tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Ember.js is an open source JavaScript client-side framework used for developing web applications. It uses the MVC(Model-View-Controller) architecture pattern. In Ember.js, *route* is used as model, *handlebar template* as view and *controller* manipulates the data in the model.

This tutorial covers most of the topics required for a basic understanding of EmberJS and to get a feel of how it works.

Audience

This tutorial is designed for software programmers who aspire to learn the basics of EmberJS and its programming concepts in simple and easy ways. This tutorial will give you enough understanding on the components of EmberJS with suitable examples.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of HTML, CSS, JavaScript, Document Object Model (DOM) and any text editor. As we are going to develop web-based applications using EmberJS, it will be good if you have a good understanding on how the Internet and the web-based applications work.

Copyright & Disclaimer

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

1. EmberJS – Overview

What is Ember.js?

Ember.js is an open source, free JavaScript client-side framework used for developing web applications. It allows building client side JavaScript applications by providing a complete solution which contains data management and an application flow.

The original name of Ember.js was *SproutCore MVC framework.* It was developed by *Yehuda Katz* and initially released on in *December 2011.* The stable release of Ember.js is 2.10.0 and this was released on November 28, 2016.

Why Ember.js?

Consider the following points to understand the use of Ember.js:

- Ember.js is an open source JavaScript framework under MIT license.
- It provides the new binding syntax using the *HTMLBars* template engine which is a superset of the *Handerlbars* templating engine.
- It provides the *Glimmer rendering engine* to increase the rendering speed.
- It provides the *Command Line Interface* utility that integrates Ember patterns into development process and focuses easily on the developer productivity.
- It supports *data binding* to create the link between two properties and when one property changes, the other property will get upgraded with the new value.

Features of Ember.js

Following are the some of the most prominent features of Ember.js:

- Ember.js is used for creating reusable and maintainable JavaScript web applications.
- Ember.js has *HTML* and *CSS* at the core of the development model.
- It provides the instance initializers.
- The routes are core features of the Ember.js which are used for managing the URL's.
- Ember.js provides *Ember Inspector* tool for debugging Ember applications.
- Ember.js uses templates that help to automatically update the model, if the content of applications gets changed.

2. EmberJS – Installation

It is easy to configure Ember.js in your system. By using the Ember CLI (Command Line Interface) utility, you can create and manage your Ember projects. The Ember CLI deals with different kinds of application asset management such as concatenation, minification and versioning and also provide generators to produce components, routes etc.

To install Ember CLI, you need to have the following dependencies:

- **Git**: It is an open source version control system for tracking the changes made in the files. For more information, check the official website of git. Ember uses Git to manage its dependencies.
    - *Installing Git on Linux*: Install the Git on Linux by using this link – http://git-scm.com/download/linux
    - *Installing Git on Mac*: Install the Git on Mac OS by using this link – https://git-scm.com/download/mac
    - *Installing Git on Linux*: Install the Git on Windows by using this link – https://git-scm.com/download/win
- Node.js and npm: Node.js is an open source, used for developing server side and networking applications. It is written in JavaScript. NPM is a node package manager used for installing, sharing and managing the dependencies in the projects. Ember CLI uses Node.js run time and npm to get the dependencies.
- Bower: It is used for managing the components such as HTML, CSS, JavaScript, image files etc and can be installed by using the npm.
- Watchman: This optional dependency can be used to watch the files or directories and execute some actions when they change.
- PhantomJS: This optional dependency can be used for running browser based unit tests to interact with web page.

Installing Ember CLI

Ember CLI integrates Ember patterns into development process and focuses easily on the developer productivity. It is used for creating Ember apps with Ember.js and Ember data.

You can install Ember using npm as in the command given below:

```
npm install -g ember-cli
```

To install the beta version, use the following command:

```
 npm install -g ember-cli@2.10
```

To check the successful installation of Ember, use the following command:

```
ember -v
```

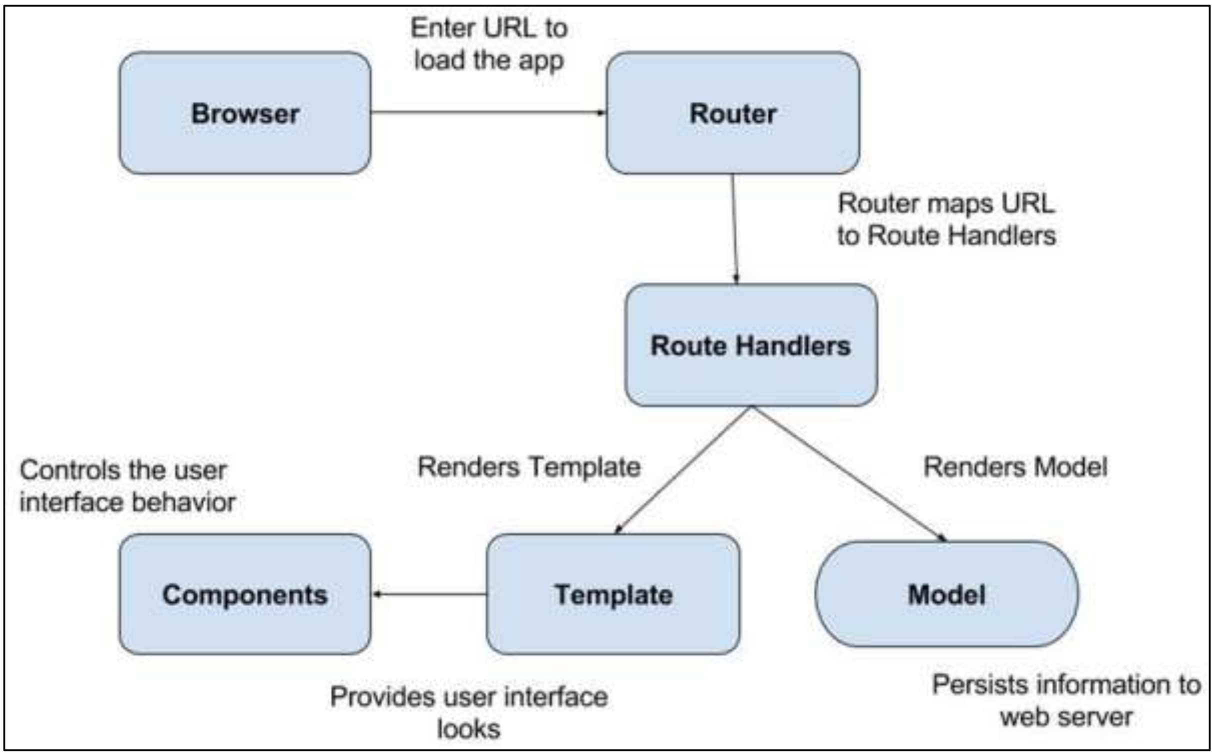
After executing the above command, it will show something like this:

```
ember-cli: 2.10.1

node: 0.12.7

os: win32 ia32
```

3. EmberJS – Core Concepts

Ember.js has the following core concepts:

- Router
- Templates
- Models
- Components


Enter URL to load the app
Browser
Router
Router maps URL to Route Handlers
Route Handlers
Controls the user interface behavior
Renders Template
Renders Model
Components
Template
Model
Provides user interface looks
Persists information to web server

Router and Route Handlers

The URL loads the app by entering the URL in the address bar and user will click a link within the app. The Ember uses the router to map the URL to a route handler. The router matches the existing URL to the route which is then used for loading data, displaying the templates and setting up an application state.

The Route handler performs the following actions:

- It provides the template.
- It defines the model that will be accessible to the template.
- If there is no permission for the user to visit a particular part of the app, then the router will redirect to a new route.

Templates

Templates are powerful UI for the end-users. Ember template provides user interface look of an application which uses the syntax of Handlebars templates. It builds the front-end application, which is like the regular HTML. It also supports the regular expression and dynamically updates the expression.

Model

The route handlers render the model that persists information to the web server. It manipulates the data stored in the database. The model is the simple class that extends the functionality of the Ember Data. Ember Data is a library that is tightly coupled with Ember.js to manipulate with the data stored in the database.

Components

The component controls the user interface behavior which includes two parts:

- a template which is written in JavaScript
- a source file which is written in JavaScript that provides behavior of the components.

4. EmberJS – Creating and Running Application

You can easily configure the Ember.js in your system. The installation of Ember.js is explained in the EmberJS Installation chapter.

Creating Application

Let us create one simple app using Ember.js. First create one folder where you create your applications. For instance, if you have created the "emberjs-app" folder, then navigate to this folder as:

```
$ cd ~/emberjs-app
```

Inside the "emberjs=app" folder, create a new project by using the *new* command:

```
$ ember new demo-app
```

When you create a project, *new* command provides the following directory structure with files and directories:

```
|-- app
|-- bower_components
|-- config
|-- dist
|-- node_modules
|-- public
|-- tests
|-- tmp
|-- vendor

bower.json
ember-cli-build.js
package.json
README.md
testem.js
```

- **app**: It specifies the folders and files of models, routes, components, templates and styles.

- **bower_components / bower.json**: It is used for managing the components such as HTML, CSS, JavaScript, image files etc and can be installed by using the npm. The *bower_components* directory contains all the Bower components and *bower.json* contains the list of dependencies which are installed by Ember, Ember CLI Shims and QUnit.

- **config**: It contains the *environment.js* directory which is used for configuring the settings of an application.

- **dist**: It includes the output files which are deployed when building the app.

- **node_modules / package.json**: NPM is a node package manager for Node.js which is used for installing, sharing and managing the dependencies in the projects. The *package.json* file includes the current npm dependencies of an application and the listed packages get installed in the **node_modules** directory.

- **public**: It includes assets like images, fonts, etc.

- **vendor**: It is a directory in which the front-end dependencies such as JavaScript, CSS are not controlled by Bower go.

- **tests / testem.js**: The automated tests are stored under the tests folder and the test runner *testem* of Ember CLI's is arranged in *testem.js*.

- **tmp**: It contains the temporary files of Ember CLI.

- **ember-cli-build.js**: It specifies how to build the app by using the Ember CLI.

Running Application

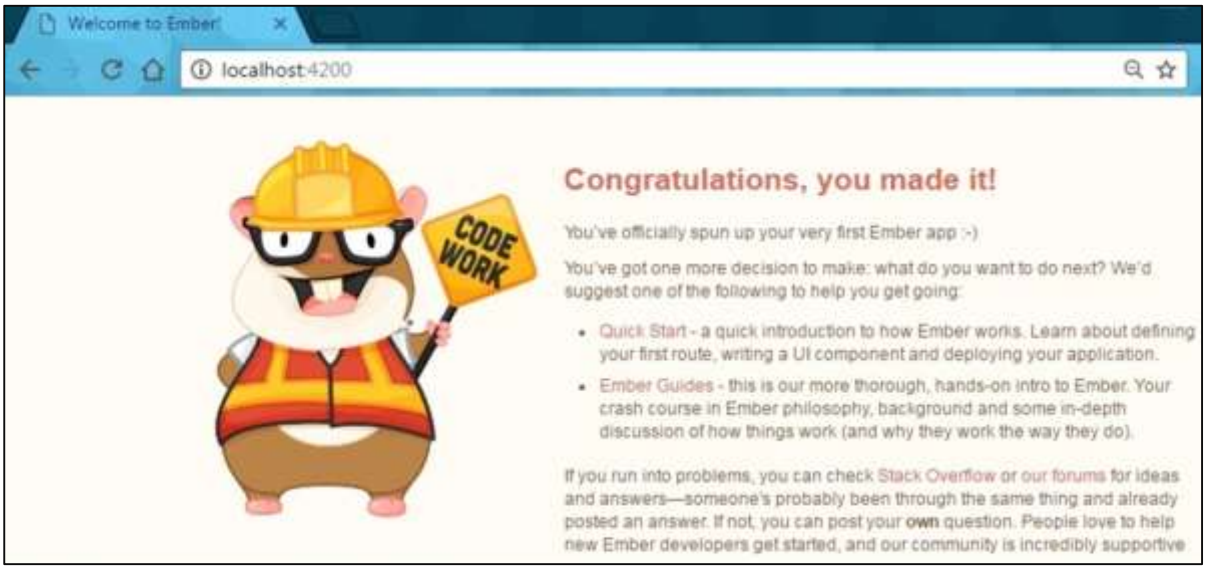
To run the application, navigate to the newly created project directory:

```
$ cd demo-app
```

We have created the new project and it is ready to run with the command given below:

```
$ ember server
```

Now open the browser and navigate to http://localhost:4200/. You will get the Ember Welcome page as shown in the image below:



Congratulations, you made it!

You've officially spun up your very first Ember app :-)

You've got one more decision to make: what do you want to do next? We'd suggest one of the following to help you get going:

- Quick Start - a quick introduction to how Ember works. Learn about defining your first route, writing a UI component and deploying your application.
- Ember Guides - this is our more thorough, hands-on intro to Ember. Your crash course in Ember philosophy, background and some in-depth discussion of how things work (and why they work the way they do).

If you run into problems, you can check Stack Overflow or our forums for ideas and answers—someone's probably been through the same thing and already posted an answer. If not, you can post your **own** question. People love to help new Ember developers get started, and our community is incredibly supportive

5. EmberJS – Object Model

In Ember.js, all objects are derived from the *Ember.Object*. Object-oriented analysis and design technique is called **object modeling**. The *Ember.Object* supports features such as mixins and constructor methods by using the class system. Ember uses the Ember.Enumerable interface to extend the JavaScript *Array* prototype to give the observation changes for arrays and also uses the formatting and localization methods to extend the *String* prototype.

The following table lists down the different types of object model in Ember.js along with their description:

S.NO.	Types & Description
1	**Classes and Instances** Class is a template or blue print, that has a collection of variables and functions, whereas instances are related to the object of that class. You can create new Ember class by using the Ember.Object's *extend()* method.
2	**Reopening Classes and Instances** This is nothing but updating the class implementation without redefining it.
3	**Computed Properties** A computed property declares functions as properties and Ember.js automatically calls the computed properties when needed and combines one or more properties in one variable.
4	**Computed Properties and Aggregate Data** The computed property accesses all items in an array to determine its value.
5	**Observers** The observer observes the property such as computed properties and updates the text of the computed property.
6	**Bindings** The binding is a powerful feature of Ember.js which helps to create a link between two properties and if one of the properties gets changed, the other one is updated automatically.

EmberJS – Classes and Instances

Class is a template or blue print, that has a collection of variables and functions, where as instances are related to the object of that class. Creating and extending the Ember class on Ember.Object is the main property of the Ember object model.

Defining Classes

You can create new Ember class by using the Ember.Object's *extend()* method:

```
const Demo = Ember.Object.extend({
   //code here
});
```

The above code creates new Ember class called "Demo" which inherits the properties from initializers, computed properties, etc. After creating the class, you need to create instance of it by using the *create()* method as shown below:

```
const state = Demo.create();
```

Using the above instance "state", access the properties by using the *set* and *get* accessor methods.

```
console.log(state.get('stateOn'));
```

You can change the "stateon" property by using the set method as shown below:

```
state.set('stateOn', true);
```

Initializing Instance

You can initialize the new instance by invoking the *init()* method. When declaring objects in the class, you need to initialize each instance with the *init()* method.

Example

The following example uses the above mentioned properties and displays an alert message when an Ember object is initialized:

```
import Ember from 'ember';   //import ember module
export default function() {
    //new ember object
    const Demo = Ember.Object.extend({
        init(){
            alert('The default property of stateOn is : ' + this.get('stateOn'));
        },
        stateOn: false
```

```
   });

   const state = Demo.create();   //new instance from object with create() method
   state.set('stateOn', true);
   console.log(state.get('stateOn'));
}
```

Now open the *app.js* file and add the following line on top of the file:

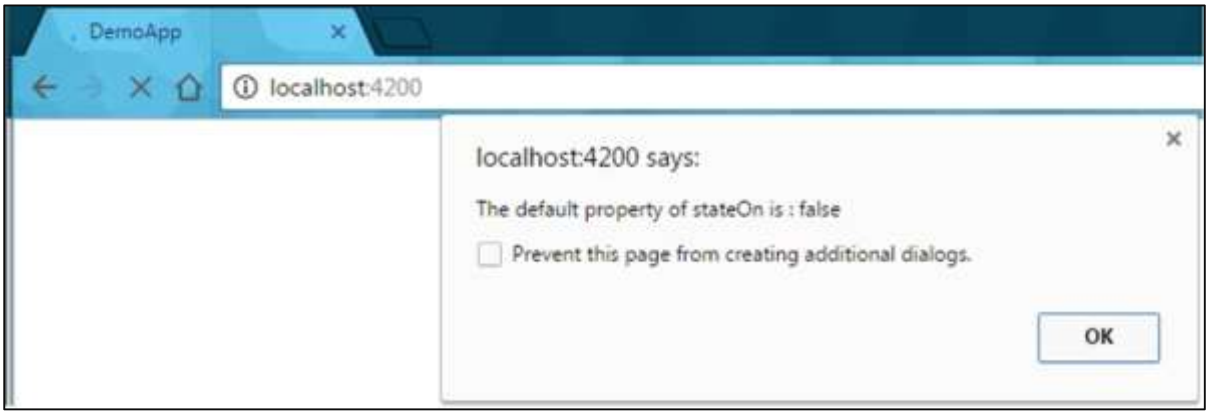
```
import classinstance from './classinstance';
```

Where, classinstance is a name of the file specified as "classinstance.js" and created under the "app" folder. Now, call the inherited "classinstance" at the bottom, before the export. This executes the classinstance function which is created in the *classinstance.js* file:

```
classinstance();
```

Output

Run the ember server and you will receive the following output:



localhost:4200 says:

The default property of stateOn is : false

☐ Prevent this page from creating additional dialogs.

EmberJS – Classes and Instances

This is nothing but updating the class implementation without redefining it and reopening the class by specifying new properties in it. This is possible by using the following methods:

- **reopen()**: It adds properties and methods to *instances*.
- **reopenClass()**: It adds properties and methods to the *classes*.

Example

The following example uses the methods mentioned above and specifies the new properties or methods in it:

```
import Ember from 'ember';
export default function() {
   // reopen() method for instances
   var Person = Ember.Object.extend({
      firstName: null,
      lastName:  null,
   });

   // adding new variable to the Person class
   Person.reopen({
      middleName: 'Smith',
   });
   document.write('Middle Name: '+Person.create().get('middleName'));
   document.write("<br>");

   // reopenClass() method for classes
   Person.reopenClass({
      //creating new function for class Person
      openClass: function() {
         return Person.create({isMan: true});
      }
   });
   document.write('isMan: '+Person.openClass().get('isMan'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import reopenclass from './reopenclass';
```

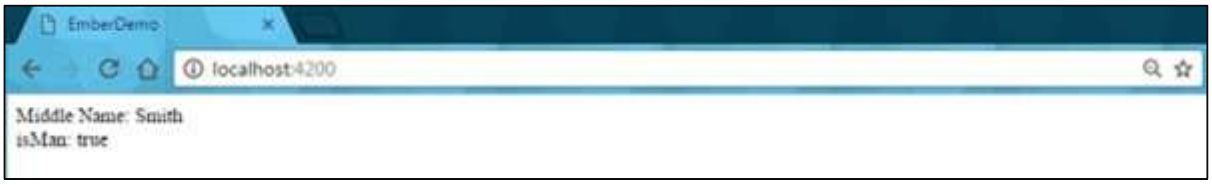
Where, reopenclass is a name of the file specified as "reopenclass.js" and created under the "app" folder.

Next call the inherited "reopenclass" at the bottom, before the export. It executes the reopenclass function which is created in the *reopenclass.js* file:

```
reopenclass();
```

Output

Run the ember server and you will receive the following output:



Middle Name: Smith
isMan: true

EmberJS – Classes and Instances

This is nothing but updating the class implementation without redefining it and reopening the class by specifying new properties in it. This is possible by using the following methods:

- **reopen()**: It adds properties and methods to *instances*.
- **reopenClass()**: It adds properties and methods to the *classes*.

Example

The example given below uses the methods mentioned above and specifies the new properties or methods in it:

```
import Ember from 'ember';
export default function() {
   // reopen() method for instances
   var Person = Ember.Object.extend({
      firstName: null,
      lastName:  null,
   });

   // adding new variable to the Person class
   Person.reopen({
      middleName: 'Smith',
   });
   document.write('Middle Name: '+Person.create().get('middleName'));
   document.write("<br>");

   // reopenClass() method for classes
   Person.reopenClass({
      //creating new function for class Person
      openClass: function() {
         return Person.create({isMan: true});
```

```
        }
    });
    document.write('isMan: '+Person.openClass().get('isMan'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

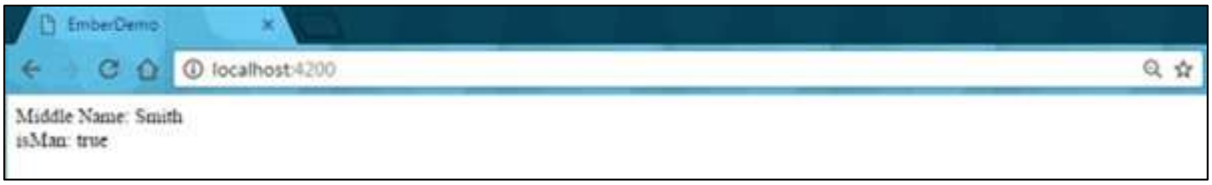
```
import reopenclass from './reopenclass';
```

Where, reopenclass is a name of the file specified as "reopenclass.js" and created under the "app" folder. Now, call the inherited "reopenclass" at the bottom, before the export. It executes the reopenclass function which is created in the *reopenclass.js* file:

```
reopenclass();
```

Output

Run the ember server and you will receive the following output:



```
Middle Name: Smith
isMan: true
```

EmberJS – Computed Properties

A computed property declares functions as properties and Ember.js automatically calls the computed properties when needed and combines one or more properties in one variable.

The following table lists down the properties of the computed property:

S.NO.	Properties & Description
1	**Chaining Computed Properties** The chaining computed propertiy is used to *aggregate* with one or more predefined computed properties.
2	**Dynamic Updating** Dynamically updates the computed property when they are called.
3	**Setting Computed Properties** Helps set up the computed properties by using the *setter and getter* methods.

Example

The following example adds the computed property to Ember.object and shows how to display the data:

```
import Ember from 'ember';
export default function () {
   var Car = Ember.Object.extend({
      //The values for below variables will be supplied by 'create' method
      CarName: null,
      CarModel: null,
      carDetails: Ember.computed('CarName', 'CarModel', function () {
         //returns values to the computed property function 'carDetails'
         return ' Car Name: ' + this.get('CarName') + '<br>' + ' Car Model: ' +
this.get('CarModel');
      })
   });

   var mycar = Car.create({
      //initializing the values of Car variables
      CarName: "Alto",
      CarModel: "800",
   });
   //Displaying the information of the car
   document.write("<h2>Details of the car: <br></h2>");
   document.write(mycar.get('carDetails'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

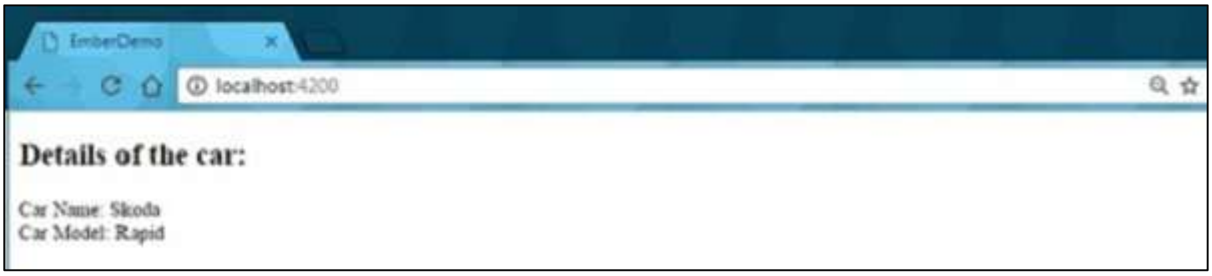
```
import computedproperties from './computedproperties';
```

Where, **computedproperties** is a name of the file specified as "computedproperties.js" and created under the "app" folder. Now, call the inherited "computedproperties" at the bottom, before the export. It executes the computedproperties function which is created in the *computedproperties.js* file:

```
computedproperties();
```

Output

Run the ember server and you will receive the following output:



Details of the car:

Car Name: Skoda
Car Model: Rapid

EmberJS – Object Model Chaining Computed Properties

The chaining computed property is used to *aggregate with one or more predefined computed properties* under a single **property**.

