Adding the default route

In this step we will add a couple of routes. The first will be the default one, the root of our application and the second one will be used to show more detailed information about the event.

First of all, delete the demo route:

```
'test': 'greet'
```

To add a default root, simply write:

```
'': 'index',
[...]
index: function() {

}
```

We want a list of events to be fetched and rendered to the screen when the root route is triggered:

```
index: function() {
  Organizer.EventsList = new Organizer.Events();
  new Organizer.EventsListView({collection: Organizer.EventsList});
  new Organizer.NewEventView();

  Organizer.EventsList.fetch();
}
```

You may notice that our router now partially serves responsibilities of a controller.

Showing a specific event

Okay, now what about showing a specific event? We will need some way to find out which event does the user want to see. Each record is assigned with a unique identifier that we may utilize specifically for this task. This is id generated

by Local Storage adapter and it is not very user friendly, but for now we'll have to deal with it. In the later step we are going to use our custom identifier.

So now we can utilize this id to find the required record. Local Storage adapter presents us with `find` method.

The last question is how to pass this id to the router. It appears this can be done easily. Each route may contain a variable part. Denote it with a colon:

```
'events/:id': 'showEvent'
```

The `:id` is the variable part here, meaning that it can take different values. Its value is then passed to the handler function as an argument:

```
showEvent: function(id) {
  console.log(id);
}
```

The next step is to navigate to this show route programmatically whenever user clicks on event's name. So I am going to format names as links and store ids in the HTML5 data attributes:

```
<script id="event-template" type="text/x-handlebars-template">
  <a href="#" data-id="{{id}}" class=”show”>{{title}}</a>
  <a href='#' class='btn btn-danger'>remove</a>
</script>
```

Now let's add an event handler. Currently I am listening to a `click` event on an anchor tag, but we have to re-write this because currently we have two separate anchor tags and each one is meant for its own purpose:

```
events: {
  'click .btn-danger': 'removeEvent',
  'click .show': 'showEvent'
},
```

Now code the event handler:

```
showEvent: function(e) {
  e.preventDefault();
  var id = $(e.currentTarget).data('id');
  Organizer.router.navigate("events/" + id, {trigger: true});
}
```

Finding the corresponding event

Next we have to find the corresponding event. This is a two-step process: firstly we need to fetch all the events and then find the required one:

```
showEvent: function(id) {
  Organizer.EventsList = new Organizer.Events();
  Organizer.EventsList.fetch();
    Organizer.EventsList.localStorage.find({id: id});
}
```

As long as we are working with local storage, the `fetch` operation completes nearly instantly, however when working with a back-end server you would rather send it only one request asking to retrieve a record with a specific id.

Lastly we want to update the view and display event's detailed information. I am going to call it `ShowEventView` and pass `event` to it as a `model`:

```
showEvent: function(id) {
  Organizer.EventsList = new Organizer.Events();
  Organizer.EventsList.fetch();
  new Organizer.ShowEventView( {model: Organizer.EventsList.localStorage.find({id: id})} );
}
```

Here is the actual view:

```
Organizer.ShowEventView = Backbone.View.extend({
  initialize: function() {
    this.render();
  },
  render: function() {
    var template = Handlebars.compile($('#show-event-template').html());
    this.$el.html( template(this.model) );
    $('#show-event').html(this.el);
    return this;
  }
});
```

This view is rendered as soon as it is initialized. We use `#show-event-template` and pass it our model as an argument. Please note that Local Storage returns the model in a JSON format, so I do not have to use `toJSON` here. Next I take the result and insert it into the show-event block.

Modify the markup:

```
<div id="app" class="container">
  <div class="page-header"><h1>Welcome to Organizer</h1></div>
  <h2>New event</h2>
```

```
  <div id="new-event"></div>
  <h2>List of events</h2>
  <div id="events-list"></div>

  <div id="show-event"></div>
</div>
```

Don't forget about the template:

```
<script id="show-event-template" type="text/x-handlebars-template">
  <div class="page-header"><h1>{{title}}</h1></div>
  <div class="well well-lg">{{description}}</div>
</script>
```

Reload the page and click the link. This is the place where things start to get really messy. The template is being rendered right after the list because I am not removing it anywhere in the code. Our view structure is hard-coded and therefore really rigid. Also we do not have any clear naming conventions and there is definitely some amount of code duplication in our views. That what is a called the **code smell**.

The theory of broken windows

There is a so-called **theory of broken windows**. Imagine there is an old building that no one actually cares about. Sooner or later some teenager passing by will draw a graffiti on its wall, because well if no one cares about it. The next day his friend will do the same, then someone will break the windows... And so on. Basically what happens is people seeing that nothing stops them from damaging the building.

The same thing may happen to your code. One day you need to add some feature to your app really fast, so you go ahead and implement it in the fastest possible way, which however introduces code duplication or uses bad naming conventions. The next day it happens again. Then your colleague is given a task to implement another feature. He opens your code and sees a big mess there. "Well, if no one cares about this code, why should I?" - he thinks. This way a small pretty app turned into a monster that you can't tame anymore. That in turn makes you spend more and more time adding new features and repairing the broken ones, because you just can't understand what is going in the code.

This is actually very bad, but good for us in this case because now you understand that the time for refactoring has come. Luckily, our app is small so refactoring it won't be that hard. However, before we proceed to the next lesson and start refactoring I would really like to get rid of that monstrous id in the URL and implement a custom one instead.