be called with proper argument');
  };

  route.actions.closeModal.bind(route)(bookingStub);
});
```

This test is far from perfect. A lot of stubs and seems almost like coupling to some very specific implementation. But you need to keep in mind that testing is about having a right level of confidence, so some aspects might depend on preference. If such unit testing is ok for you then great. If not, you still have other options, like writing acceptance test to cover this case, which is not necessarily worse.

The flow of the test is quite simple - verifying if the right methods are called with the expected arguments. We are also stubbing two methods on the route itself which might not be the best idea in most cases since stubbing object under the test is considered an anti-pattern (and for the right reason), but here it doesn't bring many issues and really helps to test this action.

Here is our implementation:

```
// book-me/app/routes/rental/show.js
import Ember from 'ember';
import moment from 'moment';

const {
  get,
  set,
} = Ember;

export default Ember.Route.extend({
  model(params) {
    const rental = this.modelFor('rental');
    const dailyRate = get(rental, 'dailyRate');
    const beginsAt = moment.utc(params.start);
    const finishesAt = moment.utc(params.end);
    const booking = this.store.createRecord('booking', {
      rental,
      beginsAt,
      finishesAt,
    });

    const price = get(booking, 'lengthOfStay') * dailyRate;
    set(booking, 'price',  price);

    return booking;
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'booking', model);
  },

  actions: {
    closeModal(booking) {
      booking.deleteRecord();

      this._transitionToRentalRoute(this);
    },

    createBooking(booking) {
      booking.save().then(this._transitionToRentalRoute.bind(this));
    },
  },

  _transitionToRentalRoute() {
    this.controllerFor('rental.show').resetRange();

    const rental = this.modelFor('rental');

    this.transitionTo('rental.show', rental);
  },
});
```

Since the same thing happens after successfully persisting a booking, let's also write a test to cover `createBooking` action:

```
// book-me/tests/unit/routes/rental/show/create-booking-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';

const {
  RSVP,
  run,
  computed,
} = Ember;

moduleFor('route:rental/show/create-booking', 'Unit | Route |
  rental/show/create booking', {
  needs: ['service:session'],
});

// previous test

test("createBooking action creates booking, resets calendar's range and
  peforms transition to `rental/show` route", function(assert) {
  assert.expect(4);

  const route = this.subject();

  const controllerStub = Ember.Object.extend({
    resetRange() {
      assert.ok(true, 'resetRange should be called');
    },
  }).create();
  const rentalStub = Ember.Object.create();
  const bookingStub = Ember.Object.extend({
    save() {
      assert.ok(true, 'save should be called');
      return RSVP.resolve();
    },
  }).create();

  route.controllerFor = (name) => {
    if (name === 'rental.show') {
      return controllerStub;
    }
  };
  route.modelFor = (name) => {
    if (name === 'rental') {
      return rentalStub;
    }
  };
  route.transitionTo = (routeName, rentalArgument) => {
    assert.equal(routeName, 'rental.show',
      'should transition to rental.show route');
    assert.deepEqual(rentalArgument, rentalStub,
      'should be called with proper argument');
  };

  run(() => {
    route.actions.createBooking.bind(route)(bookingStub);
  });
});
```

The test is quite similar, but there is one major difference - we are wrapping the method call inside `Ember.run` function, to avoid any potential surprises with promises and their async nature.

Since there are a lot of duplications in this file, you might be wondering about DRYing the tests. Personally, I'm not a big fan DRY in tests. The readability and expressiveness are essential in testing and we lose both of them when we attempt to extract some parts. I don't really do it unless the benefits are substantial. Herem it would make a small difference, so it's ok to leave those tests as they are.

There is also one more thing that should be added to both `rental/show` and `rental/show/createBooking` routes: authentication. We are going to reuse the same code as for other routes. Obviously, we will start with tests:

```
// book-me/tests/unit/routes/rental/show/create-booking-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember';

const {
  RSVP,
  run,
  computed,
} = Ember;

moduleFor('route:rental/show/create-booking', 'Unit | Route |
  rental/show/create booking', {
  needs: ['service:session'], // required for other tests
});

test("closeModal action deletes booking record, resets calendar's
  range and peforms transition to `rental/show` route", function(assert) {
  assert.expect(4);

  const route = this.subject();

  const controllerStub = Ember.Object.extend({
    resetRange() {
      assert.ok(true, 'resetRange should be called');
    },
  }).create();
  const rentalStub = Ember.Object.create();
  const bookingStub = Ember.Object.extend({
    deleteRecord() {
      assert.ok(true, 'deleteRecord should be called');
    },
  }).create();

  route.controllerFor = (name) => {
    if (name === 'rental.show') {
      return controllerStub;
    }
  };
  route.modelFor = (name) => {
    if (name === 'rental') {
      return rentalStub;
    }
  };
  route.transitionTo = (routeName, rentalArgument) => {
    assert.equal(routeName, 'rental.show',
      'should transition to rental.show route');
    assert.deepEqual(rentalArgument, rentalStub,
      'should be called with proper argument');
  };

  route.actions.closeModal.bind(route)(bookingStub);
});

test("createBooking action creates booking, resets calendar's range and
  peforms transition to `rental/show` route", function(assert) {
  assert.expect(4);

  const route = this.subject();

  const controllerStub = Ember.Object.extend({
    resetRange() {
      assert.ok(true, 'resetRange should be called');
    },
  }).create();
  const rentalStub = Ember.Object.create();
  const bookingStub = Ember.Object.extend({
    save() {
      assert.ok(true, 'save should be called');
      return RSVP.resolve();
    },
  }).create();

  route.controllerFor = (name) => {
    if (name === 'rental.show') {
      return controllerStub;
    }
  };
  route.modelFor = (name) => {
    if (name === 'rental') {
      return rentalStub;
    }
  };
  route.transitionTo = (routeName, rentalArgument) => {
    assert.equal(routeName, 'rental.show',
      'should transition to rental.show route');
    assert.deepEqual(rentalArgument, rentalStub,
      'should be called with proper argument');
  };

  run(() => {
    route.actions.createBooking.bind(route)(bookingStub);
  });
});

// new test

test('it requires authentication', function(assert) {
  assert.expect(1);

  const sessionStub = Ember.Service.extend({
    isAuthenticated: computed(() => {
      assert.ok(true, 'isAuthenticated has to be used for checking authentication');

      return true;
    }),
  });

  this.register('service:session', sessionStub);
  this.inject.service('session');

  const route = this.subject();

  route.beforeModel();
});
```

```
// book-me/tests/unit/routes/rental/show-test.js
import { moduleFor, test } from 'ember-qunit';
import Ember from 'ember'

const {
  computed,
} = Ember;

moduleFor('route:rental/show', 'Unit | Route | rental/show', {
  needs: ['service:session'],
});

// new test

test('it requires authentication', function(assert) {
  assert.expect(1);

  const sessionStub = Ember.Service.extend({
    isAuthenticated: computed(() => {
      assert.ok(true, 'isAuthenticated has to be used for checking authentication');

      return true;
    }),
  });

  this.register('service:session', sessionStub);
  this.inject.service('session');

  const route = this.subject();

  route.beforeModel();
});
```

The implementation is going to be dead-simple: just extend the route with `AuthenticatedRouteMixin` mixin:

```
// book-me/app/routes/rental/show.js
import Ember from 'ember';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

```

```
const {
  set,
} = Ember

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model() {
    return this.modelFor('rental');
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'rental', model);
  },
});
```

```
// book-me/app/routes/rental/show/create-booking.js
import Ember from 'ember';
import moment from 'moment';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  get,
  set,
} = Ember;

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model(params) {
    const rental = this.modelFor('rental');
    const dailyRate = get(rental, 'dailyRate');
    const beginsAt = moment.utc(params.start);
    const finishesAt = moment.utc(params.end);
    const booking = this.store.createRecord('booking', {
      rental,
      beginsAt,
      finishesAt,
    });

    const price = get(booking, 'lengthOfStay') * dailyRate;
    set(booking, 'price',  price);

    return booking;
  },

  setupController(controller, model) {
    this._super();

    set(controller, 'booking', model);
  },

  actions: {
    closeModal(booking) {
      booking.deleteRecord();

      this._transitionToRentalRoute(this)
    },

    createBooking(booking) {
      booking.save().then(this._transitionToRentalRoute.bind(this));
    },
  },