Syntax

```
var ClassName = Ember.Object.extend({
   NameOfComputedProperty1: Ember.computed(function() {
      return VariableName;
   }),

   NameOfComputedProperty2: Ember.computed(function() {
      return VariableName;
   });
});
```

Example

The following example shows how to use the computed properties as values to create new computed properties:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      firstName: null,
      lastName: null,
      age: null,
      mobno: null,
      // Defining the Details1 and Details2 computed property function
      Details1: Ember.computed('firstName', 'lastName', function () {
         return this.get('firstName') + ' ' + this.get('lastName');
      }),

      Details2: Ember.computed('age', 'mobno', function () {
         return 'Name: ' + this.get('Details1') + '<br>' + ' Age: ' +
this.get('age') + '<br>' + ' Mob No: ' + this.get('mobno');
      }),
   });

   var person_details = Person.create({
      //initializing the values for variables
      firstName: 'Jhon',
      lastName: 'Smith',
      age: 26,
      mobno: '1234512345'
   });
   document.write("<h2>Details of the Person: <br></h2>");
   //displaying the values by get() method
   document.write(person_details.get('Details2'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import chainingcomputedproperties from './chainingcomputedproperties';
```

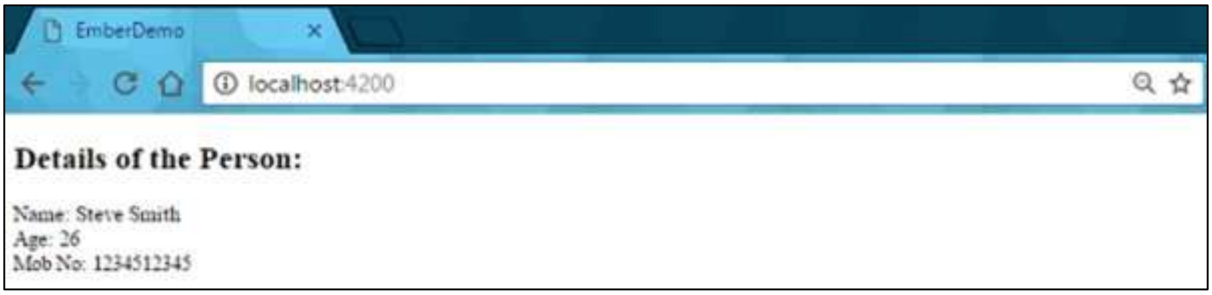
Where, **chainingcomputedproperties** is a name of the file specified as "chainingcomputedproperties.js" and created under the "app" folder.

Next call the inherited "chainingcomputedproperties" at the bottom, before the export. It executes the chainingcomputedproperties function which is created in the *chainingcomputedproperties.js* file:

```
chainingcomputedproperties();
```

Output

Run the ember server and you will receive the following output:



Details of the Person:

Name: Steve Smith
Age: 26
Mob No: 1234512345

EmberJS – Object Model Dynamic Updating

Computed properties detect the changes made on the properties and dynamically update the computed property when they are called by using the *set()* method.

Syntax

```
ClassName.set('VariableName', 'UpdatedValue');
```

Example

The following example shows dynamically updated value when changes are made to the properties:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      firstName: null,
      lastName: null,
      age: null,
      mobno: null,
      //Defining the Details1 and Details2 computed property function
      Details1: Ember.computed('firstName', 'lastName', function () {
         return this.get('firstName') + ' ' + this.get('lastName');
      }),

      Details2: Ember.computed('age', 'mobno', function () {
         return 'Name: ' + this.get('Details1') + '<br>' + ' Age: ' +
this.get('age') + '<br>' + ' Mob No: ' + this.get('mobno');
      }),
   });
```

```
    //initializing the Person details
    var person_details = Person.create({
       //Dynamically Updating the properties
       firstName: 'Jhon',
       lastName: 'Smith',
       age: 26,
       mobno: '1234512345'
    });

    // updating the value for 'firstName' using set() method
    person_details.set('firstName', 'Steve');
    document.write("<h2>Details of the Person: <br></h2>");
    document.write(person_details.get('Details2'));
}
```

Now open the *app.js* file and add below line at top of the file:

```
import dynamicupdating from './dynamicupdating';
```

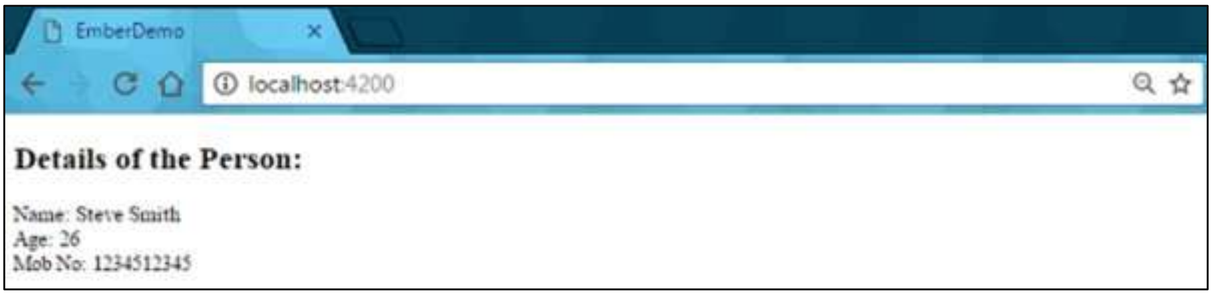
Where, dynamicupdating is a name of the file specified as "dynamicupdating.js" and created under the "app" folder.

Next call the inherited "dynamicupdating" at the bottom, before the export. It executes the dynamicupdating function which is created in the *dynamicupdating.js* file:

```
dynamicupdating();
```

Output

Run the ember server and you will receive the following output:



Details of the Person:

Name: Steve Smith
Age: 26
Mob No: 1234512345

EmberJS – Object Model Setting Computed Properties

The setting of computed properties can be done with the *Setter and Getter* methods. This manages the values of the variable declared in the computed property. The *set()* method evaluates values for a certain condition specified in the program and the *get()* method gets the values from the setter and displays the data.

Syntax

```
var ClassName = Ember.Object.extend({
   funcName: Ember.computed(function()
   {
      return VariableName;
   }
});
```

Example

The following example sets and gets the values of variable declared in the computed property and shows how to display the data:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      firstName: null,
      lastName: null,
      fullName: Ember.computed('firstName', 'lastName', function () {
         return this.get('firstName') + this.get('lastName');
      })
   });
   var nameDetails = Person.create();
   nameDetails.set('fullName', "Steve Smith");
   nameDetails.get('firstName'); // Steve
   nameDetails.get('lastName'); // Smith
   document.write("<h3>Full Name of the Person:<br><h3>" +
nameDetails.get('fullName'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

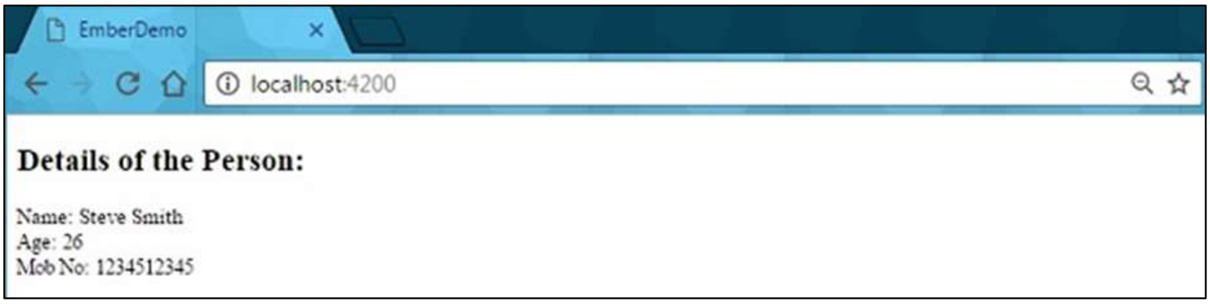
```
import settingcomputedproperties from './settingcomputedproperties';
```

Where, the settingcomputedproperties is a name of the file specified as "settingcomputedproperties.js" and created under the "app" folder. Now, call the inherited "settingcomputedproperties" at the bottom, before the export. It executes the settingcomputedproperties function which is created in the *settingcomputedproperties.js* file:

```
settingcomputedproperties();
```

Output

Run the ember server and you will receive the following output:



Details of the Person:

Name: Steve Smith
Age: 26
Mob No: 1234512345

EmberJS – Computed Properties and Aggregate Data

The computed property accesses all items in an array to determine its value. It easily adds the items and removes the items from the array. The dependent key contains a special key *@each* which updates the bindings and observer for the current computed property.

Example

The following example shows the use of computed property and the aggregate data by using Ember's *@each* key:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      //todos is an array which holds the boolean values
      todos: [
         Ember.Object.create({
            isDone: true
         }),
         Ember.Object.create({
            isDone: false
         }),
         Ember.Object.create({
            isDone: true
         })
      ],
      //dispaly the remaining values of todos
      remaining: Ember.computed('todos.@each.isDone', function () {
         var todos = this.get('todos');
         //return the todos array
         return todos.filterBy('isDone', false).get('length');
      }),
   });
   var car_obj = Person.create();
   document.write("The remaining number of cars in todo list: " +
car_obj.get('remaining'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

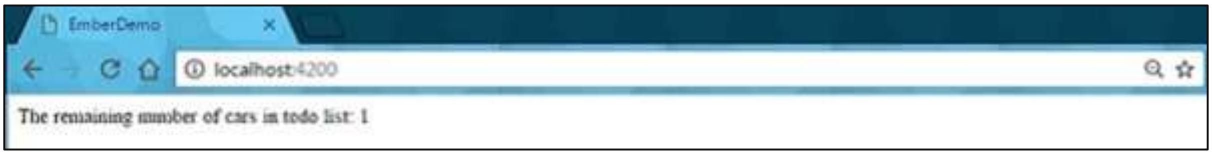
```
import computedaggregate from './computedaggregate';
```

Where, computedaggregate is a name of the file specified as "computedaggregate.js" and created under the "app" folder. Now, call the inherited "computedaggregate" at the bottom, before the export. It executes the computedaggregate function which is created in the *computedaggregate.js* file:

```
computedaggregate();
```

Output

Run the ember server and you will receive the following output:



The remaining number of cars in todo list: 1

EmberJS – Observers

The observer observes the property such as computed properties and updates the text of the computed property. It fires when the text is updated or changed.

Syntax

```
funName1: Ember.computed(function() {
   //code here
}),

funName1: Ember.observer(function() {
   //code here
});

var varname = ClassName.create({
   //code here
});
```

The following table lists down the properties of an observer:

S.NO.	Property & Description
1	**Observers and Asynchrony** Observers in an Ember.js are currently synchronous.
2	**Declaring the Observer** Declaring an obsever without the prototype extensions and outside of class definitions.

Example

The following example shows how to update the text of computed property by using observer:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      Name: null,
      // Defining the Details1 and Details2 computed property function
      Details1: Ember.computed('Name', function () {
         // get the Name value
         var Name = this.get('Name');
         // return the Name value
         return Name;
      }),

      Details2: Ember.observer('Details1', function () {
         this.set('Name','Steve Waugh');
      })
   });

   //initializing the Person details
   var person = Person.create({
     //initial value of Name varialble
     Name: 'Mark Waugh'
   });

   // updating the value for 'firstName' using set() method
   document.write('<strong>The updated name : </strong>'
+person.get('Details1'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

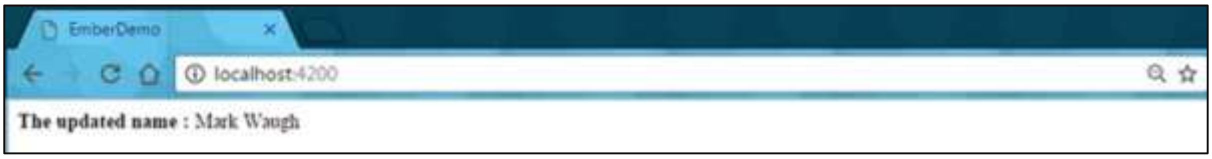
```
import observers from './observers';
```

Where, observers is a name of the file specified as "observers.js" and created under the "app" folder. Now, call the inherited "observers" at the bottom, before the export. It executes the observers function which is created in the *observers.js* file:

```
observers();
```

Output

Run the ember server and you will receive the following output:



The updated name : Mark Waugh

EmberJS – Object Model Observer and Asynchrony

Observers are synchronous in Ember.js, which fires immediately when one of the property of an observer gets updated.

Example

The following example fires as soon as one of the properties they observe changes:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      fName: null,
      lName: null,
      //Defining the Details1 and Details2 computed property function
      Details1: Ember.computed('fName', 'lName',function () {
          return this.get('fName')+' '+this.get('lName');
      }),

      Details2: Ember.observer('Details1', function () {
          this.set('fName','Will');
          this.set('lName','Smith');
      })
   });

   //initializing the Person details
   var person = Person.create({
     //initial value of fName and lName varialble
     fName: 'Mark',
     lName:'Waugh'
   });

   //updating the value for 'fName and lName' using set() method
   document.write('<strong>The updated name : </strong>'
+person.get('Details1'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import observerasynchrony from './observerasynchrony';
```

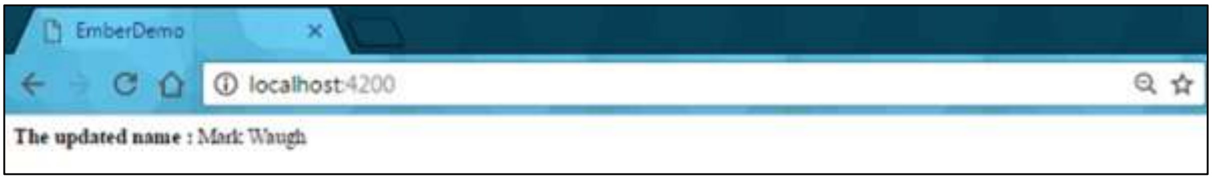
Where, observerasynchrony is a name of the file specified as "observerasynchrony.js" and created under the "app" folder.

Next, call the inherited "observerasynchrony" at the bottom, before the export. It executes the observerasynchrony function, which is created in the *observerasynchrony.js* file:

```
observerasynchrony();
```

Output

Run the ember server and you will receive the following output:



The updated name : Mark Waugh

EmberJS – Object Model Declaring the Observer

You can define the inline observers by using the *Ember.observer* method without the prototype extensions.

Following is the syntax to define the inline observers using the *Ember.observer* method.

```
App.ClassName = Ember.Object.extend({
   ComputedPropertyName: Ember.observer('ComputedPropertyNames', function() {
     //do the stuff
   })
});
```

Outside of Class Definitions

Add the observers to an object outside of a class definition by using an *addObserver()* method.

The syntax can be specified as shown below:

```
ClassName.addObserver('ComputedPropertyNames', function(){
  //do the stuff
});
```

Example

The following example specifies the inline observers by using the *Ember.observer* method:

```
import Ember from 'ember';
export default function () {
   var Person = Ember.Object.extend({
      Name: null,
      //Defining the Details1 and Details2 computed property function
      Details1: Ember.computed('Name', function () {
         //get the Name value
         var Name = this.get('Name');
         //return the Name value
         return Name;
      }),
      Details2: Ember.observer('Details1', function () {})
   });
   //initializing the Person details
   var person = Person.create({
      Name: 'Steve',
   });
   person.set('Name', 'Jhon');
   document.write('Name is Changed To: ' + person.get('Details1'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import outsideclassdefinitions from './outsideclassdefinitions';
```

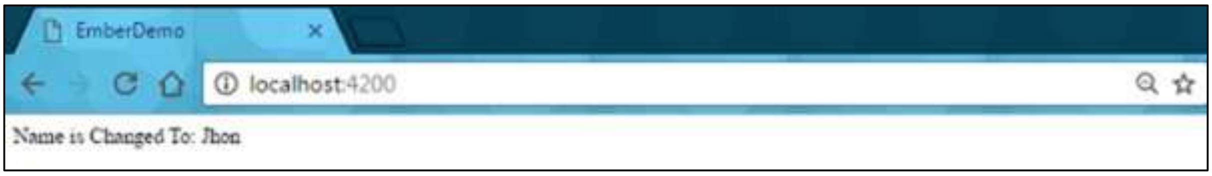
Where, **outsideclassdefinitions** is a name of the file specified as "outsideclassdefinitions.js" and created under the "app" folder.

Next call the inherited "outsideclassdefinitions" at the bottom, before the export. It executes the outsideclassdefinitions function which is created in the *outsideclassdefinitions.js* file:

```
outsideclassdefinitions();
```

Output

Run the ember server and you will receive the following output:



Name is Changed To: Jhon

EmberJS – Bindings

The binding is a powerful feature of Ember.js which helps to create a link between two properties and if one of the properties gets changed, the other one is updated automatically. You can also bind the same object or different objects.

Syntax

```
ClassName1 = Ember.Object.create({
   //code here
});

ClassName2 = Ember.Object.extend({
   //code here
});

ClassName3 = ClassName2.create({
   //code here
});
```

The syntax describes binding of two properties *ClassName1* and *ClassName2*. If *ClassName2* gets updated, it will be reflected in *ClassName1*.

Example

The following example creates link between two properties and updates one property when another property gets changed:

```
import Ember from 'ember';
export default function () {
   var CarOne = Ember.Object.create({
      //primary value
      TotalPrice: 860600
   });

   var Car = Ember.Object.extend({
      //creates property which is an alias for another property
      TotalPrice: Ember.computed.alias('CarOne.TotalPrice')
   });

   var CarTwo = Car.create({
      CarOne: CarOne
   });

   document.write('Value of car before updating: ' + CarTwo.get('TotalPrice'));
   //sets the car price
   CarTwo.set('TotalPrice', 930000);
   //above car price effects the CarOne
   document.write('<br>Value of car after updating: ' +
CarOne.get('TotalPrice'));
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import objectmodelbindings from './objectmodelbindings';
```

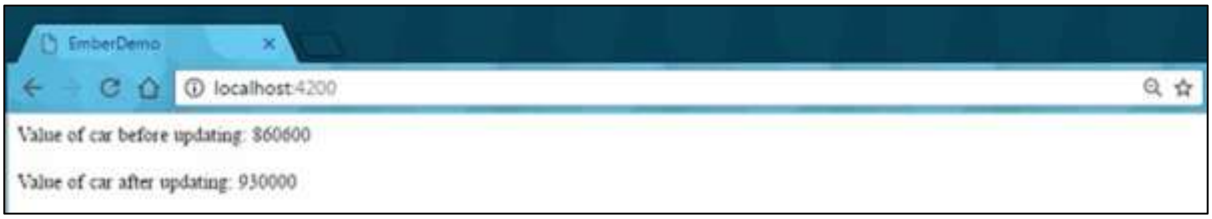
Where, objectmodelbindings is a name of the file specified as "objectmodelbindings.js" and created under the "app" folder.

Next call the inherited "objectmodelbindings" at the bottom, before the export. It executes the objectmodelbindings function which is created in the *objectmodelbindings.js* file:

```
objectmodelbindings();
```

Output

Run the ember server and you will get the following output:



```
Value of car before updating: 860600
Value of car after updating: 930000
```

The object model binding propagates changes in one direction by using the one way binding which is explained in this link –

https://www.tutorialspoint.com/emberjs/objectmodel_onewaybinding.htm.

EmberJS – One Way Binding

The object model binding specifies the changes in one direction by using the one way binding method *computed.oneWay()* and can be useful when specifying the behaviour on another property by overriding.

Example

The following example specifies the behaviour on another property by overriding:

```
import Ember from 'ember';
```

```
export default function () {
   var CarOne = Ember.Object.create({
      //primary value
      TotalPrice: 860600
   });

   var Car = Ember.Object.extend({
      TotalPrice: Ember.computed.oneWay('CarOne.TotalPrice')
   });

   var Car = Car.create({
      CarOne: CarOne
   });

   //Changing the user object name, changes the value on the view
   CarOne.set('TotalPrice', 860600);

   //Car.TotalPrice will become "860600"
   Car.set('TotalPrice', 930000); // changes to the view don't make it back to
the object.
   document.write('<h3>One Way Binding<h3>');
   document.write('Value of car : ' + CarOne.get('TotalPrice')); //display
value as 860600
}
```

Now open the *app.js* file and add the following line at the top of the file:

```
import objectmodelonewaybinding from './objectmodelonewaybinding';
```

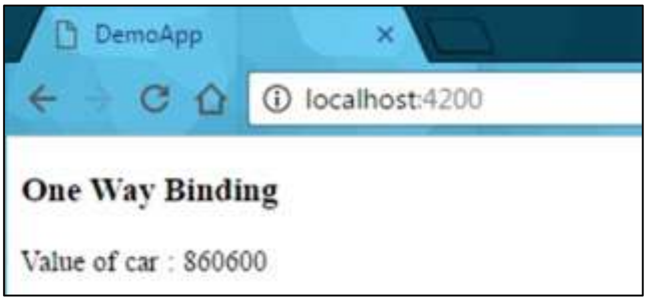
Where, **objectmodelonewaybinding** is a name of the file specified as "objectmodelonewaybinding.js" and created under the "app" folder.

Next call the inherited "objectmodelonewaybinding" at the bottom, before the export. It executes the objectmodelonewaybinding function which is created in the *objectmodelonewaybinding.js* file:

```
objectmodelonewaybinding();
```

Output

Run the ember server and you will receive the following output:



One Way Binding

Value of car : 860600

6. EmberJS – Router

Router is a core feature of EmberJs which translates an URL into a series of templates and represents the state of an application.The Ember uses the router to map the URL to a route handler. The router matches the current URL to other routes which are used for loading data, displaying the templates and to set up an application state.

Route handler performs some actions such as:

- It provides the template.
- It defines the model and it will be accessible to the template.
- If there is no permission for user to visit the particular part of an app, then router will redirect to a new route.

The following table lists down the different routers in Ember.js along with their description:

S.NO.	Types & Description
1	**Defining Routes** The router matches the current URL with routes responsible for displaying template, loading data and setting up an application state.
2	**Specifying a Route's Model** To specify a routes model, you need a template to display the data from the model.
3	**Rendering a Template** The routes are used to render the external template to the screen.
4	**Redirecting** It is a URL redirection mechanism that redirects the user to a different page when the requested URL is not found.
5	**Preventing and Retrying Transitions** The *transition.abort()* and *transition.retry()* methods can be used to abort and retry the transition respectively during a route transition.
6	**Loading/Error Substates** Ember router provides information of a route loading and errors which occur when loading a route.

7	**Query Parameters** Query parameters come into view at the right side of the *"?"* mark in a URL represented as optional key-value pairs.
8	**Asynchronous Routing** Ember.js router has the ability to handle complex async logic within an application by using asynchronous routing.

EmberJS – Defining Routes

The router matches the current URL with routes responsible for displaying template, loading data and setting up an application state. The router *map()* method is used for defining the URL mappings that pass a function which takes parameter as an object to create the routes. The *{{ link-to }}* helper navigates the router.

To define a route, use the following command in your project folder:

```
ember generate route route-name
```

It creates the route file app/routes/name_of_the_route.js, a template for the route at app/templates/name_of_the_route.hbs, and unit test file at **tests/unit/routes/route_name_of_the_test.js**.

You can define the URL mappings by using the *map()* method of router. This can be invoked with the *this* value to create an object for defining the route.

```
Router.map(function() {
  this.route('link-page', { path: '/path-to-link-page' });
  .
  .
  this.route('link-page', { path: '/path-to-link-page' });
});
```

The above code shows how to link the different pages by using the router map. It takes the *linkpage* name and *path* as an argument.

The below table shows different types of routes:

S.NO.	Routes & Description
1	**Nested Routes** It specifies the nested routes by defining a template inside another template.
2	**Dynamic Segments** It begins with a *:* in the route() method followed by an identifier.

3	**Wildcard/Globbing Routes** Wildcard routes are used for matching the multiple URL segments.

Example

The following example shows how to define a route for displaying data. Open the *.hbs* file created under *app/templates/*. Here, we have created the file as *routedemo.hbs* with the following code:

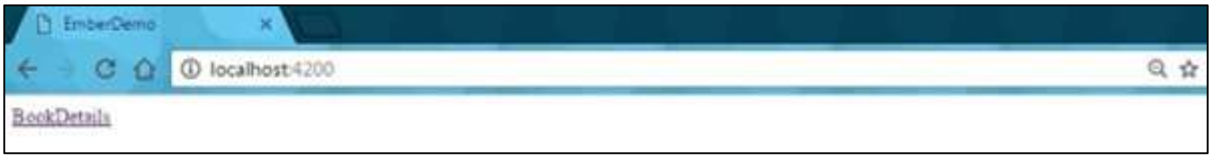
```
<h2>My Books</h2>
<ul>
    <li>Java</li>
    <li>jQuery</li>
    <li>JavaScript</li>
</ul>
```

Open the *router.js* file to define URL mappings:

```
import Ember from 'ember';                   //Access to Ember.js library as
variable Ember
import config from './config/environment';   //It provides access to app's
configuration data as variable config


//The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});


//Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('routedemo');
});


export default Router;
Create the application.hbs file and add the following code:
//link-to is a handlebar helper used for creating links
{{#link-to 'routedemo'}}BookDetails{{/link-to}}
{{outlet}} //It is a general helper, where content from other pages will appear
inside this section
```

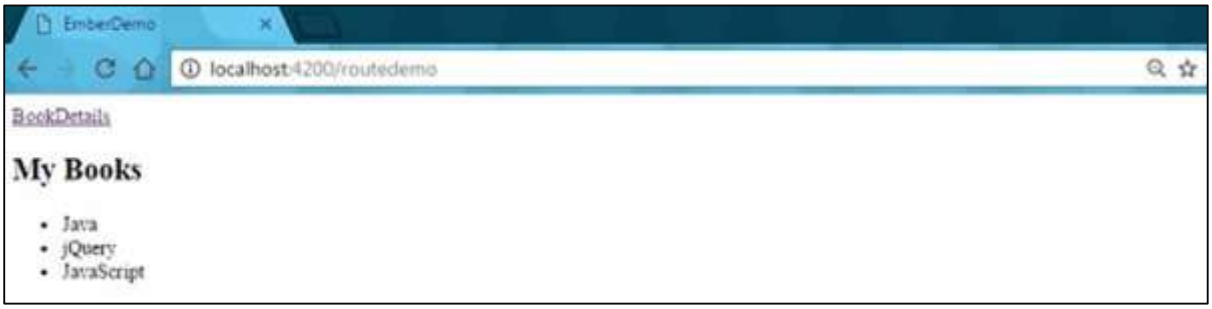
Output

Run the ember server and you will receive the following output:



BookDetails

When you click on the link of the output, a result as in the following screenshot will be generated:



BookDetails

My Books

- Java
- jQuery
- JavaScript

EmberJS – Specifying a Route's Model

You can specify a routes model, by defining the template name in the route which is the same name as the data template and implement its model hook.

```
Ember.Route.extend({
  model: function() {
    return { //value-1 },{ //value-2 },{..},{ //value-n };
  }
});
```

In the above code value-1 to value-n variables are used to store the values which are being called in the template.

The following table lists down the different types of Specifying Routes model:

S.NO.	Specifying Routes & Description
1	**Dynamic Models** It defines the routes with dynamic segment which is used by Ember to access the value from URL.
2	**Multiple Models** You can define multiple models using the *RSVP.hash* which further uses the objects to return the promises.

Example

The following example shows how to specify a route for displaying data. Create a new route as specified in the previous chapters. Now open the *router.js* file with the following code to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

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

// It specifies Router variable available to other parts of the app
export default Router;
```

Create the *application.hbs* file and add the following code:

```
// link-to is a handlebar helper used for creating links
{{#link-to 'specifyroute'}}Click here to see details{{/link-to}}
{{outlet}}
// It is a general helper, where content from other pages will appear inside
this section
```

Open the *specifyroute.hbs* file created under *app/templates/* with the following code:

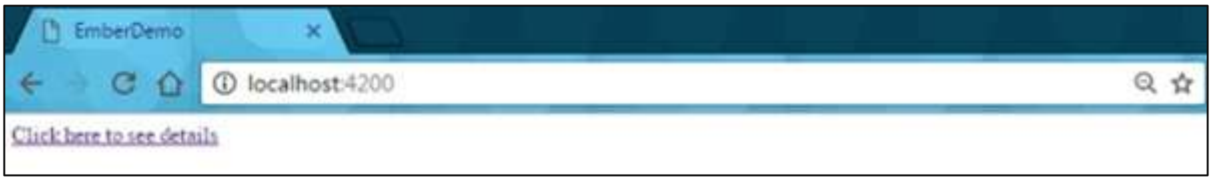
```
<h2>List of Players</h2>
<ul>
   // The <i>each</i> helper to loop over each item in the array provided from model() hook
  {{#each model as |player|}}
    <li>{{player}}</li>
  {{/each}}
</ul>
{{outlet}}
```

To construct the URL, you need to implement model to return the values:

```
import Ember from 'ember';
export default Ember.Route.extend({
  //The model() method returns the data which you want to make available to the template
  model() {
    return ['MS Dhoni', 'Steve Smith', 'Jason Roy','AB de Villiers','Kane Williamson'];
  }
});
```

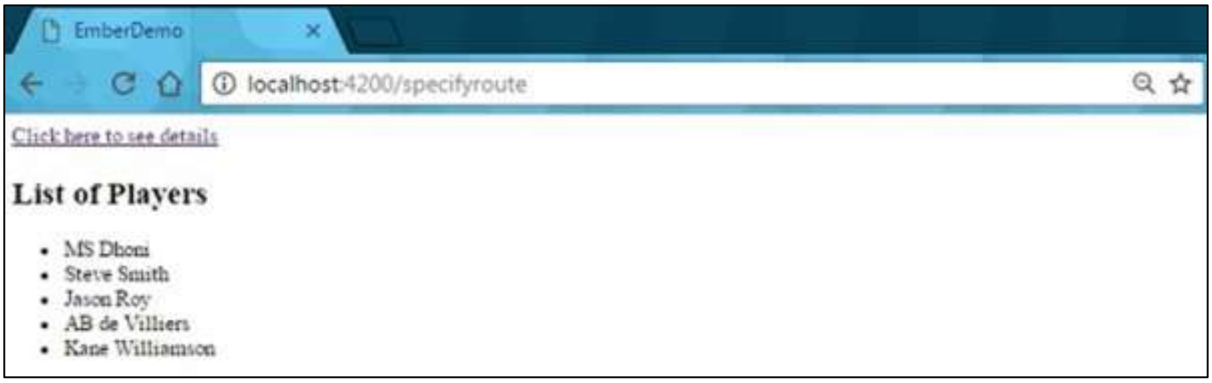
Output

Run the ember server and you will receive the following output:



Click here to see details

When you click on the link in the output, it will generate a result as in the following screenshot:



Click here to see details

List of Players

- MS Dhoni
- Steve Smith
- Jason Roy
- AB de Villiers
- Kane Williamson

EmberJS – Router Dynamic Models

Dynamic model defines the routes with *dynamic segments* which are used by Ember to access the value from URL. A dynamic segment begins with a : in the route() method and followed by an identifier. A URL must be defined with an id property in the model.

Syntax

```
Ember.Route.extend({
   model(params) {
      //code here
   }
});

Router.map(function() {
   this.route('linkpage', { path: 'identifiers' });
});
```

For example and more information on usage of dynamic model along with dynamic segment, refer the EmberJS - Router Dynamic Segment chapter.

EmberJS – Router Dynamic Models

You can define the multiple models by using *RSVP.hash*, which uses the objects to return the promises.

Syntax

```
Ember.Route.extend({
   model() {
      return Ember.RSVP.hash({
      //code here
       })
   }
});
```

Example

The example below shows how to specify multiple models for displaying data by using *RSVP.hash*. Create a new model as specified in the previous chapters. Here we have created two models as *rental* and *review* under *app/models/*.

Now open the *rental.js* file and provide its attributes:

```
import DS from 'ember-data';
export default DS.Model.extend({
  owner: DS.attr(),
  city: DS.attr()
});
```

Now open the *review.js* file and provide its attributes:

```
import DS from 'ember-data';

export default DS.Model.extend({
  author: DS.attr(),
  rating: DS.attr(),
  content: DS.attr()
});
```

You can return both the rentals and review models in one model hook and display them on index page(app/routes/index.js) by using the code given below:

```
import Ember from 'ember';


export default Ember.Route.extend({
  model() {
    //The RSVP.hash methos is built with RSVP.js library that allows to load
multiple JavaScript promises
    return Ember.RSVP.hash({
       //Find the records for the given type and returns all the records of
this type present in the store
      rentals: this.store.findAll('rental'),
      reviews: this.store.findAll('review')
    });
  },
});
```

Now you can access the model data in RSVP hash referenced in the index template, i.e., in the *app/templates/index.hbs* file:

```
<h3>Members - City </h3>
<ul>
  {{#each model.rentals as |rental|}}
    <li>{{rental.owner}} - {{rental.city}}</li>
  {{/each}}
</ul>


<h3>Member Reviews </h3>
<ul>
  {{#each model.reviews as |review|}}
<li>{{review.rating}} - {{review.content}} - by {{review.author}}</li>
  {{/each}}
</ul>
```

The code displays data from *Firebase database* which is a cloud database that stores information in JSON format. Therefore to make use of this database, create an account by using the Firebase's website.

Install the EmberFire to interface with the Firebase by Ember data.

```
ember install emberfire
```

It adds the EmberFire to *package.json* and firebase to *bower.json*.

Configuring Firebase

Login to the Firebase account and click on the *CREATE NEW PROJECT* button. Provide the same name to Firebase project which is given to Ember application.

Open the *config/environment.js* file to add the configuration info for the Ember application from the project which has been created on Firebase website.

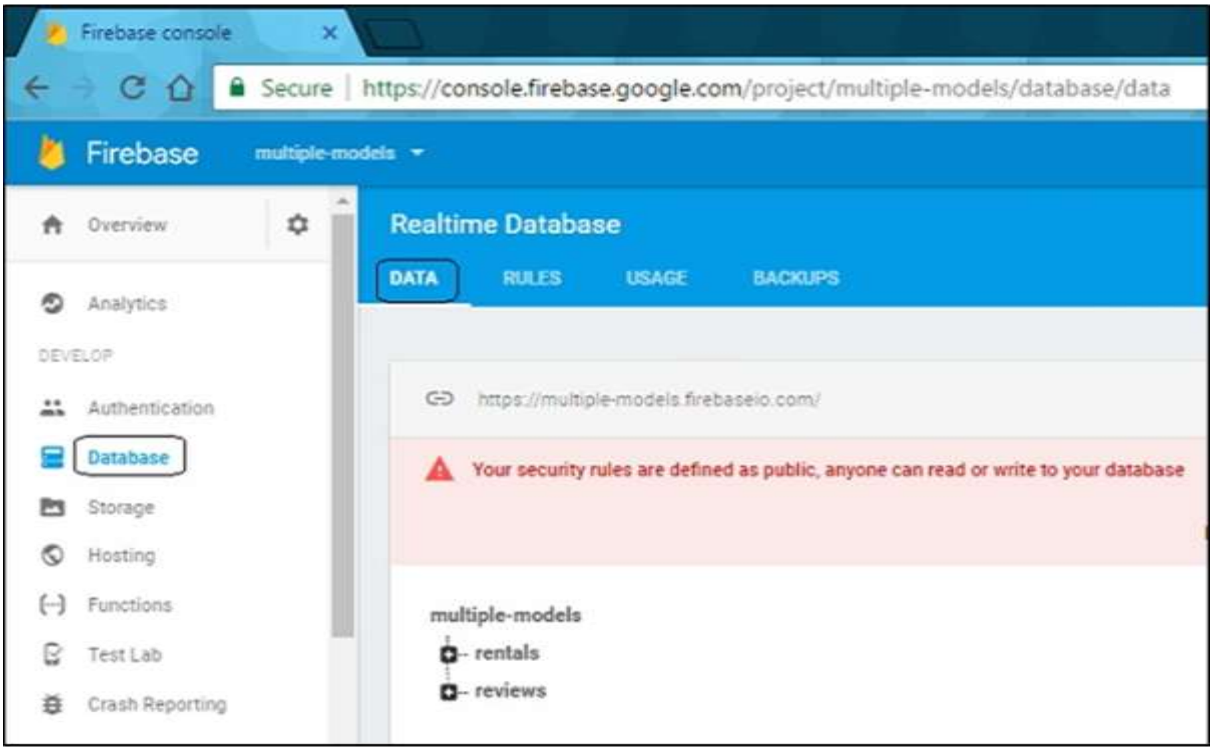
```
module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'super-rentals',
    environment: environment,
    rootURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember canary build
        // e.g. 'with-controller': true
      }
    },

    firebase: {
      apiKey: "AIzaSyAqxzlKErYeg64iN_uROKA5eN40locJSXY",
      authDomain: "multiple-models.firebaseapp.com",
      databaseURL: "https://multiple-models.firebaseio.com",
      storageBucket: "multiple-models.appspot.com"
    },

    APP: {
      // Here you can pass flags/options to your application instance
      // when it is created
    }
  };


  //other code here
```

You need to change the firebase section defined under the *ENV* section. Click on the Firebase project and click the *Add Firebase to your web app* button to include *apiKey*, *authDomain*, *databaseURL* and *storageBucket* fields from the firebase project to firebase section provided in the *environment.js* file. After configuring the EmberFire, restart the server to apply changes.

Now import the data to Firebase by using the json file. In this app, we have created a file called *rentals.json* which contains the data in JSON format.

```
{ "rentals": [{
    "owner": "Will Smith",
    "city": "San Francisco"
  }, {
    "owner": "John Davidson",
    "city": "Seattle"
  }, {
    "owner": "Shane Watson",
    "city": "Portland"
  }],
  "reviews": [{
    "author": "Will Smith",
    "rating": 4,
    "content": "Good Product"
  }, {
    "author": "John Davidson",
    "rating": 5,
    "content": "Nice Product"
  }]
}
```

Go to Firebase console, click on the *Database* section and select the *Data* tab.



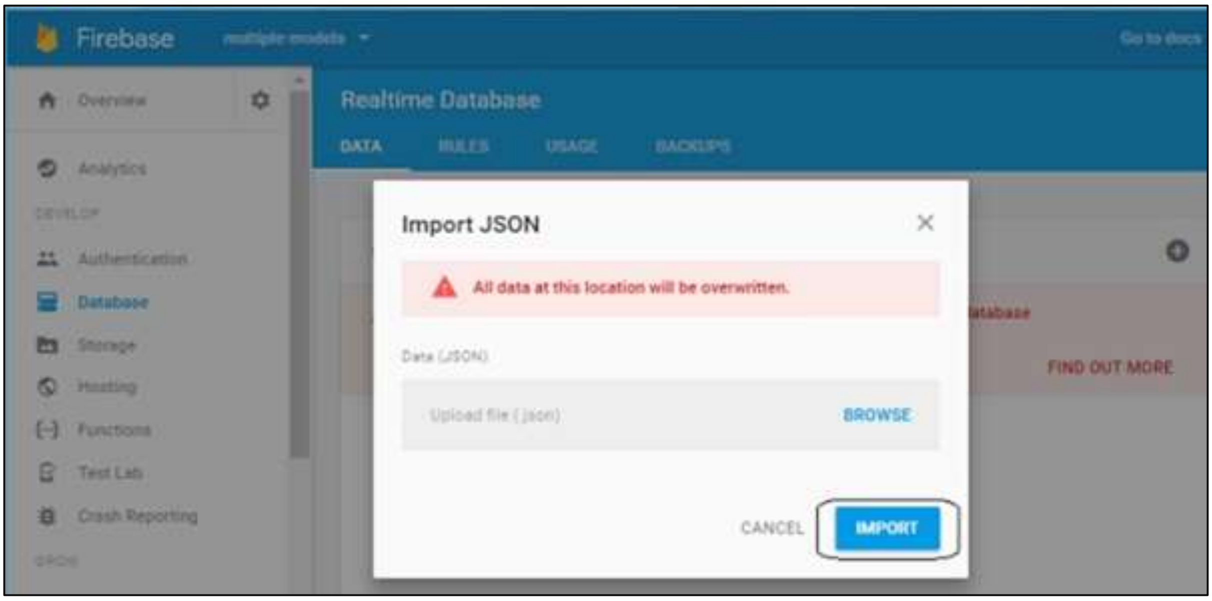
https://console.firebase.google.com/project/multiple-models/database/data

https://multiple-models.firebaseio.com/

Your security rules are defined as public, anyone can read or write to your database

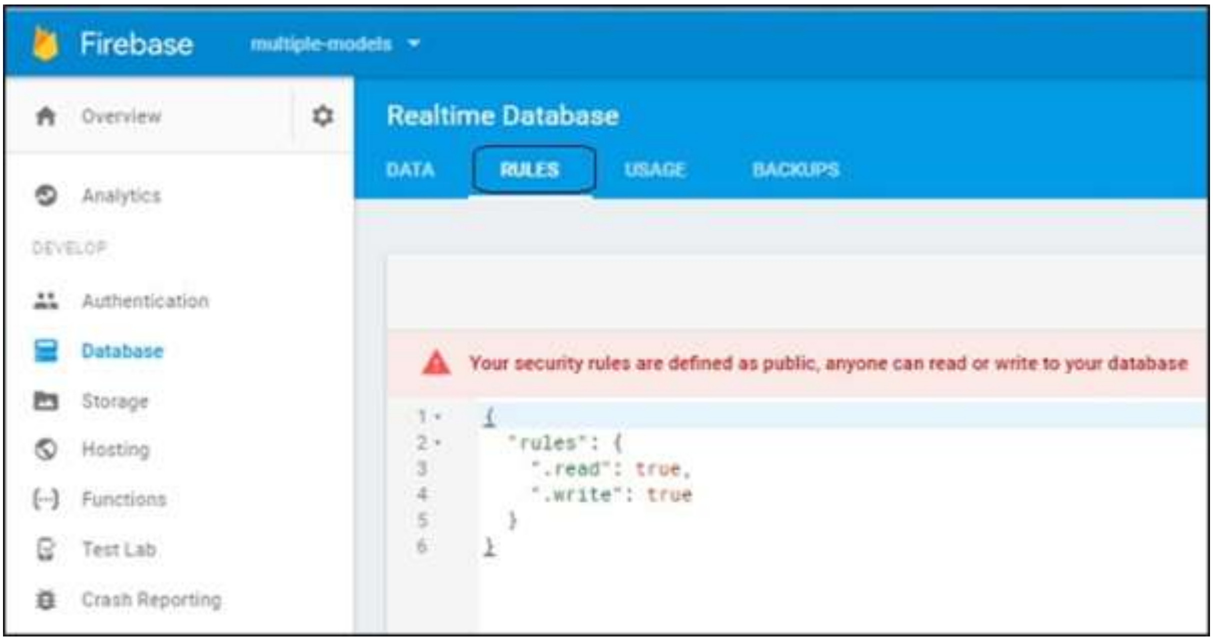
```
multiple-models
 +- rentals
 +- reviews
```

Click on the three dots on the right-hand side and select the *Import JSON* option. Next, browse the json file which you have created and click on the *IMPORT* button.



All data at this location will be overwritten.

Now set the Firebase permissions to the new database. Go to the *Rules* tab and click on *PUBLISH* to update json.



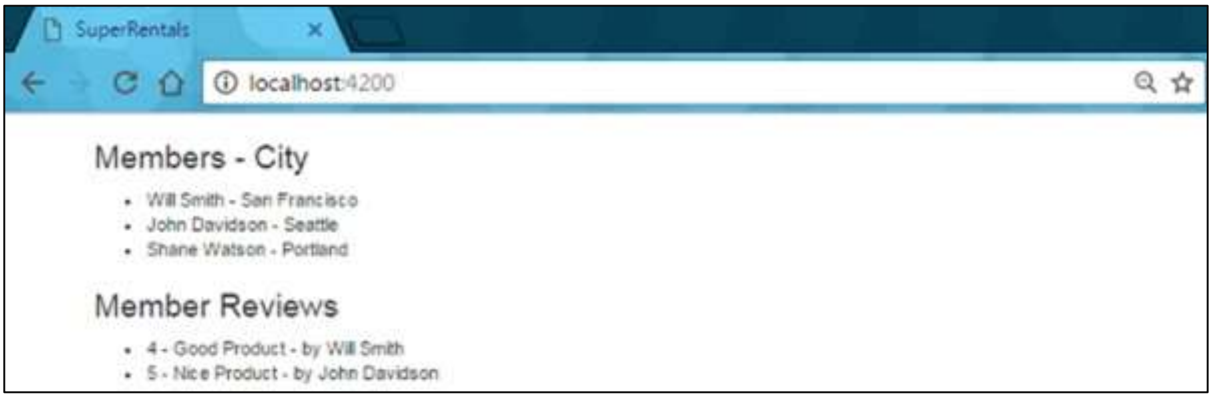
Your security rules are defined as public, anyone can read or write to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

By changing the rules, anyone can read or write to your database.

Output

Run the ember server and you will receive the following output:



Members - City

- Will Smith - San Francisco
- John Davidson - Seattle
- Shane Watson - Portland

Member Reviews

- 4 - Good Product - by Will Smith
- 5 - Nice Product - by John Davidson

EmberJS – Rendering a Template

The routes are used for rendering the external template to the screen which can be achieved by defining *templateName* in the route handler.

Syntax

```
Ember.Route.extend({
  templateName: 'path'
});
```

Example

The following example shows how to render a template for displaying data. Create a new route as specified in the previous chapters. Here we have created the route as *posts* and open the *router.js* file with the following code to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
  this.route('posts', function() {
    this.route('new');
  });
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Create the *application.hbs* file and add the following code in it:

```
// link-to is a handlebar helper used for creating links
{{#link-to 'posts'}}Click Here{{/link-to}}
{{outlet}} //It is a general helper, where content from other pages will appear inside this section
```

Open the file *posts.js* created under *app/routes/* with the following code:

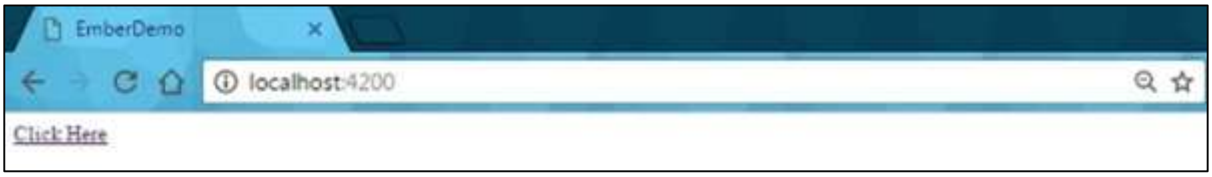
```
import Ember from 'ember';

export default Ember.Route.extend({
   templateName: 'posts/new'
});
```

Open the *posts/new.hbs* file created under *app/templates/* with the following code:

```
<h2>Posts</h2>
Page is rendered by defining templateName property.
{{outlet}}
```

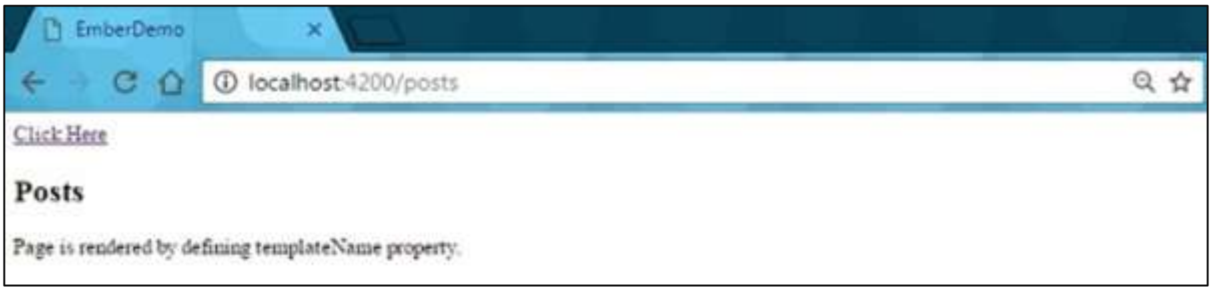
Output

Run the ember server and you will receive the following output:



Click Here

When you click the link that you receive in the output, it will generate a result as in the following screenshot:



Click Here

Posts

Page is rendered by defining templateName property.

EmberJS – Redirecting

This is a URL redirection or forwarding mechanism, that makes a web page available for more than one URL address. Ember.js defines a *transitionTo()* method moves the application into another route and it behaves like *link-to* helper.

To redirect from one route to another route, define the *beforeModel* hook into the route handler.

Syntax

```
Ember.Route.extend({
   beforeModel() {
      this.transitionTo('routeToName');
   }
});
```

Example

The example given below depicts how to redirect from one route to another. Create a new route and name it as *beforemodel* and open the *router.js* file with the following code to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('posts', function() {
    this.route('beforemodel');
  });
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *beforemodel.js* created under *app/routes/* with the following code:

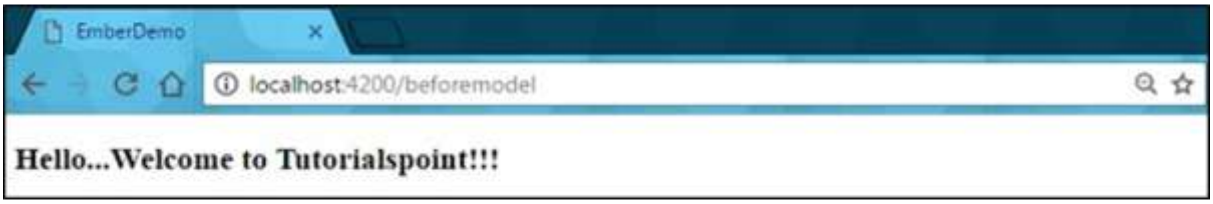
```
import Ember from 'ember';

export default Ember.Route.extend({
     beforeModel() {
     //open the beforemodel.hbs page to display the data
    this.transitionTo('beforemodel');
  }
});
```

Open the file *beforemodel.hbs* created under *app/templates/* with the following code:

```
<h2>Hello...Welcome to Tutorialspoint!!!</h2>
{{outlet}}
```

Output

Run the ember server and you will receive the following output:



Hello...Welcome to Tutorialspoint!!!

EmberJS - Preventing and Retrying Transitions

When Ember Router transfers the transition object to various hooks, then hook can abort the transition by using *transition.abort()* method and can be re-attempted, if transition object is stored by using the *transition.retry()* method.

The below table shows different ways of preventing and retrying transitions during a route transition:

S.NO.	Transition & Description
1	**Preventing Transitions Via willTransition** It fires the willTransition action on currently active routes when you re-attempt the transition by using the *{{link-to}}* helper or *transitionTo* method.
2	**Aborting Transitions** The destination routes use a transition object to abort the attempted transitions.

EmberJS – Router Preventing Transitions Via willTransition

It fires the *willTransition* action on currently active routes when you re-attempt the transition by using the *{{link-to}}* helper or the *transitionTo* method.