  _transitionToRentalRoute() {
    this.controllerFor('rental.show').resetRange();

    const rental = this.modelFor('rental');

    this.transitionTo('rental.show', rental);
  },
});
```

To finish points 1 and 2, we need to ensure those actions are handled in the component. For `createBooking` we already know that it works - the acceptance test is passing just fine. However, `closeModal` is not used yet, so we definitely need to cover it.

Let's update the template for `rental/show/createBooking`:

```
<!-- book-me/templates/rental/show/create-booking.hbs -->
{{#modal-dialog targetAttachment='center' translucentOverlay=true}}
  <h3>Create booking</h3>

  {{create-booking-form
    booking=booking
    rentalBookings=rental.bookings
    onCreateBooking=(route-action 'createBooking')
    onCancelCreation=(route-action 'closeModal')
  }}
{{/modal-dialog}}
```

As already discussed, we are not going to cover it with acceptance test, which is not perfect as a small part of this functionality won't be covered with tests at all, but it's not a business critical feature anyway, so even in worst case scenario when something breaks, it's not going to be the end of the world.

The next step will be writing component's integration tests. We will cover with tests both the action for creating booking (already implemented) and for canceling the creation (not added yet).

Again, let's start with the tests:

```
// book-me/tests/integration/components/create-booking-form-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';

const {
  set,
} = Ember;

moduleForComponent('create-booking-form', 'Integration | Component | create
  booking form', {
  integration: true
});

test('submitting form fires onCreateBooking action with booking
  as argument', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const bookingStub = Ember.Object.create();
  const onCreateBooking = (argument) => {
    assert.deepEqual(argument, bookingStub,
      'onCreateBooking should be called with booking');
  };

  set(this, 'booking', bookingStub);
  set(this, 'onCreateBooking', onCreateBooking)

  this.render(hbs`{{create-booking-form booking=booking
    onCreateBooking=onCreateBooking}}`);

  $(testSelector('create-booking')).click();
});

test('clicking cancel button fires onCancelCreation action with booking
  as argument', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const bookingStub = Ember.Object.create();
  const onCancelCreation = (argument) => {
    assert.deepEqual(argument, bookingStub,
      'onCancelCreation should be called with booking');
  };

  set(this, 'booking', bookingStub);
  set(this, 'onCancelCreation', onCancelCreation)

  this.render(hbs`{{create-booking-form booking=booking
    onCancelCreation=onCancelCreation}}`);

  $(testSelector('cancel-creation')).click();
});
```

The structure of the tests is quite simple: we are doing the necessary setup and providing stubs for bookings and actions, where we perform the assertions that the function is actually called (thanks to `assert.expect(1)` at the beginning of the tests) and that it is called with the right argument. Then, we are rendering the component and clicking the button. Here is the implementation of the component and its template to make tests happy:

```
// book-me/app/components/create-booking-form.js
import Ember from 'ember';

const {
  get,
} = Ember;

export default Ember.Component.extend({
  actions: {
    createBooking() {
      const booking = get(this, 'booking');

      get(this, 'onCreateBooking')(booking)
    },

    cancelCreation() {
      const booking = get(this, 'booking');

      get(this, 'onCancelCreation')(booking)
    },
  },
});
```

```
<!-- book-me/app/templates/components/create-booking-form.hbs -->
<form {{action 'createBooking' on='submit'}}>
  <div class="form-group">
    <label for="booking-client-email">Client Email</label>
    {{input
      data-test-booking-client-email
      id="client-email"
      value=(mut booking.clientEmail)
      class="form-control"
    }}
  </div>

  <button data-test-create-booking type="submit"
    class="btn btn-primary">Create booking</button>
  <button data-test-cancel-creation type="button" {{action 'cancelCreation'}}
    class="btn btn-danger">Close</button>
</form>
```

We are back to green!

Now, it's time for the final part of the app described in the 3rd point: validations. The most tricky one is going to be a validation that the dates do not overlap with already existing bookings, especially since format or presence validation is quite trivial to implement. That definitely sounds like `ember-changeset` validations, so let's start with generating a validator:

```
ember generate validator booking
```

To implement the proper validation for dates' overlapping, we need figure out what that even means in the first place. Let's imagine we want to create a booking with `beginsAt` time as `2017-10-01 14:00:00` and `finishesAt` as `2017-10-10 10:00:00`. We need to make sure that at no time does this dates range cover any other booking. There are four possibilities of covering some other booking:

1. Another booking has `beginsAt` before a new booking's `beginsAt` and `finishesAt` after `finishesAt` time of the new booking. The example would be a booking lasting from `2017-10-01 10:00:00` (notice the hour) to `2017-10-11 10:00:00`.
2. Other booking has `beginsAt` before a new booking's `beginsAt` and `finishesAt` between `beginsAt` / `finishesAt` time of the new booking. The example would be a booking lasting from `2017-10-01 10:00:00` (notice the hour) to `2017-10-05 10:00:00`.
3. Other booking has `beginsAt` between new booking's `beginsAt` / `finishesAt` and `finishesAt` time also between `beginsAt` / `finishesAt` time of the new booking. The example would be a booking lasting from `2017-10-02 14:00:00` to `2017-10-09 10:00:00`.
4. Other booking has `beginsAt` between new booking's `beginsAt` /

`finishesAt` and `finishesAt` after `finishesAt` time of the new booking. The example would be a booking lasting from `2017-10-05 10:00:00` to `2017-10-12 10:00:00`.

Let's assume that we are inclusive in all case on the exact times, so if one booking finishes at `2017-10-10 10:00:00`, it is also possible for another booking to start from `2017-10-11 10:00:00`.

Are you able to notice any pattern in those examples?

It looks like any potential overlapping booking would have its `beginsAt` before new booking's `finishesAt` and at the same time its `finishesAt` after new booking's `beginsAt`. So a non-overlapping booking would need to have both `beginsAt` and `finishesAt` before new booking's `beginsAt` time or both `beginsAt` and `finishesAt` after new booking's `finishesAt`.

Now that we know how the logic should be implemented, we can start thinking how it could be integrated with `ember-changeset-validator`. It sounds like we need a custom validation function. After consulting the docs, we see that it is not that hard - to make it work we need to implement a function returning another function taking `key`, `newValue`, `oldValue`, `changes` and `content` as arguments. If the result is valid, we need to return `true`. Otherwise, we return an error message. The only argument we will be interested in from those will be `content` as this is going to be a new booking with populated `beginsAt` and `finishesAt` properties.

Also, we need to somehow have access to all the bookings for given rental. Thanks to the design of validation functions, it is going to be pretty straightforward. We will just take advantage of JS closures. Since we have to implement a function returning another function, we can pass `bookings` as the argument to the outer function. This argument will be available in the inner function's scope. `ember-changeset-validators` addon expects that the factory function will take object argument. In that case, the potential implementation of such custom validator might look like this:

```
import Ember from 'ember';

const {
  get,
} = Ember;

function validateNoOverlapping({ bookings }) {
  return (_key, _value, _oldValue, _changes, content) => {
    const overlappingBookings = bookings.filter((booking) => {
      return content !== booking &&
             get(booking, 'beginsAt').isBefore(get(content, 'finishesAt')) &&
             get(booking, 'finishesAt').isAfter(get(content, 'beginsAt'))
    });

    if (overlappingBookings.length === 0) {
      return true;
    } else {
      return 'Dates must not overlap with other bookings';
    }
  };
}
```

Thanks to `moment` functions like `isBefore` or `isAfter`, implementing such logic is not that hard. We are just looking for overlapping bookings by the just mentioned criteria and if we don't find any, we consider the result valid.

It looks like we have the most challenging part covered. What else do we need to have when it comes to booking's validations?

For client's email we just need to validate the presence of the email address and the right format and, as far as price goes, it must be an integer that is greater than 0.

One optional validation would be making sure that `finishesAt` time is after `beginsAt`, but it's not possible to perform such selection with `ember-power-calendar`, so it might be enough if such validation existed exclusively server-side.

Let's write the tests then. However, the structure of the validator itself is going to be different than the ones in previous cases. For example, in case of a rental validator, we are just returning a simple object with attributes as keys and validation functions as values. Here, we need to implement a function taking `bookings` collection as an argument that returns an object with attributes as keys and validation functions as values - that's the only way to have access to `bookings`. Here are the tests that cover all the mentioned use cases:

```
// book-me/tests/unit/validators/booking-test.js
import { module, test } from 'qunit';
import validateBooking from 'book-me/validators/booking';
import Ember from 'ember';
import moment from 'moment';

module('Unit | Validator | booking');

test('it validates presence of client email', function(assert) {
  const bookings = [];
  const validator = validateBooking(bookings);

  assert.equal(validator.clientEmail[0]('clientEmail', ''),
    "Client email can't be blank");

  assert.ok(validator.clientEmail[0]('clientEmail',
    'example@mail.com'));
```

```
});

test('it validates format of client email', function(assert) {
  const bookings = [];
  const validator = validateBooking(bookings);

  assert.equal(validator.clientEmail[1]('clientEmail', 'example@'),
    'Client email must be a valid email address');

  assert.ok(validator.clientEmail[1]('clientEmail',
    'example@mail.com'));
});

test('it validates if price is an integer greater than 0', function(assert) {
  const bookings = [];
  const validator = validateBooking(bookings);

  assert.equal(validator.price('price', null), 'Price must be a number');
  assert.equal(validator.price('price', 123.12), 'Price must be an integer');

  assert.ok(validator.price('price', 100));
});

test('it validates if dates overlap with existing bookings', function(assert) {
  const bookings = [
    Ember.Object.create({
      beginsAt: moment.utc('2017-10-01 14:00:00'),
      finishesAt: moment.utc('2017-10-10 10:00:00')
    })
  ];
  const validator = validateBooking(bookings);
  const _ = null;

  let content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-01 14:00:00'),
    finishesAt: moment.utc('2017-10-10 10:00:00')
  });
  assert.equal(validator.dates(_, _, _, _, content),
    'Dates must not overlap with other bookings');

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-01 10:00:00'),
    finishesAt: moment.utc('2017-10-11 10:00:00')
  });
  assert.equal(validator.dates(_, _, _, _, content),
    'Dates must not overlap with other bookings');

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-01 10:00:00'),
    finishesAt: moment.utc('2017-10-05 10:00:00')
  });
  assert.equal(validator.dates(_, _, _, _, content),
    'Dates must not overlap with other bookings');

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-02 14:00:00'),
    finishesAt: moment.utc('2017-10-09 10:00:00')
  });
  assert.equal(validator.dates(_, _, _, _, content),
    'Dates must not overlap with other bookings');

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-05 10:00:00'),
    finishesAt: moment.utc('2017-10-12 10:00:00')
  });
  assert.equal(validator.dates(_, _, _, _, content),
    'Dates must not overlap with other bookings');

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-10 10:00:00'),
    finishesAt: moment.utc('2017-10-12 10:00:00')
  });
  assert.ok(validator.dates(_, _, _, _, content));

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-09-10 10:00:00'),
    finishesAt: moment.utc('2017-10-01 14:00:00')
  });
  assert.ok(validator.dates(_, _, _, _, content));

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-09-10 10:00:00'),
    finishesAt: moment.utc('2017-10-01 13:00:00')
  });
  assert.ok(validator.dates(_, _, _, _, content));

  content = Ember.Object.create({
    beginsAt: moment.utc('2017-10-10 11:00:00'),
    finishesAt: moment.utc('2017-10-15 10:00:00')
  });
  assert.ok(validator.dates(_, _, _, _, content));
});
```

TDDing such functionality certainly requires some knowledge of the `ember-changeset-validations` API, but after writing few validators like this, it gets much simpler.

Here's the implementation:

```
// book-me/app/validators/booking.js
import {
  validatePresence,
  validateFormat,
  validateNumber,
} from 'ember-changeset-validations/validators';

import Ember from 'ember';

const {
  get,
} = Ember;

export default function createBookingValidator(bookings) {
  return {
    clientEmail: [
      validatePresence(true),
      validateFormat({ type: 'email' })
    ],
    price: validateNumber({ integer: true, gt: 0 }),
    dates: validateNoOverlapping({ bookings }),
  }
}

function validateNoOverlapping({ bookings }) {
  return (_key, _value, _oldValue, _changes, content) => {
    const overlappingBookings = bookings.filter((booking) => {
      return content !== booking &&
             get(booking, 'beginsAt').isBefore(get(content, 'finishesAt')) &&
             get(booking, 'finishesAt').isAfter(get(content, 'beginsAt'))
    });

    if (overlappingBookings.length === 0) {
      return true;
    } else {
      return 'Dates must not overlap with other bookings';
    }
  };
}
```

Nice! All tests are passing again, so we can just take advantage of the validations in the `create-booking-form` component. Obviously, we are going to start with the integration test verifying that the error messages are properly displayed:

```
// book-me/tests/integration/components/create-booking-form-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import Ember from 'ember';
import testSelector from 'ember-test-selectors';

const {
  set,
} = Ember;

moduleForComponent('create-booking-form', 'Integration | Component |
  create booking form', {
  integration: true
});

test('it displays validation error when the data is invalid', function(assert) {
  assert.expect(2);

  const {
    $,
  } = this;
  const bookingStub = Ember.Object.create();

  set(this, 'booking', bookingStub);
  set(this, 'rentalBookings', []);
  set(this, 'onCreateBooking', () => {
    throw new Error('action should not be called');
  });

  this.render(hbs`{{create-booking-form booking=booking
    onCreateBooking=onCreateBooking rentalBookings=rentalBookings}}`);

  assert.notOk($(testSelector('booking-errors')).length,
    'errors should not initially be visible')

  $(testSelector('create-booking')).click();

  assert.ok($(testSelector('booking-errors')).length,
    'errors should be visible when submitting form with invalid data');
});
```

The idea behind the test is simple - initially, the errors should not be visible, but after attempting to create a booking, some errors should be displayed if data is invalid.

Here is the implementation that would make the new test pass:

```
// book-me/app/components/create-booking-form.js
import Ember from 'ember';
import Changeset from 'ember-changeset';
import lookupValidator from 'ember-changeset-validations';
import BookingValidators from 'book-me/validators/booking';

const {
  get,
  set,
} = Ember;

export default Ember.Component.extend({
  init() {
    this._super(...arguments);

    const rentalBookings = get(this, 'rentalBookings');
    const booking = get(this, 'booking')
    const validators = BookingValidators(rentalBookings);
    const changeset = new Changeset(booking, lookupValidator(validators), validators);

    set(this, 'changeset', changeset);
  },

  actions: {
    createBooking() {
      const changeset = get(this, 'changeset');

      changeset.validate().then(() => {
        if (get(changeset, 'isValid')) {
          get(this, 'onCreateBooking')(changeset)
        }
      });
    },

    cancelCreation() {
      const booking = get(this, 'booking');

      get(this, 'onCancelCreation')(booking);
    },
  },
});
```

The structure is almost the same as the one for persisting a rental. The only difference is setting up validators where we are passing `rentalBookings`.

Here is the template:

```
<!-- book-me/app/templates/components/create-booking-form.hbs -->
{{#if changeset.isInvalid}}
  <section data-test-booking-errors>
    {{#each changeset.errors as |error|}}
      <div class="alert alert-danger" role="alert">
        {{error.validation}}
      </div>
    {{/each}}
  </section>
{{/if}}

<form {{action 'createBooking' on='submit'}}>
  <div class="form-group">
    <label for="booking-client-email">Client Email</label>
    {{input
      data-test-booking-client-email
      id="client-email"
      value=(mut changeset.clientEmail)
      class="form-control"
    }}
  </div>

  <button data-test-create-booking type="submit"
    class="btn btn-primary">Create booking</button>
  <button data-test-cancel-creation type="button" {{action 'cancelCreation'}}
    class="btn btn-danger">Close</button>
</form>
```

Notice that we also adjusted `input` - we are no longer dealing with `booking` directly but `changeset`.

We managed to make the new test pass, but the old one, verifying that `onCreateBooking`, action gets called fails. The acceptance test for creating a new booking fails as well.

Let's start with the component's integration test. We need to adjust it by making sure the data is valid before submitting the form and that the action gets called with `changeset` argument, not a model. Also, we need to pass `rentalBookings` array

Here is the test after adjustments:

```
// book-me/tests/integration/components/create-booking-form-test.js
test('submitting form fires onCreateBooking action with booking as
  argument', function(assert) {
  assert.expect(1);

  const {
    $,
  } = this;
  const bookingStub = Ember.Object.create({
    beginsAt: moment.utc('2017-10-01 14:00:00'),
    finishesAt: moment.utc('2017-10-10 10:00:00'),
    price: 100,
  });
  const onCreateBooking = (argument) => {
    assert.deepEqual(argument._content, bookingStub,
      'onCreateBooking should be called with booking changeset');
  };

  set(this, 'booking', bookingStub);
  set(this, 'rentalBookings', []);
  set(this, 'onCreateBooking', onCreateBooking)

  this.render(hbs`{{create-booking-form booking=booking
    onCreateBooking=onCreateBooking rentalBookings=rentalBookings}}`);

  $(testSelector('booking-client-email')).val('client@example.com').change();

  $(testSelector('create-booking')).click();
});

// other tests
```

Now it's time for the final thing - making the acceptance test green again. The problem is that we are not passing `rentalBookings` to `create-booking-form`. Since this is a route that is nested inside `rental` route, we can easily get the rental via `this.modelFor('rental')`. Let's assign then a `rental` property to `controller` in `rental/show/create-booking` route:

```
// book-me/app/routes/rental/create-booking.js
import Ember from 'ember';
import moment from 'moment';
import AuthenticatedRouteMixin from 'ember-simple-auth/mixins/authenticated-route-mixin';

const {
  get,
  set,
} = Ember;

export default Ember.Route.extend(AuthenticatedRouteMixin, {
  model(params) {
    const rental = this.modelFor('rental');
    const dailyRate = get(rental, 'dailyRate');
    const beginsAt = moment.utc(params.start);
    const finishesAt = moment.utc(params.end);
    const booking = this.store.createRecord('booking', {
      rental,
      beginsAt,
      finishesAt,
    });

    const price = get(booking, 'lengthOfStay') * dailyRate;
    set(booking, 'price',  price);

    return booking;
  },

  setupController(controller, model) {
    this._super();

    const rental = this.modelFor('rental');

    set(controller, 'booking', model);
    set(controller, 'rental', rental)
  },

  actions: {
    closeModal(booking) {
      booking.deleteRecord();

      this._transitionToRentalRoute(this)
    },

    createBooking(booking) {
      booking.save().then(this._transitionToRentalRoute.bind(this));
    },
  },

  _transitionToRentalRoute() {
    this.controllerFor('rental.show').resetRange();

    const rental = this.modelFor('rental');

    this.transitionTo('rental.show', rental);
  },
});
```

And now we can just pass the bookings of this rental to `create-booking-form` component. Since we are relying on the values of the bookings' attributes for validations, it would make sense to not render the component until the bookings are fetched and populated. Just passing `rental.bookings` won't be enough as it merely returns a promise that is not initially resolved. To have all the properties of the bookings available in the data fetched from the server, we need to make sure that the promise is settled. Fortunately, this is pretty simple. We just need to check the state of the promise. In this case, we will render the component only if `rental.bookings.isSettled` is true:

```
<!-- book-me/app/templates/components/create-booking-form.hbs -->
{{#modal-dialog targetAttachment='center' translucentOverlay=true}}
  <h3>Create booking</h3>

  {{#if rental.bookings.isSettled}}
    {{create-booking-form
      booking=booking
      rentalBookings=rental.bookings
      onCreateBooking=(route-action 'createBooking')
      onCancelCreation=(route-action 'closeModal')
    }}
  {{/if}}
{{/modal-dialog}}
```

And this all! We've successfully implemented all the features. Of course, there are a lot of small things that could be improved (e.g., handling server-side validation errors for new bookings), but after reading this booking adding, such simple features by practicing TDD should be your second nature :).

Closing Thoughts

And that would be it! I hope you enjoyed reading the book and learned some new concepts along the way. The example app was not that complex, yet it should be good enough to illustrate how to apply **TDD** rules when adding new features in any of the Ember apps and how to handle some use cases that are unique for **Ember** or **JavaScript apps** in general. Obviously there are more things that could be mentioned; however, my intention wasn't to cover every possible testing scenario, but to do something much better - to explain testing in such a way that even if you encounter something that looks unfamiliar, you will still be able to isolate the problem and deal with it without much hassle.

If there is still something that is not entirely clear or you disagree with some parts, or maybe you just want to share positive feedback, feel free to reach me at **karol.galanciak@gmail.com**.