Syntax

```
Ember.Route.extend({
   actions: {
      willTransition(transition) {
         //handle the transition
      }
   }
});
```

Example

The example given below depicts preventing transitions via the **willTransition** action on active route. Create a route called *willtransition* and open the *router.js* file with the following code to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

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

// It specifies Router variable available to other parts of the app
export default Router;
```

Create the *application.hbs* file and add the following code:

```
// link-to is a handlebar helper used for creating links
{{link-to 'Click For Transition' 'willtransition'}}
{{outlet}}
// It is a general helper, where content from other pages will appear inside this section
```

Open the file *willtransition.js* file created under *app/routes/* with the following code:

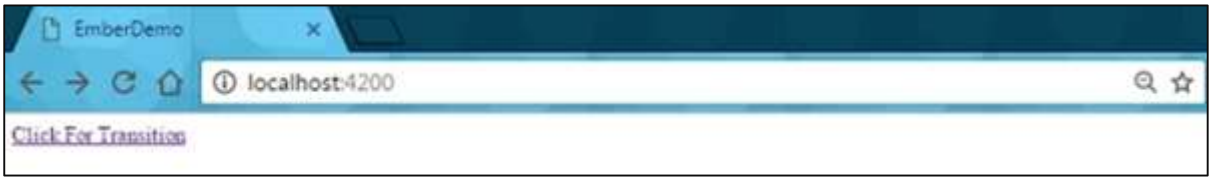
```
import Ember from 'ember';

export default Ember.Route.extend({
   actions: {
      willTransition(transition) {
         // decalring the self variable
         var self = this;
         // checking whether self variable is false or not
         if (!this.get('allowTransition')) {
            document.write('<b><font color="red">');
            // display the message
            document.write("transition abort");
            document.write('</font><br>');
            transition.abort(); //calling abort function

            Ember.run.later(function () {
               // setting the self variable to true
               self.set('allowTransition', true);
               document.write('<b><font color="blue">');
               // display the message
               document.write("transition retry");
               document.write('</font>');
               transition.retry();    // calling retry function
            }, 500);
         }
      }
   }
});
```

Open the *willtransition.hbs* file created under *app/templates/* with the following code:

```
<h2>Hello...Welcome to Tutorialspoint!!!</h2>
{{outlet}}
```

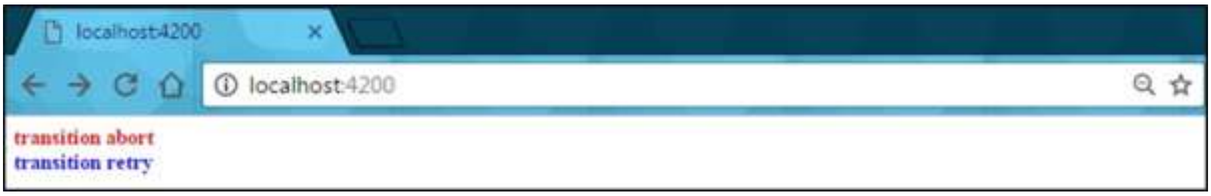
Output

Run the ember server and you will receive the following output:



Click For Transition

When you click on the link, it will display the data. But if you click on the back link, the *willTransition* action calls the *transition.abort()* and then *transition.retry()* method.



transition abort
transition retry

EmberJS – Router Aborting Transitions

The destination routes use a transition object to abort the attempted the transitions.

Syntax

```
Ember.Route.extend({
   actions: {
      willTransition(transition) {
      //do the condition for abort transiton
      transition.abort();
   }
});
```

For example, refer this link – https://www.tutorialspoint.com/routing_prvnt_trans.htm. This depicts the aborting transitions on active route by using the *willTransition* action.

EmberJS – Loading/Error Substates

The Ember.js overrides transitions for customizing asynchronization between the routes by making use of error and loading substates.

Syntax

```
Ember.Route.extend({
   model() {
      //code here
      }
});
Router.map(function() {
   this.route('path1', function() {
      this.route('path2');
   });
});
```

Example

The example given below demonstrates the use of Loading / Error Substates which occurs while loading a route. Create a new route and name it as *loaderror* and open the *router.js* file with the following code to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('loaderror', function() {
    this.route('loaderr');
  });
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *loaderror.js* file created under *app/routes/* with the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
     model() {
     return new Ember.RSVP.Promise(function (resolve, reject) {
            setTimeout(function () {
                resolve({});
            }, 1500);
        });
    }
});
```

Open the file *application.hbs* created under *app/templates/* with the following code:

```
{{outlet}}
```

Open the file *index.hbs* and add the following code:

```
{{link-to 'loaderror' 'loaderror'}}
<small>(this link displays the 'loading' route/template correctly)</small>
{{outlet}}
```

When you click on the *loaderror* link, the page should open with the loading state. Therefore, create a *loading.hbs* file to specify the loading state:

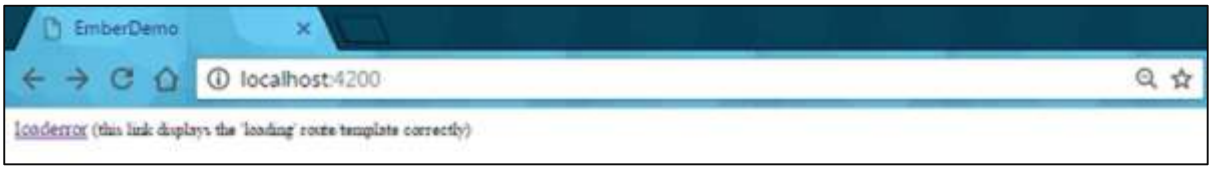
```
<h2 style="color: #f00;">template: loading</h2>
```

Now open the *loaderror.hbs* file that displays the error message:

```
<h2>--error--!</h2>
{{link-to 'loaderror.loaderr' 'loaderror.loaderr'}}
<small>(doesn't display the 'loading' route/template, because
'loaderror/loading' does not exist!!!</small>
{{outlet}}
```

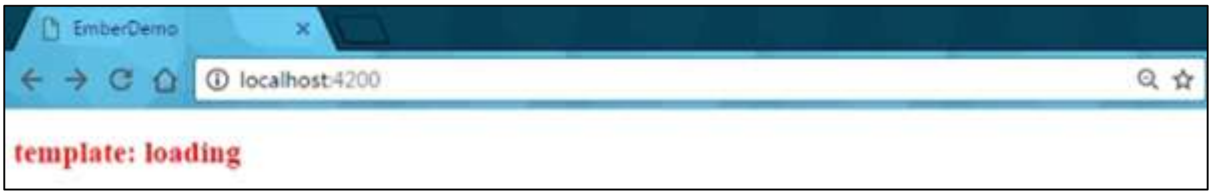
Output

Run the ember server and you will receive the following output:



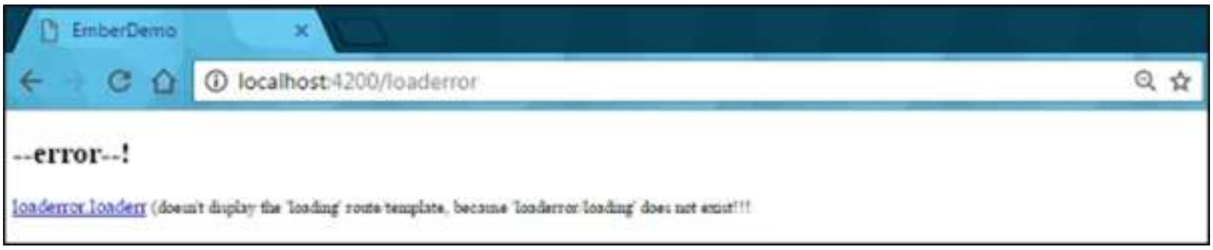
loaderror (this link displays the 'loading' route template correctly)

When you click on the link, it will show the template loading message:



template: loading

Then it displays an error substate when errors are encountered during a transition:



--error--!

loaderror.loaderr (doesn't display the 'loading' route template, because 'loaderror.loading' does not exist!!!

EmberJS – Query Parameters

Query parameters are specified on route driven controllers which appear to the right of the *?* in a URL and are represented as optional key-value pairs.

For instance:

```
http://mysite.com/articles?sort=ASC&page=2
```

The above URL has the two query parameters; one is *sort* and the other is *page* which contains values *ASC* and *2* respectively.

The following table lists down the different ways of using query parameters:

S.NO.	Query Parameters & Description
1	**Specifying Query Parameters** You can specify the query parameters on the route-driven controllers.
2	**Opting Into a Full Transition** You can use optional queryParams configuration when a controller query parameter property changes to opt into a full transition.
3	**Update URL with Replacestate Instead** It prevents from adding an item to your browser's history.
4	**Map a Controller's Property to a Different Query Param Key** Mapping a controller query parameter property to a different query parameter key.
5	**Default Values and Deserialization** Specifying the default values to the query parameter.
6	**Sticky Query Param Values** In Ember, the query parameter values are sticky by default; so that any changes made to the query parameter, the new value of query parameter will be preserved by re-entering the route.

EmberJS – Router Specifying Query Parameters

You can specify the query parameters on the route-driven controllers which can bind in the URL and configure the query parameters by declaring them on controller to make them active. You can display the template by defining a computed property of query parameter-filter of an array.

Syntax

```
Ember.Controller.extend({
   queryParams: ['queryParameter'],
   queryParameter: null
});
```

Example

The example given below shows specifying query parameters on route-driven controllers. Create a new route and name it as *specifyquery* and open the *router.js* file to define URL mappings:

```
import Ember from 'ember';                   //Access to Ember.js library as
variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('specifyquery');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* created under *app/templates/* with the following code:

```
<h2>Specifying Query Parameters</h2>
{{#link-to 'specifyquery'}}Click Here{{/link-to}}
```

When you click on the above link, the page should open with a form. Open the *specifyquery.hbs* file to send the parameters on the route-driven controllers:

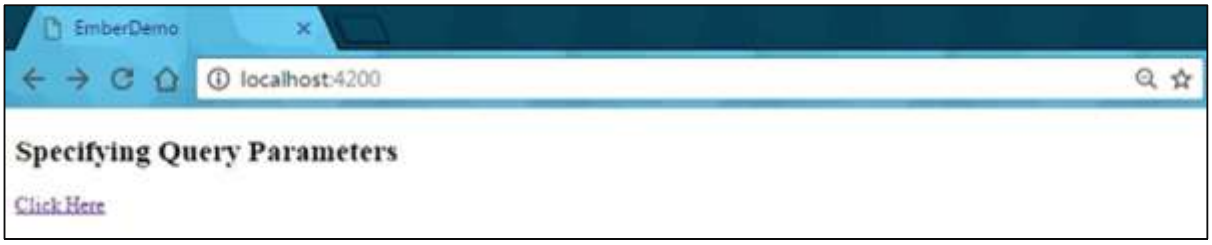
```
<form {{action "addQuery" on="submit"}}>
 {{input value=queryParam}}
 <input type="submit" value="Send Value"/>
</form>
{{outlet}}
```

Now define the computed property of *queryParam* filtered array which will display the *specifyquery* template:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   // specifying the 'query' as one of controller's query parameter
   queryParams: ['query'],

   // initialize the query value
   query: null,

   // defining a computed property queryParam
   queryParam: Ember.computed.oneWay('query'),
   actions: {
      addQuery: function () {
         // setting up the query parameters and displaying it
         this.set('query', this.get('queryParam'));
         document.write(this.get('query'));
      }
   }
});
```

Output

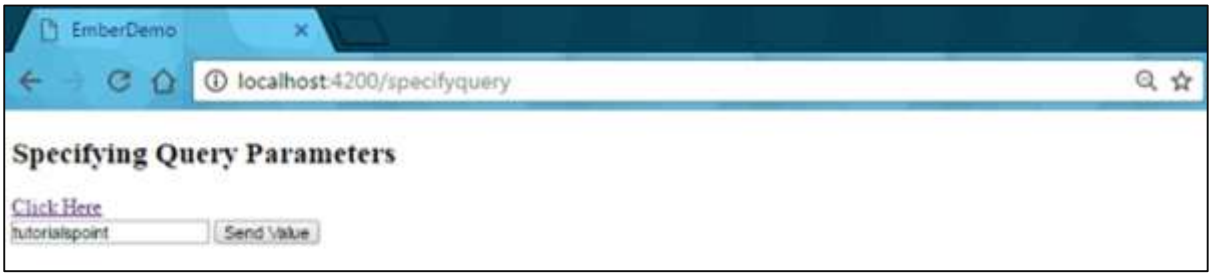
Run the ember server and you will receive the following output:



Specifying Query Parameters

Click Here

When you click on the link, it will provide an input box to enter a value and sends an action to the addQuery method:

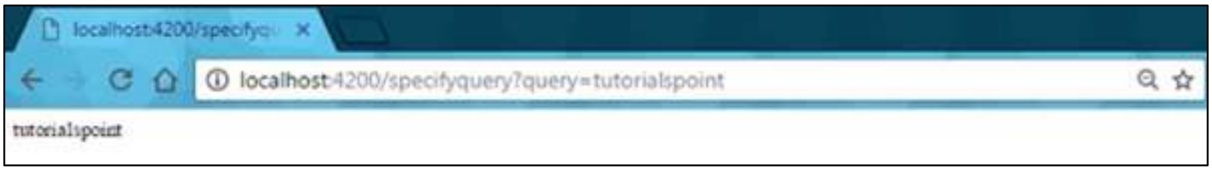


Specifying Query Parameters

Click Here

tutorialspoint

After clicking the button, it shows the key value pairs to the right of the *?* in a URL:



tutorialspoint

EmberJS – Router Opting Into a Full Transition

You can use the optional queryParams configuration when a controller query parameter property changes to opt into a full transition by setting the *refreshModel* config property to *true*. The *transitionTo* or *link-to* arguments will change in the query parameter values, but do not change in the route hierarchy; the controller properties will get updated with new query param values as well in the URL.

Syntax

```
Ember.Route.extend({
   queryParams: {
      queryParameterName: {
          refreshModel: true
      }
   }
});
```

Example

The example given below shows opting into a full transition when a controller query param property changes. Create a new route and name it as *paramfulltrans* and open the *router.js* file to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('paramfulltrans');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Opting Into a Full Transition</h2>
{{#link-to 'paramfulltrans'}}Click Here{{/link-to}}
```

When you click the above link, the page should open with an input box which takes a value entered by an user. Open the *paramfulltrans.hbs* file to opt into a full transition by using the *queryParams* configuration:

```
// sending action to the addQuery  method
<form {{action "addQuery" on="submit"}}>
 {{input value=queryParam}}
 <input type="submit" value="Send Value"/>
</form>
{{outlet}}
```

Now define the computed property of the *queryParam* filtered array which will display the *paramfulltrans* template:

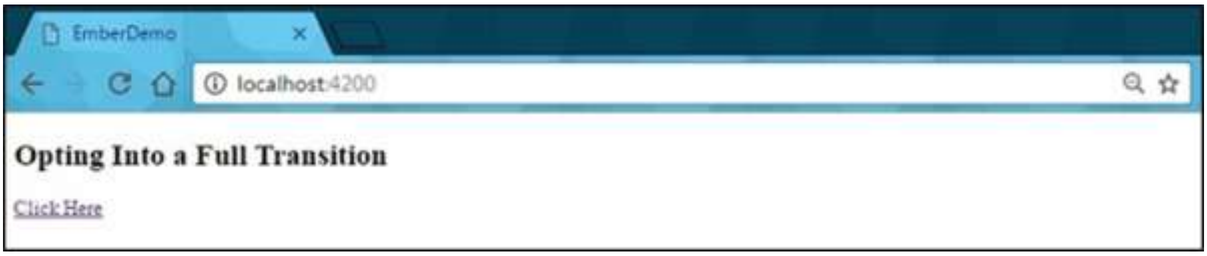
```
import Ember from 'ember';

export default Ember.Controller.extend({
   // specifying 'query' as one of controller's query parameter
   queryParams: ['query'],
   // initialize the query value
   query: null,
   // defining a computed property queryParam
   queryParam: Ember.computed.oneWay('query'),
   actions: {
      addQuery: function () {
         // setting the query parameters and displaying it
         this.set('query', this.get('queryParam'));
         document.write(this.get('query'));
      }
   }
});
```

Now use the *queryParams* configuration on the Route with the respective controller and set the *refreshModel* config property to *true* in the *paramfulltrans.js* file defined under *app/routes/*.

```
import Ember from 'ember';

export default Ember.Route.extend({
   queryParams: {
      query: {
         //opting into full transition
         refreshModel: true
      }
   }
});
```

Output

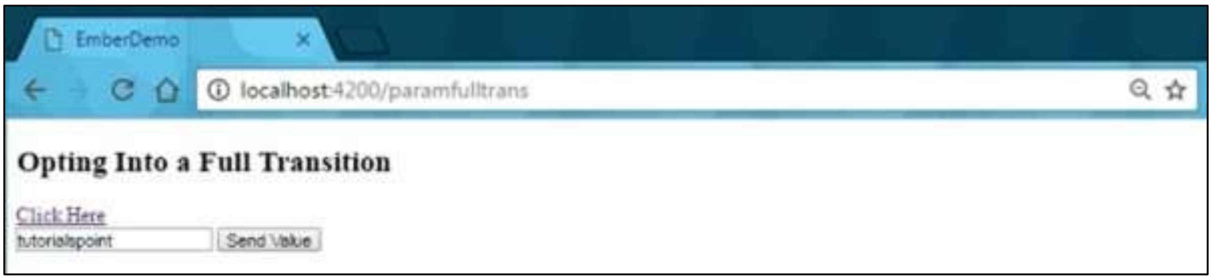
Run the ember server and you will receive the following output:



Opting Into a Full Transition

Click Here

When you click on the link, it will generate an input box wherein you can enter a value and send an action to the addQuery method:

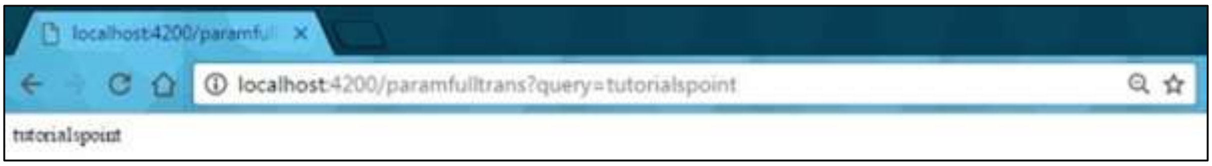


Opting Into a Full Transition

Click Here

tutorialspoint Send Value

After clicking the button, it will show the parameter value to the right of the " ? " in a URL:



tutorialspoint

EmberJS – Router Update URL with replaceState Instead

You can prevent from adding an item to your browser's history by using the *replaceState* transition. You can specify this by using the *queryParams* configuration hash on the Route and opt into a *replaceState* transition by setting the *replace* transition to true.

Syntax

```
Ember.Route.extend({
   queryParams: {
      queryParameterName: {
          replace: true
      }
   }
});
```

Example

The example given below shows how to update a URL with the replaceState transition. Create a new route and name it as *paramreplaceState* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

//The const declares read only variable
```

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

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('paramreplaceState');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Update URL with replaceState</h2>
{{#link-to 'paramreplaceState'}}Click Here{{/link-to}}
```

When you click the above link, the page should open with a button to change the URL after clicking on it. Open the *paramreplaceState.hbs* file with the following code:

```
// sending action to the addQuery  method
<button {{action 'change'}}>Replace State</button>
{{outlet}}
```

Now open the *paramreplaceState.js* file created under *app/controllers/* which is rendered by router when entering a Route:

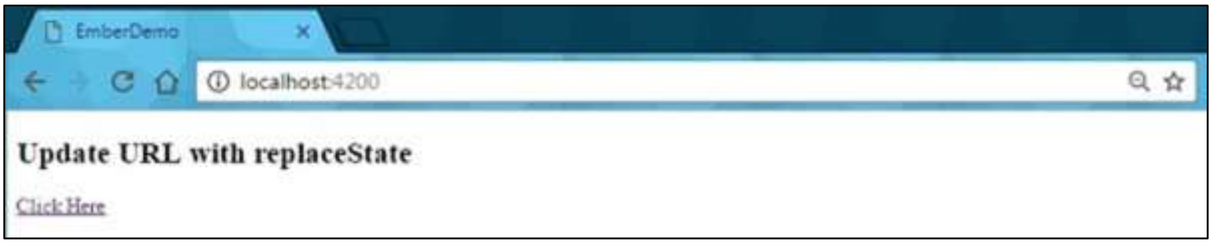
```
import Ember from 'ember';
var words = "tutorialspoint";

export default Ember.Controller.extend({
   queryParams: ['query'],
   actions: {
      change: function () {
         //assigning value of variable 'words' to the 'query' i.e. query
parameter
         this.set('query', words);
      }
   }
});
```

Now use the *queryParams* configuration on the Route with the respective controller and set the *replace* config property to *true* in the *paramreplaceState.js* file created under *app/routes/*.

```
import Ember from 'ember';


export default Ember.Route.extend({
   queryParams: {
      query: {
         //assigning replace state as true
         replace: true
      }
   }
});
```

Output

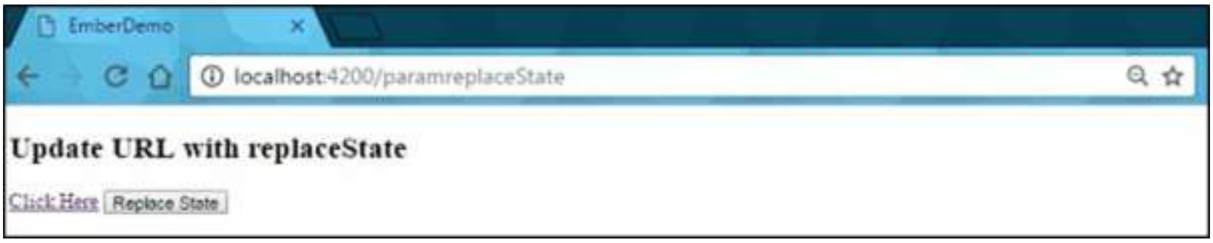
Run the ember server and you will receive the following output:



Update URL with replaceState

Click Here

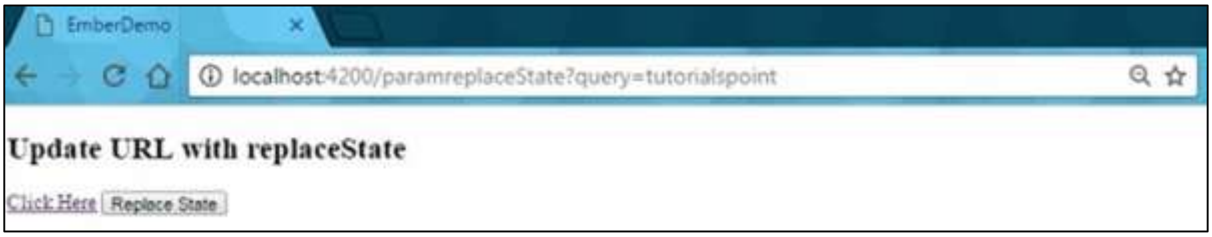
When you click on the link, it displays the button which sends an action to the addQuery method:



Update URL with replaceState

Click Here Replace State

After clicking on the button, it will show the parameter value to the right of the "***?***" in a URL:



Update URL with replaceState

Click Here Replace State

EmberJS – Router Map a Controller's Property to a Different Query Param Key

The controller has a default query parameter property which attaches a query parameter key to it and maps a controller property to a different query parameter key.

Syntax

```
Ember.Controller.extend({
   queryParams: {
      queryParamName: "Values"
   },
   queryParamName: null
});
```

Example

The example given below shows mapping a controller's property to a different query param key. Create a new route and name it as *parammapcontrol* and open the *router.js* file to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('parammapcontrol');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Map a Controller's Property</h2>
{{#link-to 'parammapcontrol'}}Click Here{{/link-to}}
```

When you click the above link, the page should open with an input box which takes a value entered by an user. Open the *parammapcontrol.hbs* file and add the following code:

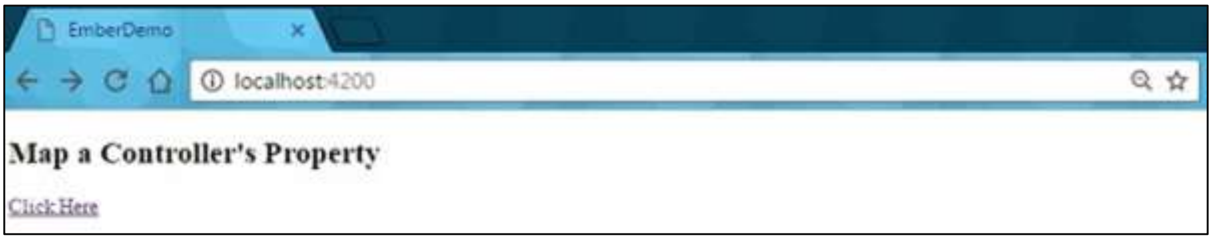
```
// sending action to the addQuery  method
<form {{action "addQuery" on="submit"}}>
 {{input value=queryParam}}
 <input type="submit" value="Send Value"/>
</form>
{{outlet}}
```

Now open the *parammapcontrol.js* file created under *app/controllers/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   queryParams: [{
      // mapping the string 'querystring' of the 'query's' query parameter
      query: "querystring"
   }],
   // initialy query's 'query parameter' will be null
   query: null,

   queryParam: Ember.computed.oneWay('query'),
   actions: {
      addQuery: function () {
         this.set('query', this.get('queryParam'));
         document.write(this.get('query'));
      }
   }
});
```

Output

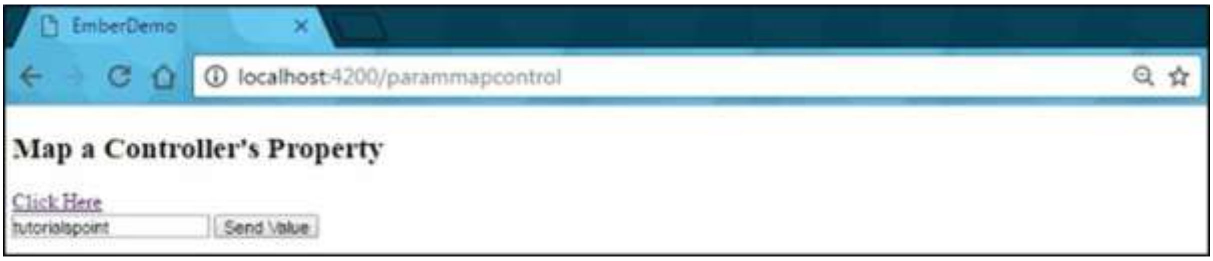
Run the ember server and you will receive the following output:



Map a Controller's Property

Click Here

When you click on the link, it will generate an input box wherein you can enter a value. This will send an action to the addQuery method:

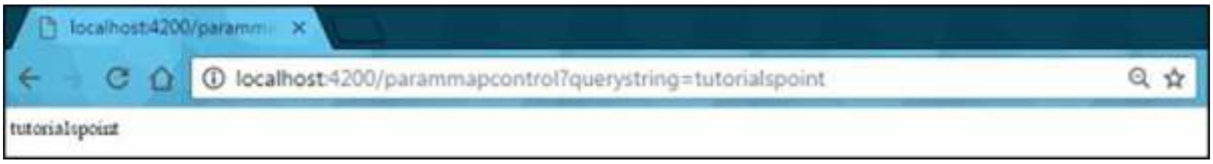


Map a Controller's Property

Click Here

tutorialspoint

After clicking the button, it will show the parameter value to the right of the "?" in a URL:



tutorialspoint

EmberJS – Router Default Values and Deserialization

You can set the default value for the controller query parameter property the value of which will not be serialized into the URL.

Syntax

```
Ember.ArrayController.extend({
   queryParams: 'queryParameterName',
   queryParameterName: defaultValue
});
```

Example

The example given below specifies setting the default value to the query parameter. Create a new route and name it as *defaultvaluedeserialize* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});
```

```
// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('defaultvaluedeserialize');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Default Values and Deserialization</h2>
{{#link-to 'defaultvaluedeserialize'}}Click Here{{/link-to}}
```

When you click the above link, the page should open with a input box which takes a value entered by the user. Open the *defaultvaluedeserialize.hbs* file and add the following code:

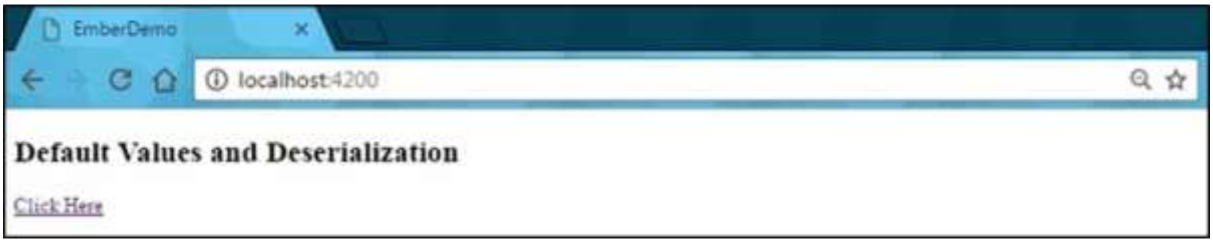
```
// sending action to the addQuery  method
<form {{action "addQuery" on="submit"}}>
 {{input value=queryParam}}
 <input type="submit" value="Send Value"/>
</form>
{{outlet}}
```

Now open the *defaultvaluedeserialize.js* file created under *app/controllers/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   // assigning query parameter name as 'query'
   queryParams: 'query',
   // assigning the query param to a default value as 1
   query: 1,

   queryParam: Ember.computed.oneWay('query'),
   actions: {
      addQuery: function () {
         this.set('query', this.get('queryParam'));
         document.write(this.get('query'));
      }
   }
});
```

Output

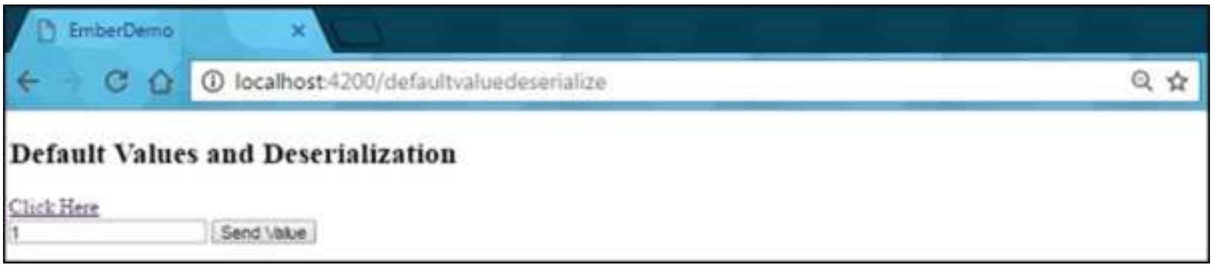
Run the ember server and you will receive the following output:



Default Values and Deserialization

Click Here

When you click on the link, it displays the specified default value in the input box:

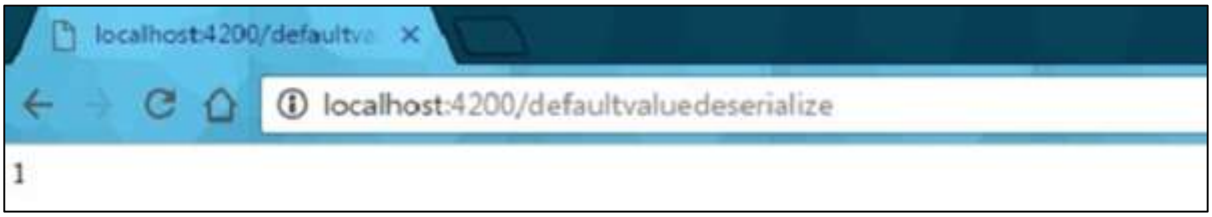


Default Values and Deserialization

Click Here

1

After clicking the button, it will show the default value and it won't be serialized into the URL:



1

EmberJS – Router Sticky Query Param Values

In Ember, the query parameter values are sticky by default; in a way that any changes made to the query parameter, the new value of query parameter will be preserved by re-entering the route.

Syntax

```
Ember.Controller.extend({
   queryParams: ['paramValue'],
   paramValue:true/false
});
```

Example

The example given below specifies the use of sticky query parameter values. Create a new route and name it as *stickyqueryparam* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('stickyqueryparam');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* created under *app/templates/* with the following code:

```
<h2>Sticky Query Param Values</h2>
{{#link-to 'stickyqueryparam'}}Click here to open the page{{/link-to}}
```

When you click the above link, it opens the sticky query parameter template page. The *stickyqueryparam.hbs* file contains the following code:

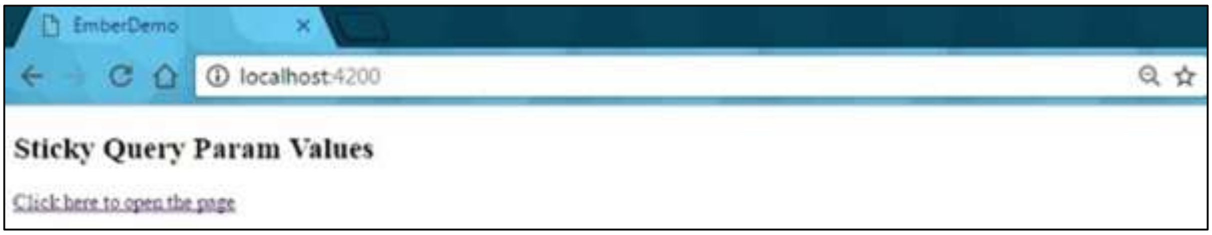
```
<h2>My Page</h2>
{{link-to 'Show' (query-params showThing=true)}}
{{link-to 'Hide' (query-params showThing=false)}}
<br>
{{#if showThing}}
    <b>Welcome to Tutorialspoint..</b>
{{/if}}
{{outlet}}
```

Now open the *stickyqueryparam.js* file created under *app/controllers/* with the below code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   queryParams: ['showThing'],

   // showThing would be false, if only the route's model is changing
   showThing: false
});
```

Output

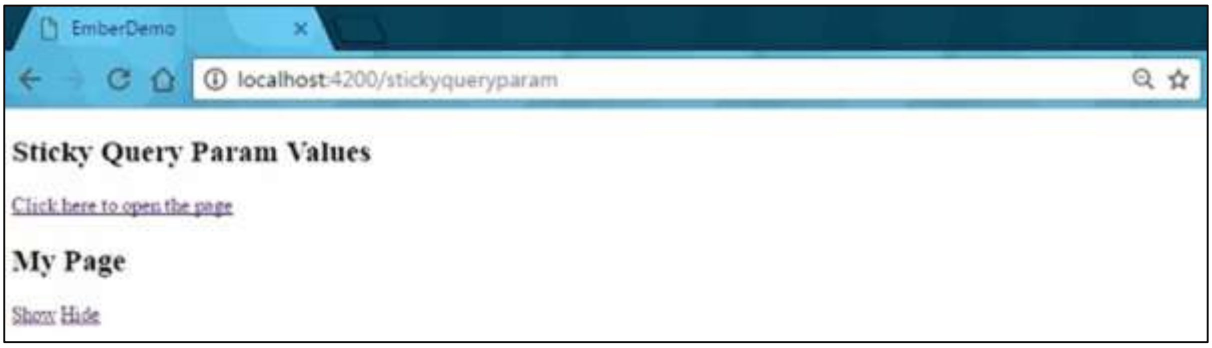
Run the ember server and you will receive the following output:



Sticky Query Param Values

Click here to open the page

When you click on the link, it will open the sticky query param template page by providing *Show* and *Hide* links:



Sticky Query Param Values

Click here to open the page

My Page

Show Hide

EmberJS – Asynchronous Routing

The Ember.js router has ability to handle complex async logic within an application by using asynchronous routing.

The table given below shows the different types of handling asynchronous logic in the router:

S.NO.	Async Routers & Description
1	**The Router Pauses for Promises** The transition can be paused by returning a promise from the model hook.
2	**When Promises are Rejected** The transition will be aborted if a promise is rejected by the model during a transition.
3	**Recovering from Rejection** Recovering from the aborted transition.

EmberJS – Router Pauses for Promises

The transition can be paused by returning a promise from the model hook. The transition can be completed immediately by returning normal objects or arrays from the model.

Syntax

```
Ember.Route.extend({
   model() {
      return new Ember.RSVP.Promise(function(param) {
         //code here
      });
   }
});
```

Example

The example given below shows how transition will pause if the model returns a promise. Create a new route and name it as *promisepause* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
//It provides access to app's configuration data as variable config
```

```
// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('promisepause');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Router Pauses for Promises</h2>
{{#link-to 'promisepause'}}Click Here{{/link-to}}
```

When you click the above link, it will open the promise pause template page. The *promisepause.hbs* file contains the following code:

```
{{model.msg}}
{{outlet}}
```

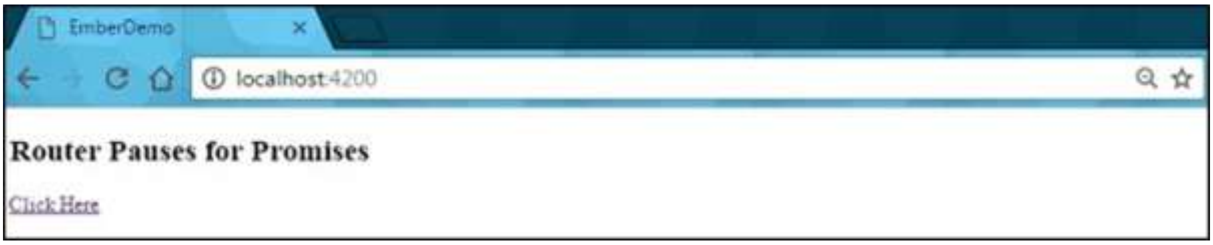
Now open the *promisepause.js* file created under *app/routes/* with the following code:

```
import Ember from 'ember';
import RSVP from 'rsvp';

export default Ember.Route.extend({
   model() {
      // RSVP.js is an implementation of Promises
      return new Ember.RSVP.Promise(function (resolve) {
         Ember.run.later(function () {
            // display the promise message
            resolve({
               msg: "This is Promise Message..."
            });
```

```
        }, 3000);       // time in milli second to display promise
      });
   }
});
```

Output

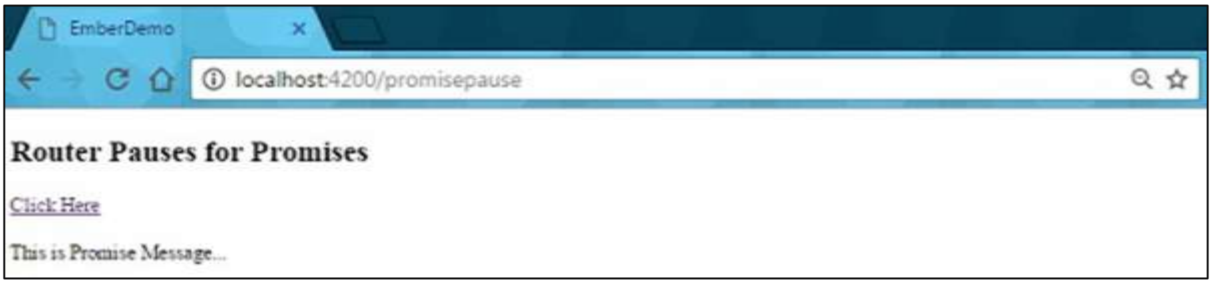
Run the ember server and you will receive the following output:



Router Pauses for Promises

Click Here

When you click on the link, model returns the promise which is not resolved until 3 seconds and when the promise fulfills, the router will start transitioning:



Router Pauses for Promises

Click Here

This is Promise Message...

EmberJS – Router When Promises Reject

The transition will be aborted if a promise is rejected by the model during a transition and there will be no display of new destination route templates and no error message in the console.

Syntax

```
Ember.Route.extend({
   model() {
      //code here
   },

   actions: {
      error: function(reason) {
        // display or return the "Failure Message"
      }
   }
});
```

Example

The example given below shows how transition will be aborted if model rejects the promise. Create a new route and name it as *promisereject* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('promisereject');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

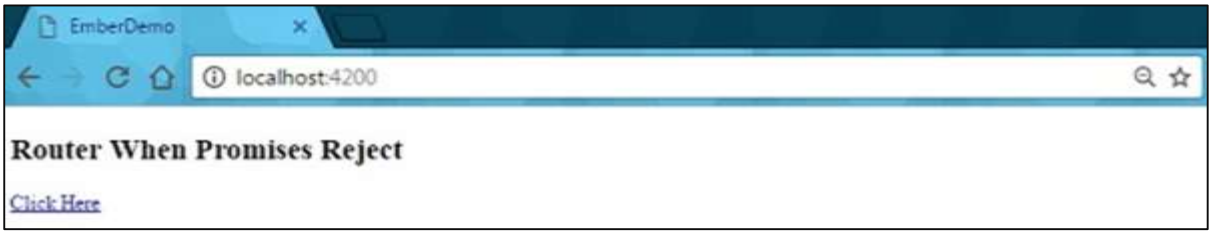
```
<h2>Router When Promises Reject</h2>
{{#link-to 'promisereject'}}Click Here{{/link-to}}
```

Now open the *promisereject.js* file created under *app/routes/* with the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function () {
      // RSVP.js is an implementation of Promises
      return Ember.RSVP.reject("Failure of promises");
   },
   actions: {
      // actions for displaying failure of promises using error hook and it
takes reason as parameter
      error: function (reason) {
         document.write("<h3>" + reason + "</h3>");
      }
   }
});
```

Output

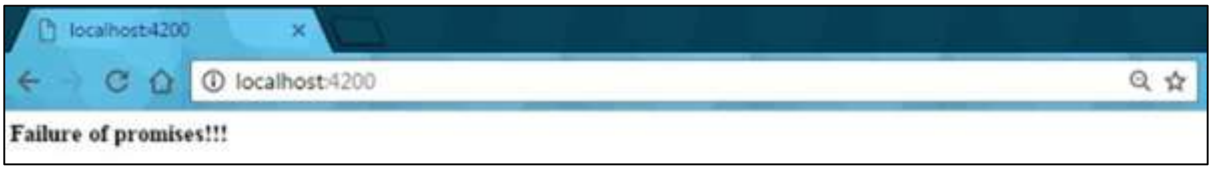
Run the ember server and you will receive the following output:



Router When Promises Reject

Click Here

When you click on the link, no new route templates will be rendered and it will display a failure message:



Failure of promises!!!

EmberJS – Router Recovering from Rejection

The promise rejects can be cached within the model hook which can be converted into fulfills that will not put the transition on halt.

Syntax

```
Ember.Route.extend({
   model() {
      // return the recovery message
   }
});
```

Example

The example given below shows how transition will be aborted if model rejects the promise. Create a new route and name it as *promisereject* and open the *router.js* file to define URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('recoveryrejection');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Recovering from Rejection</h2>
{{#link-to 'recoveryrejection'}}Click Here{{/link-to}}
```

When you click the above link, it will open the recoveryrejection template page. The *recoveryrejection.hbs* file contains the following code:

```
{{model.msg}}
{{outlet}}
```

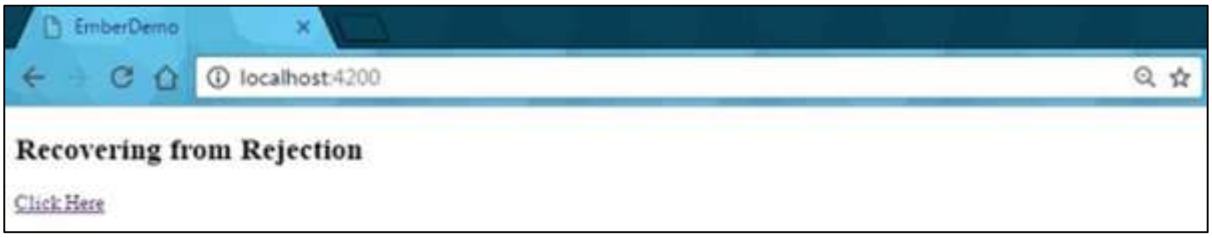
Now open the *recoveryrejection.js* file created under *app/routes/* with the following code:

```
import Ember from 'ember';
import RSVP from 'rsvp';
export default Ember.Route.extend({
   model() {
      // returning recovery message
      return {
         msg: "Recovered from rejected promise"
```

```
          };
      }
 });
```

Output

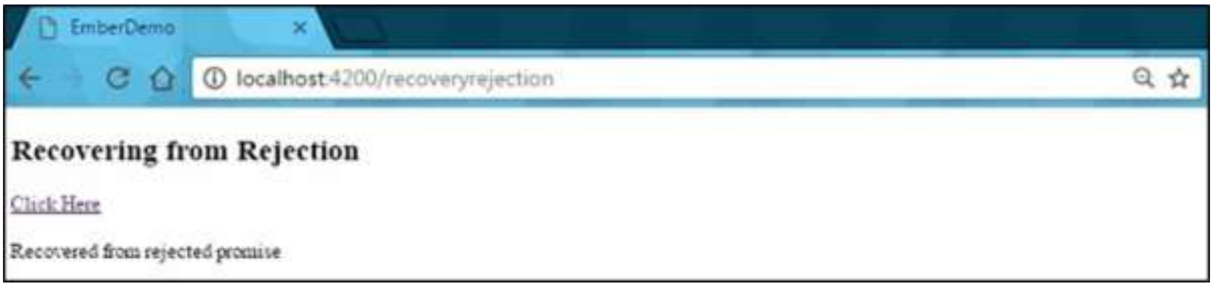
Run the ember server and you will receive the following output:



Recovering from Rejection

Click Here

When you click on the link, promise will be rejected and it will display a recovery message to continue with the transition:



Recovering from Rejection

Click Here

Recovered from rejected promise

7. EmberJS – Templates

A template is used to create a *standard layout* across multiple pages. When you change a template, the pages that are based on that template automatically get changed. Templates provide *standardization controls*.

The below table shows some more details about templates:

S.NO.	Types & Description
1	**Handlebars Basics** The Handlebars templating library allows building rich user interface by including static HTML and dynamic content.
2	**Built-in Helpers** Helpers provide extra functionality to the templates and modifies the raw value from models and components into proper format for users.
3	**Conditionals** Ember.js defines the two conditional statements which help to control the flow of program.
4	**Displaying List of Items** You can display the list of items in an array by using the *#each* helper.
5	**Displaying Keys in an Object** You can display the keys in the object by using the *#each-in* helper.
6	**Links** The *{{link-to}}* component can be used to create a link to a route.
7	**Actions** The HTML element can be made clickable by using the *{{action}}* helper.
8	**Input Helpers** The common form controls can be created by using the *{{input}}* and *{{textarea}}* helpers in the Ember.js
9	**Development Helpers** The template developement can be made easier by using some helpers of Handlebars and Ember.

10	**Writing Helpers** You can add extra functionality to the templates and converts the raw values from models and components into proper format for the users.

EmberJS – Handlebars Basics

The Handlebars templating library allows building rich user interface by including static HTML and dynamic content, which can be specified in the double curly braces - {{}}.

Syntax

```
 Ember.Controller.extend({
  property1: value,
  property2: value,
  .....
  propertyn: valuen,
});
```

Example

The following example shows how to display properties from the application controller. Create a template called *application.hbs* under *app/templates/* with the following code:

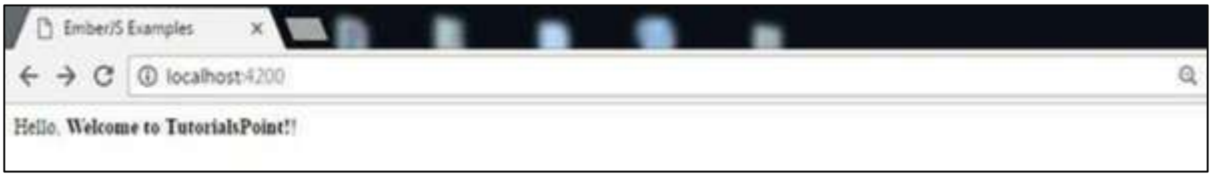
```
// displaying the values of firstSentence and lastSentence
Hello, <strong>{{firstSentence}} {{lastSentence}}</strong>!
```

Now create the controller with the same name (template file) to add the properties. The *application.js* file will be created under *app/controller/* with the following code:

```
import Ember from 'ember';


export default Ember.Controller.extend({
  // initializing values
  firstSentence: 'Welcome to',
  lastSentence: 'TutorialsPoint!'
});
```

Output

Run the ember server and you will receive the following output:



Hello, Welcome to TutorialsPoint!!

We will see the detail concept of *Helpers* in the Writing Helpers chapter.

EmberJS – Built-in Helpers

Helpers provide extra functionality to the templates and modify the raw value from models and components into proper format for users.

Getting Property Dynamically

You can make use of *{{}}* helper to send the variable value to another helper or component. It is also helpful for displaying various values based on the computed property result. For instance:

```
{{get icc part}}
```

If we take the *part* as *teamrank*, then computed property returns the result of *this.get('icc.teamrank')*.

Nesting Built-in Helpers

Helpers can be used to nest one helper within other helper and also with component invocations. The helpers can be nested by using the parentheses *()* and not by curly braces *{{}}*. For instance:

```
{{sum (addition 4 5) 2}}
```

Here, helper can be used to add 4 and 5 before passing the value to {{sum}}. It will display the result as "11".

EmberJS – Conditionals

The Ember.js defines the two conditional statements, which helps to control the flow of program. It begins with the #(Hash) before the helper name and ends with the closing expression, i.e., *{{/}}* double curly brace.

The following table shows conditional statments of Ember.js:

S.NO.	Condition Statment & Description
1	**#if** It is used to display a block of the template.

2	**#unless** It executes only falsy block of statements.

EmberJS – Template Condition If

The *#if* statement uses a boolean expression wherein, if the Boolean expression is true, then the block of code inside the **if** statement will be executed; if the Boolean expression is false, then the **else** block will be executed.

Syntax

```
{{#if property-name}}
   //statement
{{else}}
   //statement
{{/if}}
```

Example

The example given below shows the use of the if conditional helper in Ember.js. Create a template called *application.hbs* under *app/templates/* with the following code:

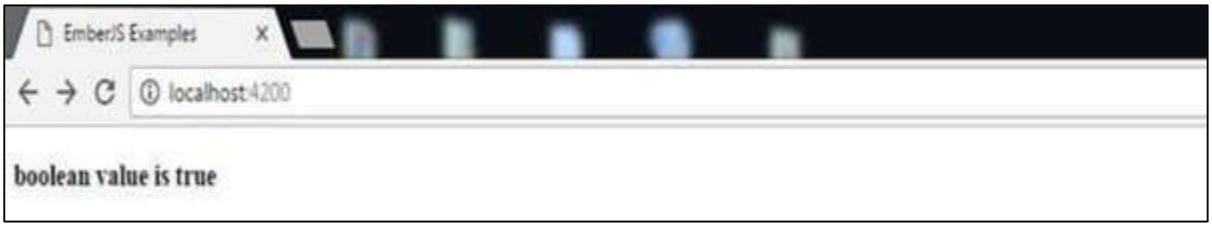
```
{{#if check}}
    // true block of statement
    <h3> boolean value is {{check}}</h3>
    {{else}}
    // false block of statement
    <h3>boolean value is {{check}}</h3>
{{/if}}
```

Next, create the controller called *application.js* file which will be defined under *app/controller/* with the following code:

```
import Ember from 'ember';
export default Ember.Controller.extend({
   bool: true,
   check: function () {
      // returning the boolean value to the called function
      return this.bool;
   }.property('content.check'),
});
```

Output

Run the ember server and you will receive the following output:



boolean value is true

EmberJS – Template Condition Unless

It executes only false block of statements.

Syntax

```
{{#unless falsy_condition}}
   //block of statement
{{/unless}}
```

Example

The example given below shows the use of the unless conditional helper in the Ember.js. Create a template called *application.hbs* under *app/templates/* with the following code:

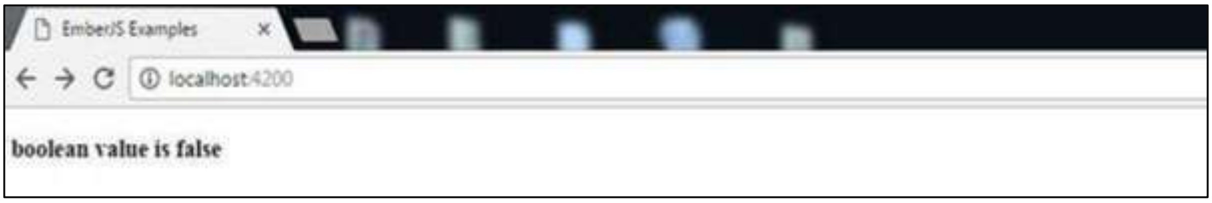
```
{{#unless check}}
    <h3> boolean value is {{check}}</h3>
{{/unless}}
```

Now create the controller called *application.js* file, which will be defined under *app/controller/* with the following code:

```
import Ember from 'ember';


export default Ember.Controller.extend({
   bool: false,
   check: function () {
      return this.bool;
   }.property('content.check')
});
```

Output

Run the ember server and you will receive the following output:



boolean value is false

EmberJS – Displaying a List of Items

You can display the list of items in an array by using the *#each* helper and it iterates once for each item present in an array.

Syntax

```
<ul>
   {{#each array_name as |block-param| }}
      <li>{{block-param}}</li>
   {{/each}}
</ul>
```

In the above code, template iterates *array_name*, which includes objects and each item in the array specified as *block-param*.

Example

The example given below displays a list of items by using the *#each* helper. To display the items, create a component by using the following command:

```
ember g component group-list
```

Next, open the *group-list.js* created under *app/component/* along with the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
     arrayOFgroup:['apple','pineapple','banana']
});
```

Create a template called *group-list.hbs* under *app/templates/* with the following code:

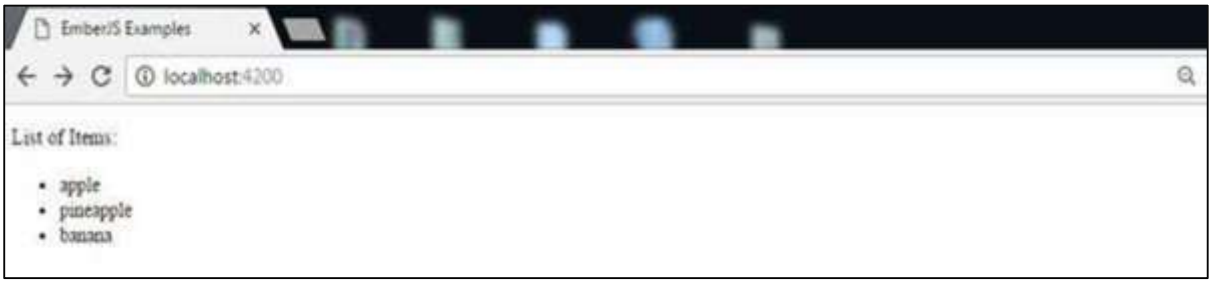
```
<ul>
  {{#each arrayOFgroup as |fruit|}}
    <li>{{fruit}}</li>
  {{/each}}
</ul>
```

To list the items from an array, use the below code in the *application.hbs* file created under *app/templates/*:

```
<p>List of Items:</p>
{{group-list}}
{{outlet}}
```

Output

Run the ember server and you will receive the following output:



List of Items:

- apple
- pineapple
- banana

EmberJS – Displaying Keys in an Object

You can display the keys in the object by using the *#each-in* helper and it iterates once for each key passed in object.

Syntax

```
<ul>
   {{#each-in array_name as |block-param| }}
      <li>{{block-param}}</li>
   {{/each}}
</ul>
```

In the above code, template iterates *array_name*, which includes objects and each key in the object specified as *block-param*.

Example

The example given below displays the keys in the object by using the *#each-in* helper. To display the items, create a component by using the following command:

```
ember g component store-categories
```

Now open the *store-categories.js* created under *app/component/* along with the following code:

```
import Ember from 'ember';

export default Ember.Component.extend({
   willRender() {
      this.set('typesOfvehicles', {
         'Cars': ['Ferrari', 'Audi', 'BMW'],
         'Motor bikes': ['Harley-Davidson', 'Yamaha','Honda']
       });
   }
});
```

Create a template called *store-categories.hbs* under *app/templates/* with the following code:

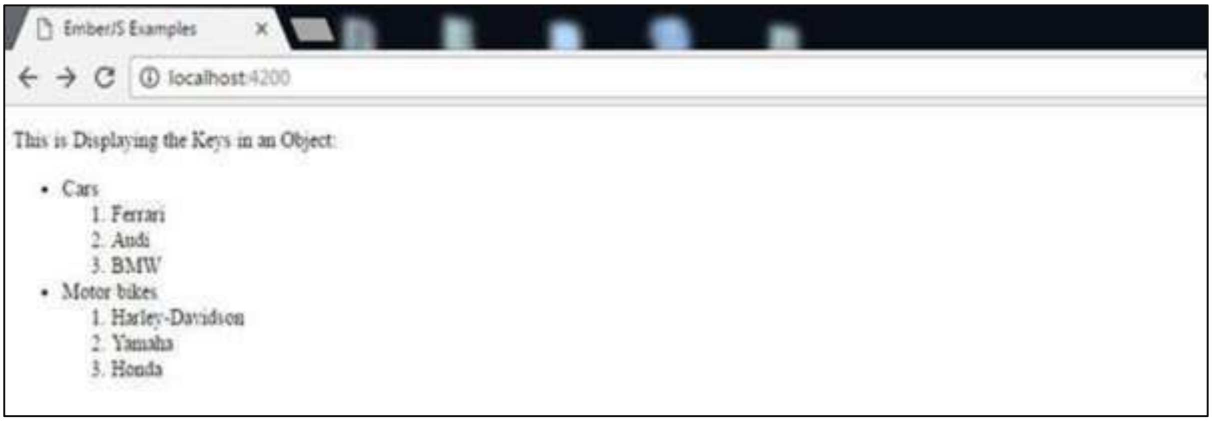
```
<ul>
  {{#each-in typesOfvehicles as |category products|}}
    <li>{{category}}
      <ol>
        {{#each products as |product|}}
          <li>{{product}}</li>
        {{/each}}
      </ol>
    </li>
  {{/each-in}}
</ul>
```

To list the keys in the object, use the following code in the *application.hbs* file created under *app/templates/*:

```
<p>This is Displaying the Keys in an Object:</p>
{{store-categories}}
{{outlet}}
```

Output

Run the ember server and you will receive the following output:



This is Displaying the Keys in an Object:

- Cars
  1. Ferrari
  2. Audi
  3. BMW
- Motor bikes
  1. Harley-Davidson
  2. Yamaha
  3. Honda

EmberJS – Links

The *{{link-to}}* component can be used to create a link to a route.

Syntax

```
{{#link-to route}}
   //code here
{{/link-to}}
```

The following table lists down the properties of the links:

S.NO.	Links & Description
1	**Multiple Segments** For multiple segments, you can provide model or an identifier for each segment if the route is nested.
2	**Using Link-to as an Inline Helper** Use the *link-to* as an inline component by providing link text as the first argument to the helper.
3	**Adding Additional Attributes on a Link** You can add the additional attributes on a link while creating it.
4	**Replacing History Entries** You can add entries to the browser's history while moving between the routes by using the *link-to* helper.

Example

The following example shows how to link to a different routes. Create a new route and name it as *note* and open the *router.js* file to define the URL mappings:

```
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('note');
});

export default Router;
```

Open the *application.hbs* file created under *app/templates/* with the following code:

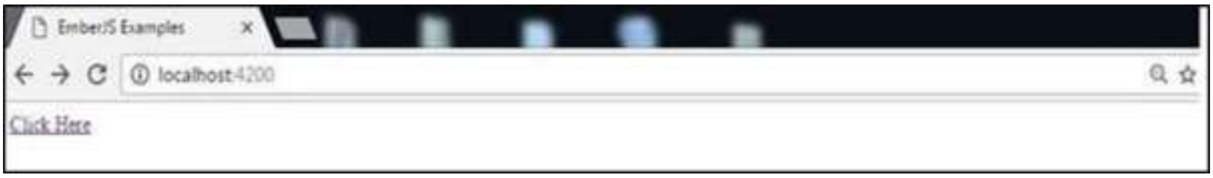
```
{{#link-to 'note'}}Click Here{{/link-to}}
{{outlet}}
```

When you click the above link, page should open the *note.hbs* file with the following text:

```
<h4>Welcome to TutorialsPoint</h4>
{{outlet}}
```

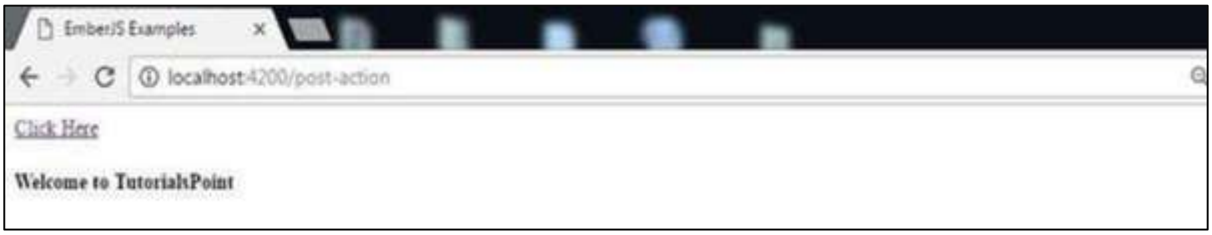
Output

Run the ember server and you will receive the following output:



Click Here

When you click on the link, it will display the text from the template file as in the following screenshot:



Click Here

Welcome to TutorialsPoint

EmberJS – Template Multiple Segments

For multiple segments, you can provide model or an identifier for each segment if the route is nested.

Syntax

```
Router.map(function() {
   this.resource('route_name');
   this.resource('route_name', { path: 'route_path' });
});
```

Example

The example shows the use of multiple segments in the nested route by providing an identifier to the segment. Create two routes with the names as *info* and *record* and open the *router.js* file to define the URL mappings:

```
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('info');
  this.route('record', { path: 'records/:records_id' });
});

export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
{{#link-to 'info'}}Fruits{{/link-to}}
{{#link-to 'record' recoModel}}Some Record{{/link-to}}
{{outlet}}
```

When you click the "Fruits" link, page should open the *info.hbs* file, which contains the following code:

```
<p>Some Fruits</p>
<ul>
   <li>Orange</li>
   <li>Banana</li>
</ul>
{{outlet}}
```

If you click on the *Some Record* link, page should open the *record.hbs* file, which contains the following code:

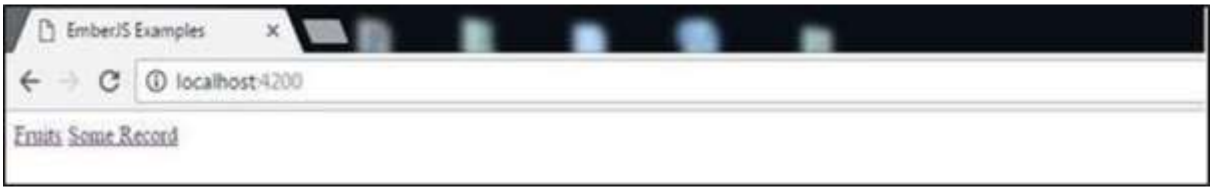
```
<p>Some Records</p>
{{model.name}}
{{outlet}}
```

Now create the controller *application.js*, which will be created under *app/controller/* to with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   recoModel: function(){
      // return the records value to the called route
      return {records_id:1, name:'Docs List'};
   }.property()
});
```

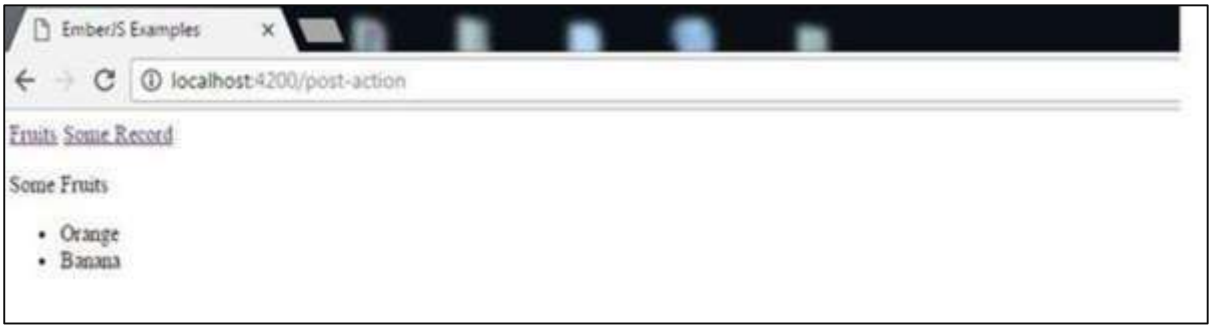
Output

Run the ember server; you will receive the following output:



Fruits Some Record

When you click on the *Fruits* link, it will display the following text from the template file:

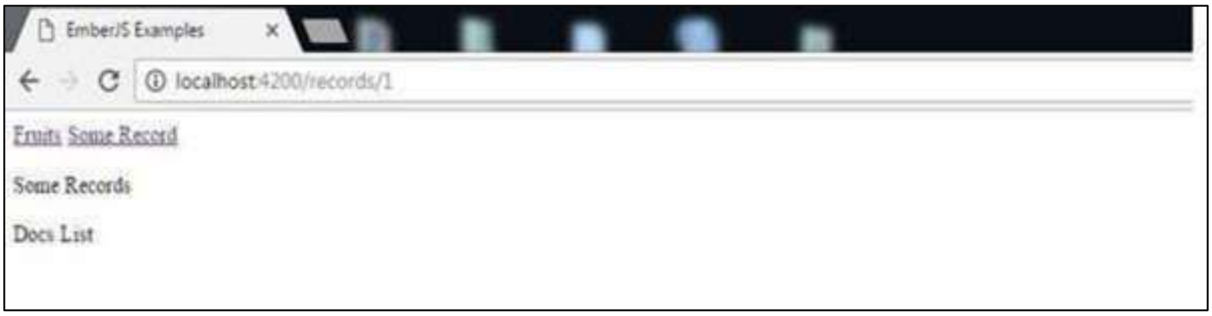


Fruits Some Record

Some Fruits

- Orange
- Banana

When you click on the *Some Record* link, it will display the following text from the template file:



Fruits Some Record

Some Records

Docs List

EmberJS – Template link-to as Inline Helper

You can use the *link-to* as an inline component by providing link text as the first argument to the helper.

Syntax

```
Click for {{#link-to 'link1'}}more info{{/link-to}},
info of {{link-to 'link text' 'link2'}}.
```

Example

The example given below shows the use of *link-to* as an inline component by specifying the first argument to the helper. Create two routes with the names as *info* and *record* and open the *router.js* file to define the URL mappings:

```
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('info');
  this.route('record');
});
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
Click for the {{#link-to 'info'}}Fruits{{/link-to}} information, for the documentation
{{link-to 'Click for records''record'}}
{{outlet}}
```

When you click the "Fruits" link, page should open the *info.hbs* file which contains the following code:

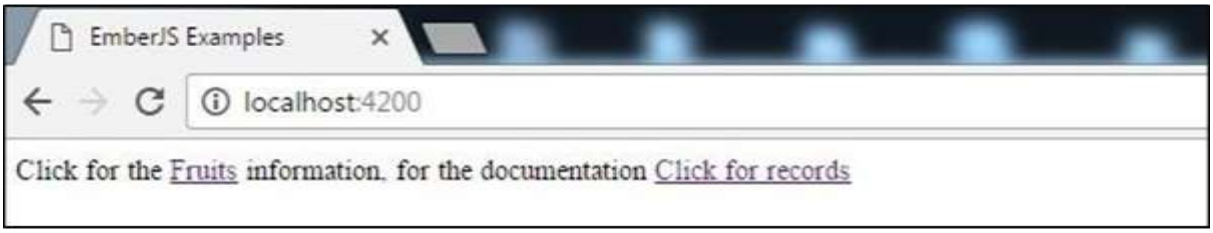
```
<p>Some Fruits</p>
<ul>
   <li>Orange</li>
   <li>Banana</li>
</ul>
{{outlet}}
```

If you click on the *Click for records* link, page should open the *record.hbs* file which contains the following code:

```
<p>Some Records</p>
<ul>
   <li>Orange.doc</li>
   <li>Banana.doc</li>
</ul>
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Click for the Fruits information, for the documentation Click for records

EmberJS – Template Additional Attributes on a Link

You can add the additional attributes on a link with the help of *link-to* helper.

Syntax

```
{{link-to 'link-text' 'route-name' class="btn-primary"}}
```

Example

The example shows how to add additional attributes on a link. Create a route with the name as *info* and open the *router.js* file to define the URL mappings:

```
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('info');
});

export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

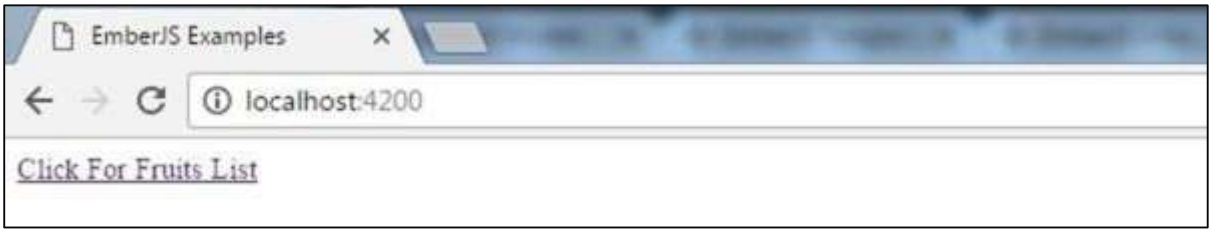
```
// Adding the additional attributes on a link as class="btn btn-primary"
{{link-to 'Click For Fruits List' 'info' class="btn btn-primary"}}
{{outlet}}
```

When you click the "Click For Fruits List" link, page should open the *info.hbs* file which contains the following code:

```
<p>Some Fruits</p>
<ul>
   <li>Orange</li>
   <li>Banana</li>
</ul>
{{outlet}}
```

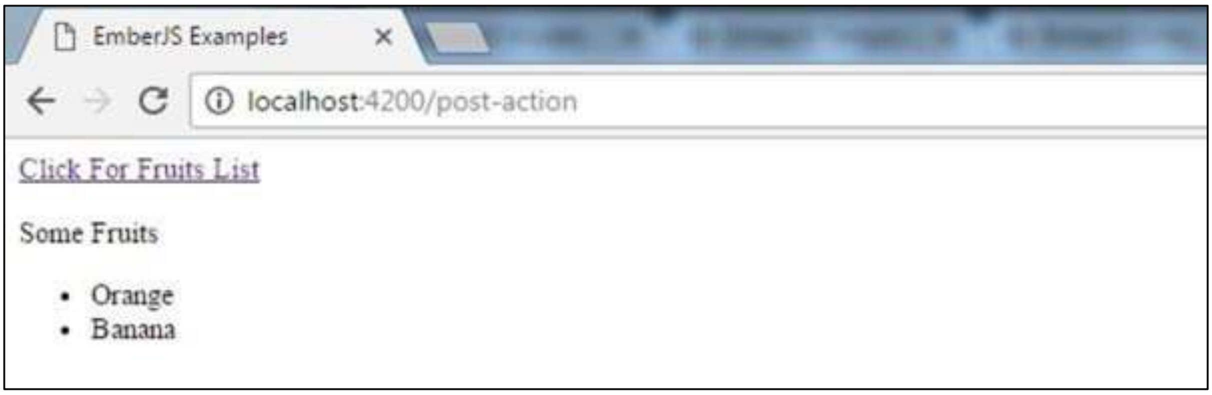
Output

Run the ember server; you will receive the following output:



Click For Fruits List

Next, click on the *Click For Fruits List*, it will display the following text from the template file:



Click For Fruits List

Some Fruits

- Orange
- Banana

EmberJS – Template Replacing History Entries

You can add entries to the browser's history while moving between the routes by using the *link-to* helper and replace the current entry by using the *replace=true* option.

Syntax

```
{{#link-to 'link-text' 'route-name' replace=true}}
   //text here
{{/link-to}}
```

Example

The example shows how to replace the current entry in the browser's history. Create a route with the name as *info* and open the *router.js* file to define the URL mappings:

```
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('info');
});

export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

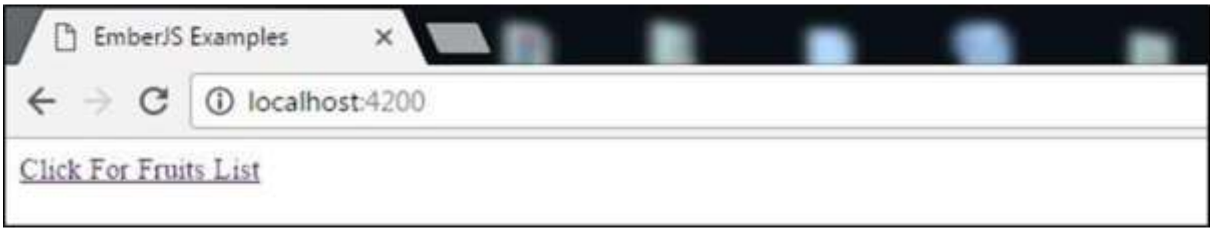
```
//put the replace=true option to replace the browser history entries
{{link-to 'Click For Fruits List' 'info' replace=true}}
{{outlet}}
```

When you click on "Click For Fruits List" link, page should open the *info.hbs* file, which contains the following code:

```
<ul>
   <li>Orange</li>
   <li>Banana</li>
</ul>
{{outlet}}
```

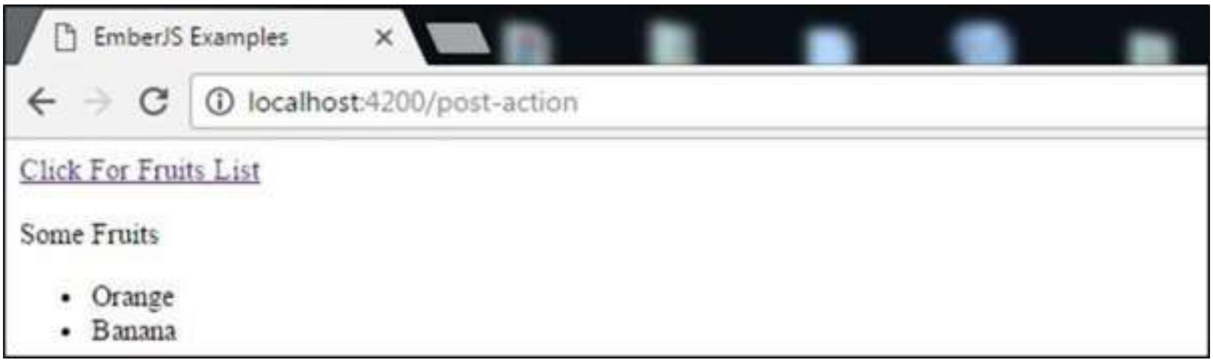
Output

Run the ember server; you will receive the following output:



Click For Fruits List

When you click on *Click For Fruits List*, it will display the following text from the template file:



Click For Fruits List

Some Fruits

- Orange
- Banana

EmberJS – Actions

The *{{action}}* helper class is used to make the HTML element clickable and action will be forwarded to the application when the user clicks an event.

Syntax

```
<button {{action 'action-name'}}>Click</button>
```

The above code adds the button *Click* to your application in which the action will be forwarded to the specified action method when a user clicks the button.

The following table lists down the action events of actions along with their description:

S.NO.	Action Events & Description
1	**Action Parameters** The arguments can be passed to an action handler with the help of *{{action}}* helper.
2	**Specifying the Type of Event** The alternative event can be specified on *{{action}}* helper by using the *on* option.
3	**Allowing Modifier Keys** You can allow modifier keys along with the *{{action}}* helper by using the *allowedKeys* option.
4	**Modifying Action's first Parameter** You can modify the action's first parameter by specifying a *value* option for the *{{action}}* helper.

Example

The example given below shows the usage of *{{action}}* helper to make the HTML element clickable and action will be forwarded to the specified action method. Create a component with name *post-action* by using the following command:

```
ember g component post-action
```

Open the *post-action.js* file created under *app/component/* and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
  actions: {
    toggleBody() {
      this.toggleProperty('isShowingBody');
    }
  }
});
```

Open the file *post-action.hbs* file created under *app/templates/* with the following code:

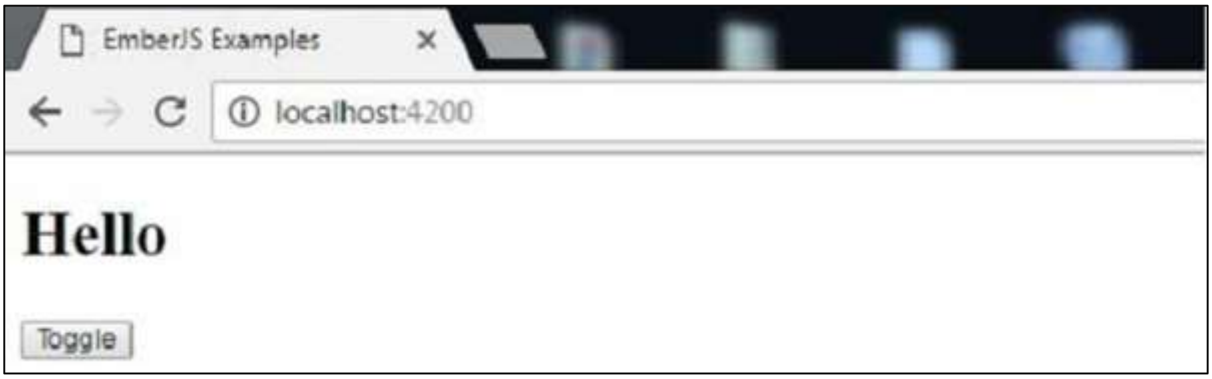
```
<h1>Hello</h1><h3><button {{action "toggleBody"}}>{{title}}Toggle</button></h3>
{{#if isShowingBody}}
  <h2>Welcome To Tutorials Point</h2>
{{/if}}
{{yield}}
```

In the *index.hbs* file, copy the below code which is created under *app/templates/*:

```
{{post-action}}
{{outlet}}
```

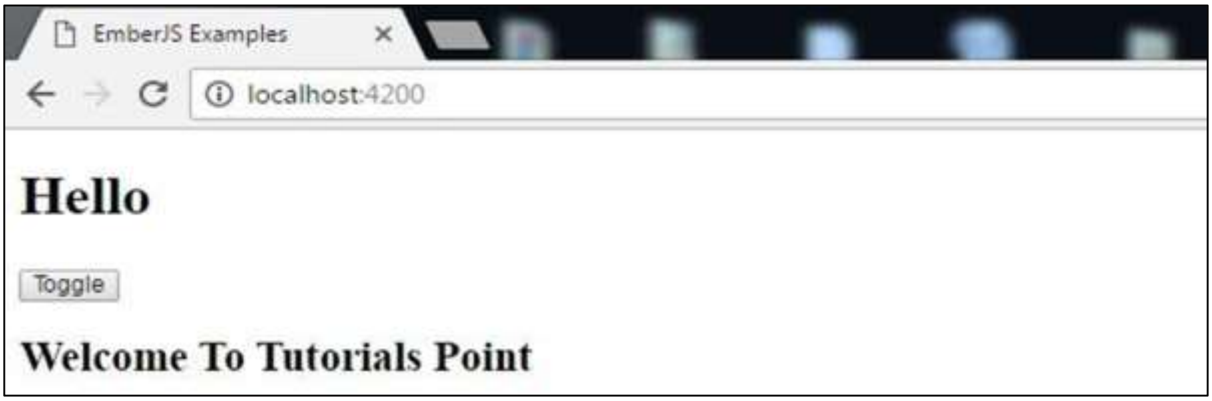
Output

Run the ember server; you will receive the following output:



Hello

When you click on the *Toggle* button, it will display the following text from the template file:



Hello

Welcome To Tutorials Point

EmberJS – Template Action Parameter

The arguments can be passed to an action handler with the help of *{{action}}* helper. These values passed with this helper will be passed as arguments to the helper.

Syntax

```
<button {{action "action-name" argument}}>Click</button>
```

Example

The example given below shows passing arguments to the action handler. Create a new route and name it as *actionparam.js* with the following code:

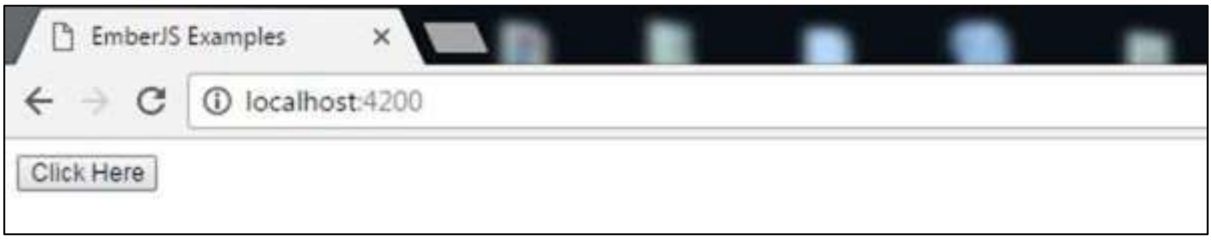
```
import Ember from 'ember';
export default Ember.Route.extend({
   actions: {
      // passing the 'user' as parameter to the User function
      User: function (user) {
         document.write('Welcome.. To Tutorialspoint');
      }
   }
});
```

Open the *actionparam.hbs* file created under *app/templates/* with the following code:

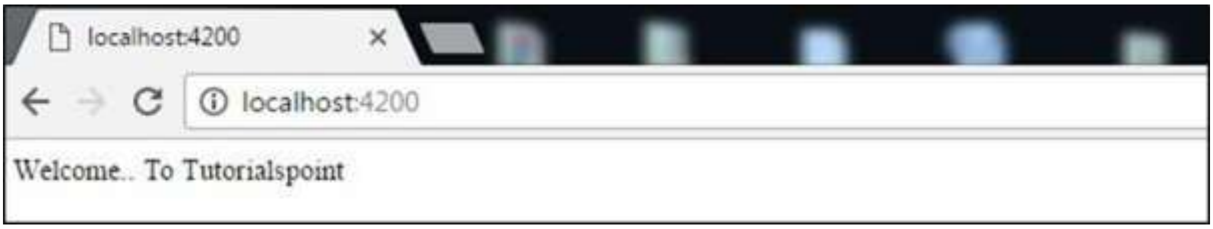
```
//passing the 'user' as parameter to a button
<button {{action "User" user}}>Click Here </button>
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Now you click on the button, the *User* action handler will be called with an argument containing the "user" model. This further displays the following result:



Welcome.. To Tutorialspoint

EmberJS – Template Action Specifying Type of Event

The alternative event can be specified on *{{action}}* helper by using the *on* option.

Syntax

```
<button {{action "action-name" on="event-name"}}>Click</button>
```

Example

The example given below specifies an alternative event on the *{{action}}* helper. Create a new route and name it as *post-action.js* with the following code:

```
import Ember from 'ember';
export default Ember.Component.extend({
   actions: {
      //toggling the text
      toggleBody: function () {
         this.toggleProperty('isShowing');
      }
   }
});
```

Open the *post-action.hbs* file created under *app/templates/* with the following code:

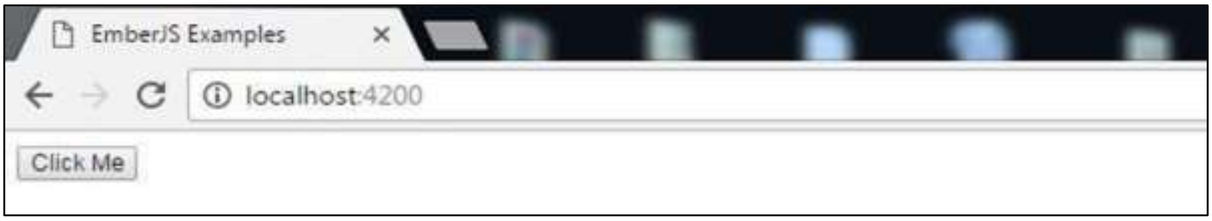
```
<button {{action "toggleBody" on='click'}}>{{title}}</button>
{{#if isShowing}}
<h2>Welcome to TutorialsPoint</h2>
{{/if}}
{{outlet}}
```

Next, open the *application.hbs* file created under *app/templates/* with the following code:

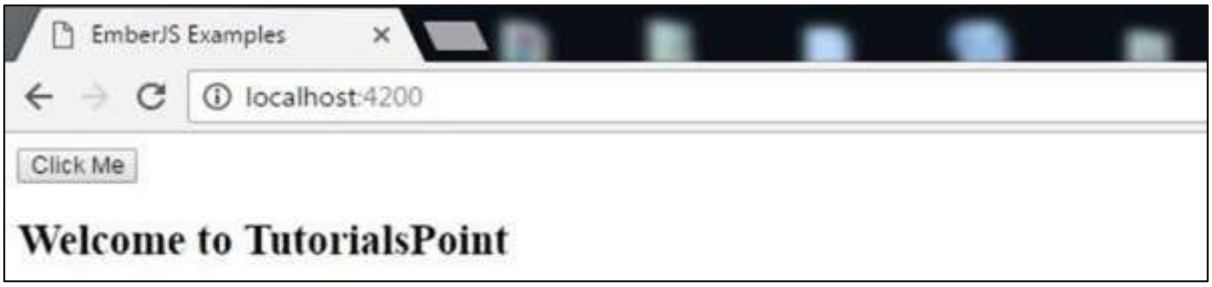
```
{{post-action title="Click Me"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Next, click on the button, the *{{action}}* helper triggers the action on the specified element and displays the following result:



Welcome to TutorialsPoint

EmberJS – Template Modifier Keys

You can allow modifier keys along with the *{{action}}* helper by using the *allowedKeys* option. Sometimes the *{{action}}* helper discards the click events with pressed modifier keys. Therefore, the *allowedKeys* option specifies, which keys should not be ignored.

Syntax

```
<button {{action 'action-name' allowedKeys="alt"}}></button>
```

Example

The example given below shows usage of *allowedKeys* option on the *{{action}}* helper. Create a new component and name it as *post-action.js* with the following code:

```
import Ember from 'ember';
export default Ember.Component.extend({
   actions: {
      //toggling the text
      toggleBody: function () {
         this.toggleProperty('isShowing');
      }
   }
});
```

Open the *post-action.hbs* file created under *app/templates/* with the following code:

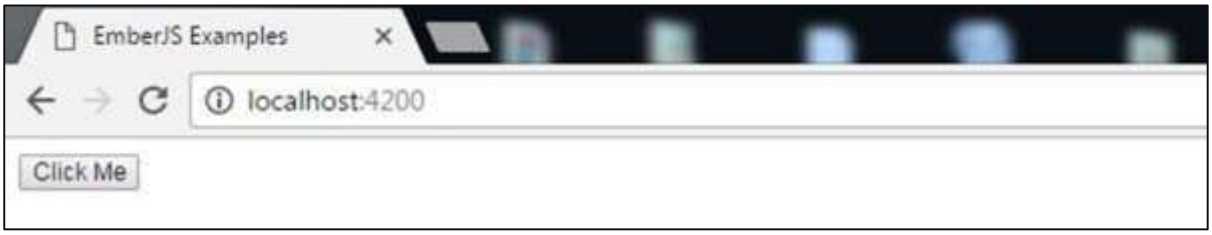
```
<button {{action "toggleBody" on='click' allowedKeys="alt"}}>{{title}}</button>
{{#if isShowing}}
<h2>Welcome to TutorialsPoint</h2>
{{/if}}
{{outlet}}
```

Now open the *application.hbs* file created under *app/templates/* with the following code:

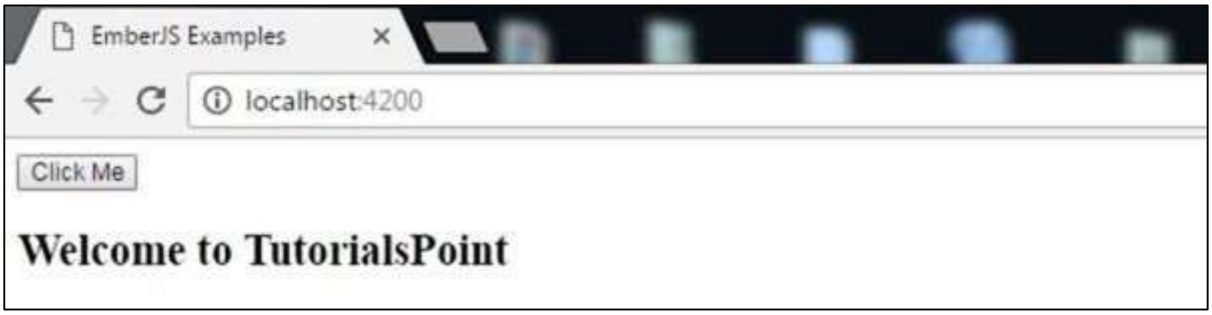
```
{{post-action title="Click Me"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Now you click on the button, the *{{action}}* helper triggers along with the *allowedKeys* option to allow the modifier keys:



Welcome to TutorialsPoint

EmberJS – Template Modifying Action's First Parameter

You can modify the action's first parameter by specifying a *value* option for the *{{action}}* helper.

Syntax

```
<input type="text" value={{name}} onblur={{action "action-name"}} />
```

Example

The example given below shows modifying the action's first parameter by using the *{{action}}* helper with *value* option. Create a new component and name it as *post-action.js* with the following code:

```
import Ember from 'ember';
export default Ember.Component.extend({
   actions: {
      actionFirstParameter(newName) {
         document.write('Name is:'+' '+newName);
      }
   }
});
```

Open the *post-action.hbs* file created under *app/templates/* with the following code:

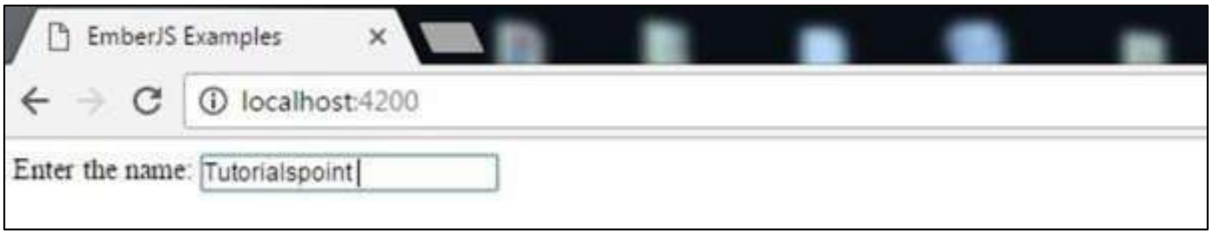
```
<label>Enter the name:</label>
<input type="text" value={{yourName}} onblur={{action "actionFirstParameter"
value="target.value"}} />
{{outlet}}
```

Next, open the *application.hbs* file created under *app/templates/* with the following code:

```
{{post-action}}
{{outlet}}
```

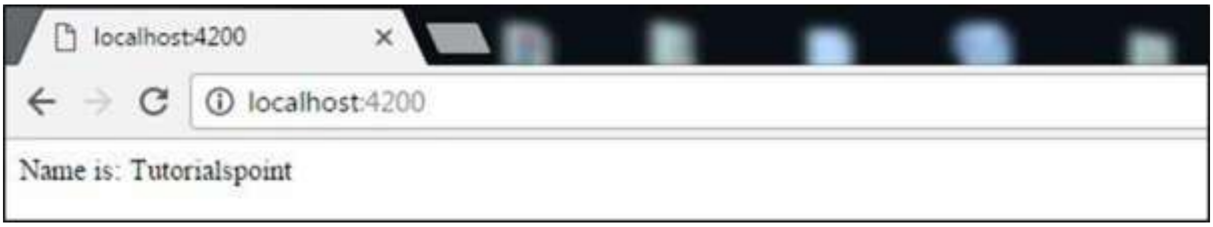
Output

Run the ember server and you will get the input box to enter the value:



Enter the name: Tutorialspoint

It will display the value of the input field, where the user has mentioned as shown in the screenshot below:



Name is: Tutorialspoint

EmberJS – Input Helpers

The common form controls can be created by using the *{{input}}* and *{{textarea}}* helpers in Ember.js. The *{{input}}* helper uses the built-in Ember.TextField, Ember.Checkbox classes and the *{{textarea}}* uses Ember.TextArea class.

The following table lists down the input helpers of Ember.js:

S.NO.	Helpers & Description
1	**Text Fields** It provides an input field which allows users to enter the data.
2	**CheckBoxes** It is a square box in which a user can toggle on and off, i.e., allows to select between one of two possible options.
3	**Text Areas** It is a multi-line text form field where a user can enter unlimited number of characters.

EmberJS – Template Input Helper Text Fields

Text field provides an input field, which allows users to enter the data. The following are the attributes, which can be used within the input helper:

'readonly'	'required'	'autofocus'
'value'	'placeholder'	'disabled'
'size'	'tabindex'	'maxlength'
'name'	'min'	'max'
'pattern'	'accept'	'autocomplete'
'autosave'	'formaction'	'formenctype'
'formmethod'	'formnovalidate'	'formtarget'
'height'	'inputmode'	'multiple'
'step'	'width'	'form'
'selectionDirection'	'spellcheck'	'type'

Syntax

```
{{input type="type-of-input" value="name-of-input-element"}}
```

Example

The example given below specifies the usage of textfields in the input helper. Create a route with name as *textfield* and open the *router.js* file to define the URL mappings:

```
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('textfield');
});

export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Input Helper Text Field</h2>
{{#link-to 'textfield'}}Click Here{{/link-to}}
{{outlet}}
```

When you click the link, page should open the *textfield.hbs* file, which contains the following code:

```
Enter Name : {{input type="text" placeholder="Enter the name" value=name}}
<button {{action "send"}}>Send</button>
{{outlet}}
```

Open the *textfield.js* file created under *app/routes/* with the following code:

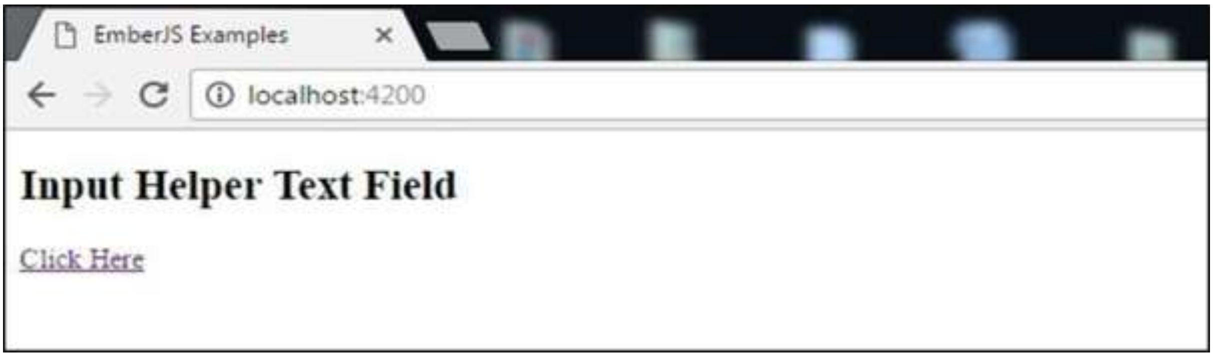
```
import Ember from 'ember';
export default Ember.Route.extend({
   model: function () {
      // initializing the variable 'name' as null by using create method
      return Ember.Object.create({
         name: null
      });
   }
});
```

Now open the *textfield.js* file created under *app/controllers/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   actions: {
      // this actions get the name from the text field
      send: function () {
         document.write('Name is: ' + this.get('name'));
      }
   }
});
```

Output

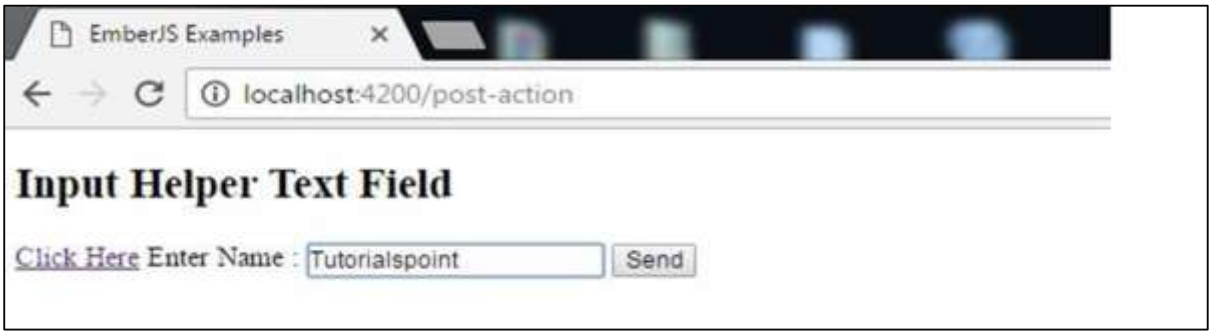
Run the ember server; you will receive the following output:



Input Helper Text Field

Click Here

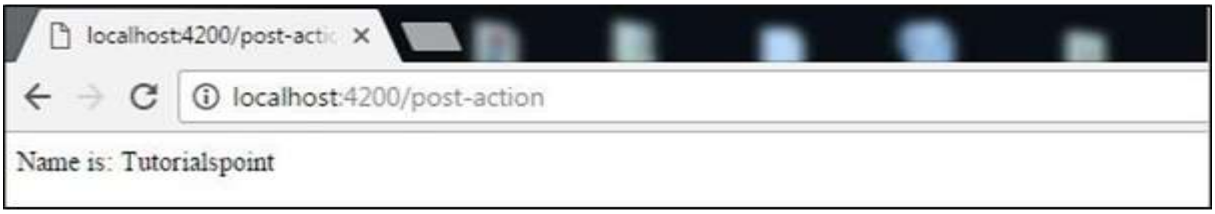
When you click on the link, an input field will get display, which allows users to enter the data:



Input Helper Text Field

Click Here Enter Name : Tutorialspoint Send

Now click on the send button, it will display the result as shown in the screenshot below:



Name is: Tutorialspoint

EmberJS Template Input Helper CheckBox

It is a square box in which a user can toggle on and off, i.e., it allows to select between one of two possible options. Checkboxes supports the following properties:

- checked
- disabled
- tabindex
- indeterminate
- name
- autofocus
- form

Syntax

```
{{input type="checkbox" name="NameOfCheckBox" checked=NameOfCheckBox}}
```

Example

The example given below specifies the usage of checkbox in the input helper. Create a route with name as *checkbox* and open the *router.js* file to define the URL mappings:

```
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('checkbox');
});
export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Input Helper CheckBox</h2>
{{#link-to 'checkbox'}}Click Here{{/link-to}}
{{outlet}}
```

When you click the link, page should open the *checkbox.hbs* file, which contains the below code:

```
{{input type="checkbox" checked=checkMe}} Check Box
<button {{action "send"}}>Click the checkbox</button>
{{outlet}}
```

Open the *checkbox.js* file created under *app/routes/* with the following code:

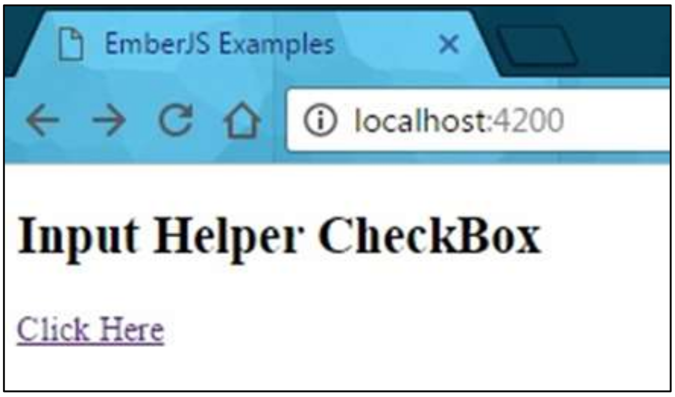
```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function () {
      return Ember.Object.create({
         checkMe: false
      });
   }
});
```

Now open the *checkbox.js* file created under *app/controllers/* with the following code:

```
import Ember from 'ember';
export default Ember.Controller.extend({
   actions: {
      send: function () {
         document.write('checkbox value: ' + this.get('checkMe'));
      }
   }
});
```

Output

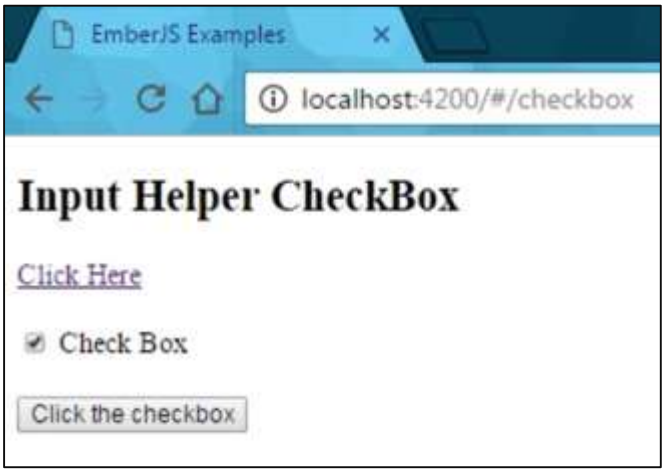
Run the ember server; you will receive the following output:



Input Helper CheckBox

Click Here

When you click on the link, a checkbox will get display and click on it:

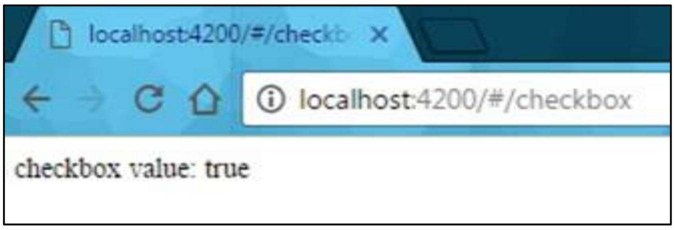


Input Helper CheckBox

Click Here

☑ Check Box

Next click the button, it will show the result as true as shown in the screenshot below:



```
checkbox value: true
```

EmberJS – Template Input Helper Text Areas

It is a multi-line text form field where a user can enter unlimited number of characters. The textarea binds the value of text to the current context.

The *{{textarea}}* supports the following properties:

- value
- name
- rows
- cols
- placeholder
- disabled
- maxlength
- tabindex
- selectionEnd
- selectionStart
- selectionDirection
- wrap
- readonly
- autofocus
- form
- spellcheck
- required

Syntax

```
{{textarea value=name cols="width_of_textarea" rows="number_of_lines"}}
```

Example

The example given below specifies multi-line text input control to enter unlimited number of characters. Create a route with name as *textarea* and open the *router.js* file to define the URL mappings:

```
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('textarea');
});

export default Router;
```

Open the file *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Input Helper Textarea</h2>
{{#link-to 'textarea'}}Click Here{{/link-to}}
{{outlet}}
```

When you click the link, page should open the *textarea.hbs* file, which contains the following code:

```
Enter text here: <br/><br/>{{textarea value=name cols="15" rows="5"
placeholder="Message"}}<br/>
<button {{action "send"}}>Send</button>
{{outlet}}
```

Open the *textarea.js* file created under *app/routes/* with the following code:

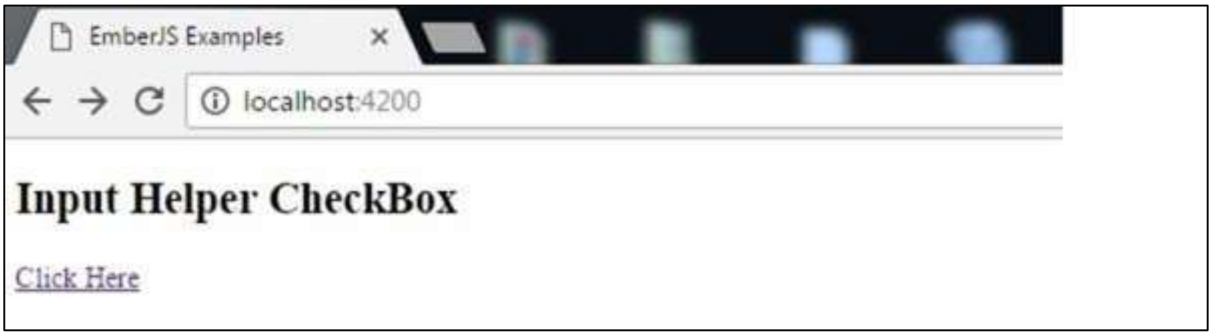
```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function () {
      return Ember.Object.create({
         name: null
      });
   }
});
```

Now open the *textarea.js* file created under *app/controllers/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
   actions: {
      send: function () {
            document.write('Entered text is: ' + this.get('name'));
        }
    }
});
```

Output

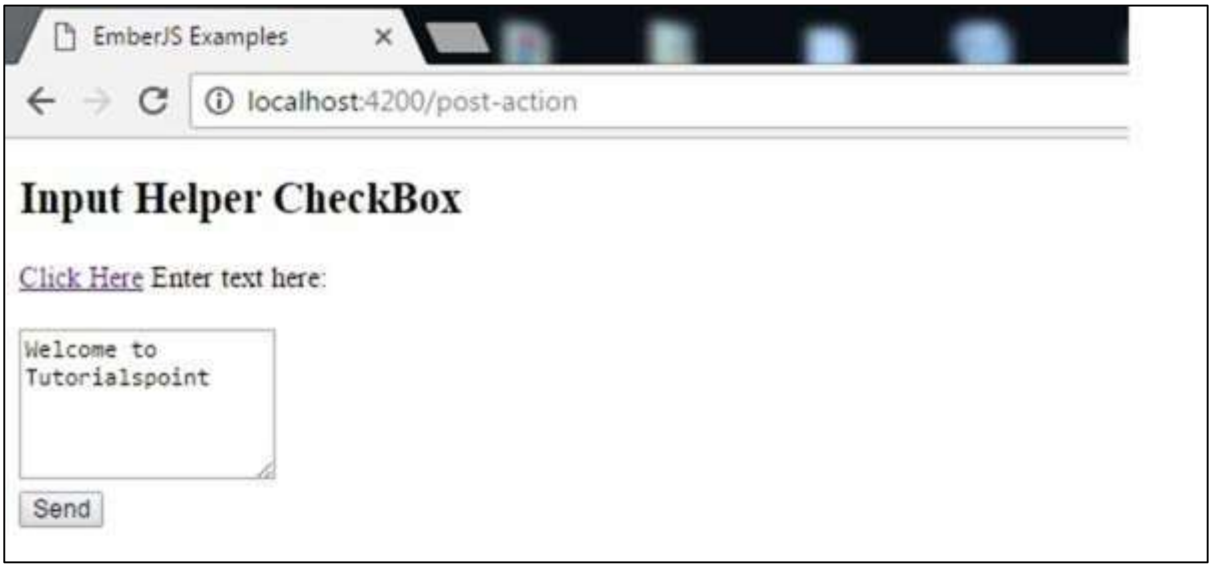
Run the ember server; you will receive the following output:



Input Helper CheckBox

Click Here

When you click on the link, a textarea will get display, enter the text and click on the send button:

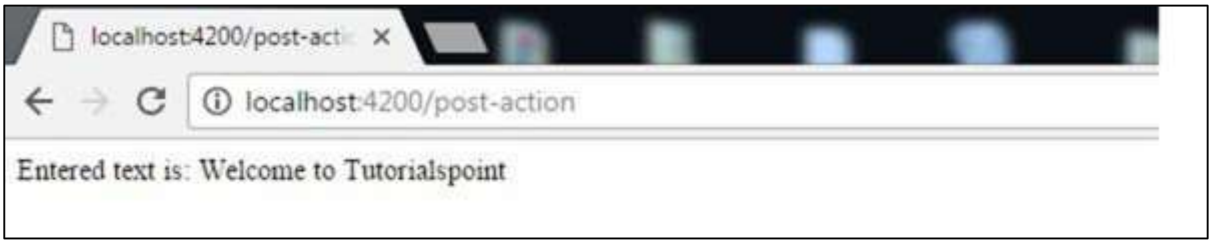


Input Helper CheckBox

Click Here Enter text here:

Welcome to Tutorialspoint

Now click the send button, it will display the result as shown in the screenshot below:



Entered text is: Welcome to Tutorialspoint

EmberJS – Development Helpers

The template developement can be made easier by using some helpers of Handlebars and Ember. The following table lists down the helpers used for developing the templates:

S.NO.	Helpers & Description
1	**Logging** It displays the output variables in the context of browser's console.
2	**Adding a Breakpoint** It halts an execution program to inspect the current rendering context.

EmberJS – Template Development Helper Log

You can display the output variables in the context of browser's console by using the *{{log}}* helper. With this helper, you can also receive the primitive types such as strings or numbers.

Syntax

```
{{log 'Statment' VarName}}
```

Example

The example given below shows how to render an output to the browser's console. Create a component with the name *post-action* and add the following code to it:

```
import Ember from 'ember';


export default Ember.Component.extend({
 actions: {
    send() {
      this.toggleProperty('isShowingBody');
    }
  }
});
```

Now open the *post-action.hbs* file created under *app/templates/components/* with the following code:

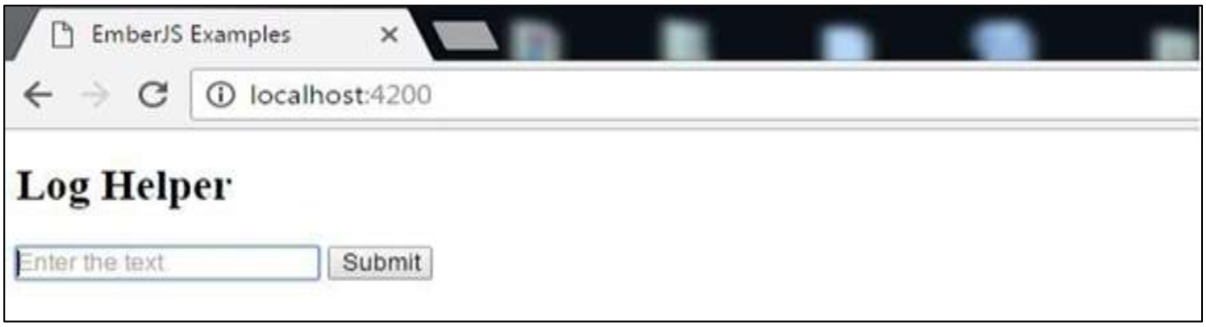
```
<h2>Log Helper</h2>
{{#if isShowingBody}}
{{log 'Name is:' firstName}}
{{/if}}


{{input type="text" placeholder="Enter the text" value=firstName
disabled=entryNotAllowed}}
<button {{action "send"}}>Submit</button>
{{yield}}
```

Open the *index.hbs* file, which is created under *app/templates/* with the below code:

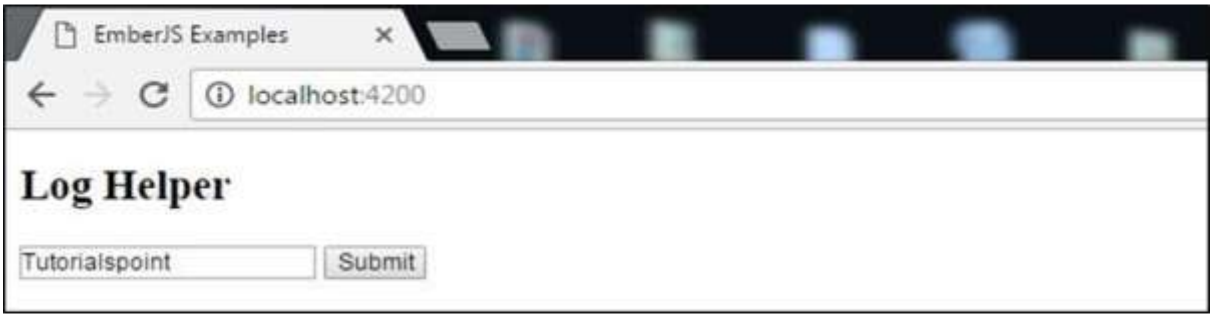
```
{{post-action}}
{{outlet}}
```

Output

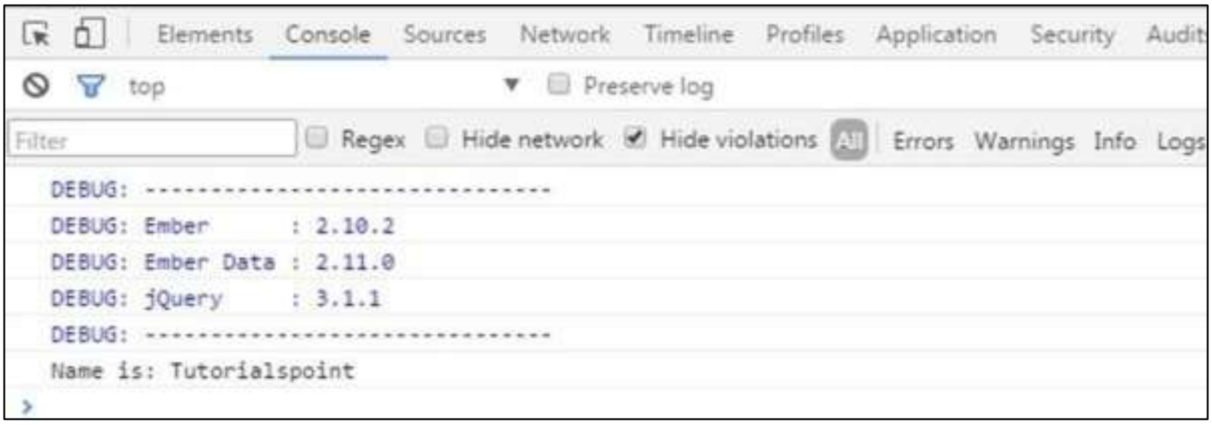
Run the ember server; you will receive the following output:



Next, enter the text in the input box and click on the submit button:



Next, it will display the result in the browser's console as shown in the screenshot below:



```
DEBUG: -----------------------------
DEBUG: Ember      : 2.10.2
DEBUG: Ember Data : 2.11.0
DEBUG: jQuery     : 3.1.1
DEBUG: -----------------------------
Name is: Tutorialspoint
```

EmberJS – Template Development Adding a Breakpoint

You can halt the execution program inside the debugger helper by using the *{{debugger}}* helper, which allows inspecting the current rendering context.

Syntax

```
{{debugger}}
```

The above helper helps to access the values of the template by using the *get* function.

EmberJS – Writing Helpers

You can add extra functionality to the templates and convert the raw values from models and components into proper format for the users. If you are using the HTML application multiple times, then you can add a custom helper from any *Handlebars template*. If the current context changes, then Ember.js will automatically execute the helper and updates the DOM with the updated value.

Syntax

```
export function Helper_Name([values]) {
   // code here
}


export default Ember.Helper.helper(Helper_Name);
```

The following table lists down the different ways of using helper names:

S.NO.	Helper Names & Description
1	**Helper Arguments** You can pass more than one argument to the helper by specifying after the helper name.
2	**Named Arguments** You can pass the named arguments along with the related value.
3	**Escaping HTML Content** It is used to escape the HTML tags while displaying the result.

Example

The example given below implement the helper, which takes more than one input and returns single output. Create a new helper with the following command:

```
ember generate helper helper-name
```

In this example, we have created the helper with the name *writinghelper*. Now open the *writinghelper.js* file which is created under *app/helpers/*.

```
import Ember from 'ember';


export function formatHelper([value])  {
  let var1 = Math.floor(value * 100);
  let cents = value % 100;
  let var3 = '$';


  if (cents.toString().length === 1)
  return `${var3}${var1}`;
}


export default Ember.Helper.helper(formatHelper);
```

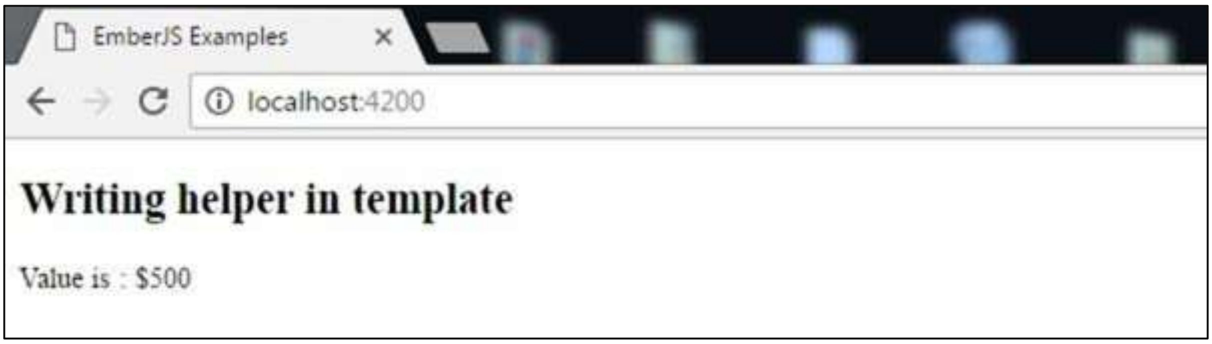
You can use the "writinghelper" helper in the template within curly braces. Open the *index.hbs* file and write the following code:

```
Value is : {{writinghelper 5}}
{{outlet}}
```

In the above code, we have passed the helper value in the template, which displays the count of cents into formatted string.

Output

Run the ember server; you will receive the following output:



Writing helper in template

Value is : $500

EmberJS – Helper Arguments

You can pass more than one argument to the helper by specifying after the helper name.

Syntax

```
export default Ember.Helper.helper(function(params) {
   //code here
}
```

Example

The example given below passes the multiple arguments to the helper. Create a new helper as *helperarguments* and add the following code to it:

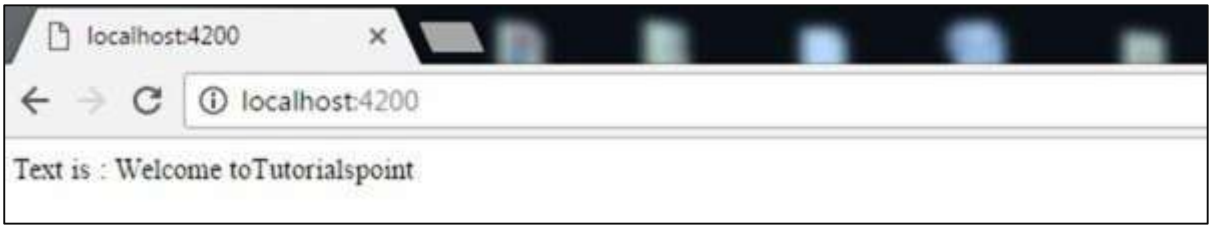
```
import Ember from 'ember';

export default Ember.Helper.helper(function(params) {
  let [arg1, arg2] = params;
  document.write("Text is : " +arg1+ ''+arg2);
});
```

Open the *index.hbs* file and write the following code:

```
{{helperarguments "Welcome to" "Tutorialspoint"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Text is : Welcome toTutorialspoint

EmberJS - Named Arguments

You can pass the named arguments along with the related value.

Syntax

```
export default Ember.Helper.helper(function([value],namedArgs) {
   // code here
});
```

Example

The example given below passes named arguments along with the related value to the helper. Create two helpers as *namedarguments* and *namedarguments1*. Open the *namedarguments.js* file, which is created under *app/helpers/* and add the following code to it:

```
import Ember from 'ember';

export default Ember.Helper.helper(function([value], args) {
  let var1 = Math.floor(value * 100);
  let var2 = value % 100;
  let var3 = args.var3 === undefined ? '$' : args.var3;

  if (var2.toString().length === 1)
  return `${var3}${var1}`;
});
```

Open another helper file *namedarguments1.js* file and add the following code:

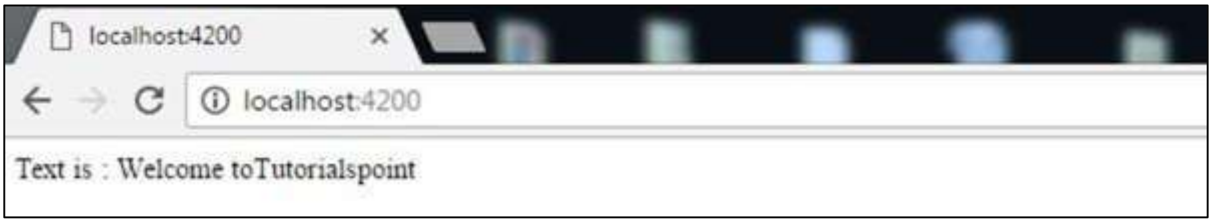
```
import Ember from 'ember';

export default Ember.Helper.helper(function(params, args) {
  document.write("Text is : " +args.option1+ ''+args.option2);
});
```

Open the *index.hbs* file and write the following code:

```
{{namedarguments1 option1="Welcome to" option2="Tutorialspoint"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Text is : Welcome toTutorialspoint

EmberJS – Escaping HTML Content

You can escape the HTML tags while displaying the result.

Syntax

```
export default Ember.Helper.helper(function(params) {
   //code here
}
```

Example

The example given below will escape the HTML tags. Create a new helper *eschtmlcontent* and add the following code to it:

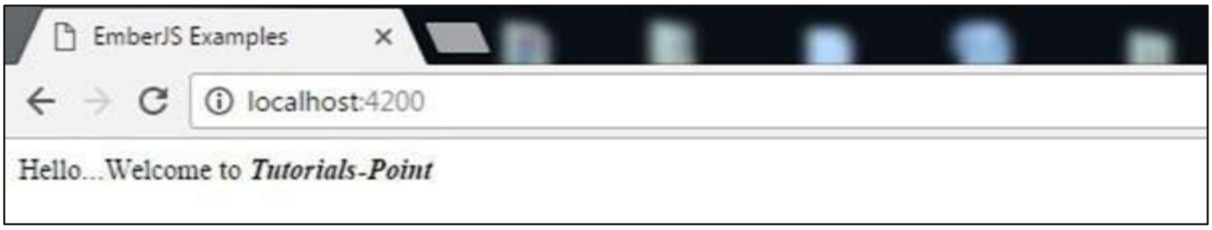
```
import Ember from 'ember';

export default Ember.Helper.helper(function(param) {
  return Ember.String.htmlSafe(`<i><b>${param}</b></i>`);
});
```

Open the *index.hbs* file and write the following code:

```
Hello...Welcome to {{eschtmlcontent "Tutorials-Point"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Hello...Welcome to ***Tutorials-Point***

8. EmberJS – Components

The Ember.js components uses the W3C web component specification and provides true encapsulation UI widgets. It contains the three main specification as *templates*, *shadow DOM* and *custom elements*. The component is declared within the data-template-name which has a path name instead of a plain string and are prefixed with "components/".

The following table lists down the action events of actions:

S.NO.	Action Events & Description
1	**Defining a Component** You can easily define a component in Ember.js and each component must have a dash in their name.
2	**Component Lifecycle** Component lifecycle uses some of the methods in order to execute the code at specific times in a component's life.
3	**Passing Properties to a Component** The component doesn't access the property directly in the template scope. Therefore, just declare the property at the time of component deceleration.
4	**Wrapping Content in a Component** You can wrap the content in a component by using the templates.
5	**Customizing a Component's Element** You can customize the component's element such as attributes, class names by using a subclass of *Ember.Component* in the JavaScript.
6	**Using Block Params** The passed properties in a component can give back the result in a block expression.
7	**Handling Events** The user events such as double-click, hovering, key press etc can be handled by event handlers. To do this, apply the event name as a method on the component.
8	**Triggering Changes with Actions** Components can trigger the changes and communicate with events by using the actions.

EmberJS – Defining a Component

You can easily define the component in Ember.js and each component must have a dash in their name (ex: my-component). Ember.js has the power of defining the component subclasses by using an *Ember.Component* class.

The component can be created by using the below command:

```
ember generate component component-name
```

Example

The example given below describe how to define a component in Ember.js. Create a component with the name *post-action*, which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
      toggleBody:['Welcome to Tutorialspoint!!!']
});
```

Now open the component template file *post-action.hbs* with the following code:

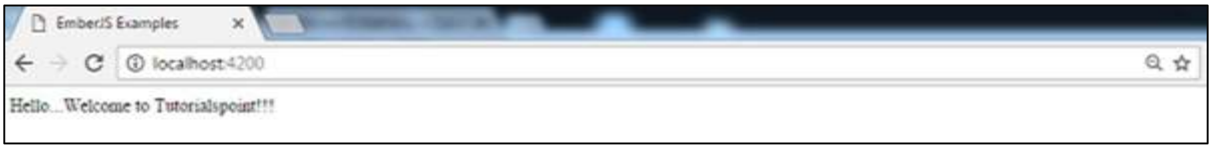
```
{{#each toggleBody as |body|}}
Hello...{{body}}
{{/each}}
{{yield}}
```

Open the *index.hbs* file and add the following code:

```
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Hello...Welcome to Tutorialspoint!!!

EmberJS – Defining a Component

The component's lifecycle contains three methods which execute according to the render scenario.

On Initial Render

- init
- didReceiveAttrs
- willRender
- didInsertElement
- didRender

On Re-Render

- didUpdateAttrs
- didReceiveAttrs
- willUpdate
- willRender
- didUpdate
- didRender

On Component Destroy

- willDestroyElement
- willClearRender
- didDestroyElement

The following table lists down the different ways of using lifecycle hooks within components:

S.NO.	Lifecycle Ways & Description
1	**Attribute Change with didUpdateAttrs** The *didUpdateAttrs* hook can be used when the component's attributes have changed and called before re-rendering the component.
2	**Attributes with didReceiveAttrs** The *didReceiveAttrs* hook can be used after the *init* method and called when the component's attributes are updated.
3	**Third-Party Libraries with didInsertElement** You can initialize and attach the 3rd party libraries into the DOM element by using this hook.

4	**Rendered DOM with didRender** The *didRender* hook is called to make update to the DOM when the template has rendered.
5	**Detaching and Tearing Down with willDestroyElement** You can remove the component elements from the DOM by triggering the *willDestroyElement* hook.

EmberJS – Attribute Change with didUpdateAttrs

The *didUpdateAttrs* hook can be used when component's attributes have changed and called before re-rendering the component.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
...
didUpdateAttrs() {
   // code here
},
...
})
```

Example

The example given below describes the use of *didUpdateAttrs* hook to be used when the component's attributes have changed. Create a component with the name *post-action* which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
      didInitAttrs(options) {
    console.log('didInitAttrs', options);
  },


  didUpdateAttrs(options) {
    console.log('didUpdateAttrs', options);
  },

  willUpdate(options) {
    console.log('willUpdate', options);
  },

  didReceiveAttrs(options) {
    console.log('didReceiveAttrs', options);
  },

  willRender() {
    console.log('willRender');
  },

  didRender() {
    console.log('didRender');
  },

  didInsertElement() {
    console.log('didInsertElement');
  },

  didUpdate(options) {
    console.log('didUpdate', options);
  },
});
```

Now open the component template file *post-action.hbs* with the following code:

```
<p>name: {{name}}</p>
<p>attrs.data.a: {{attrs.data.a}} is in console</p>
{{yield}}
```

Open the *index.hbs* file and add the following code:

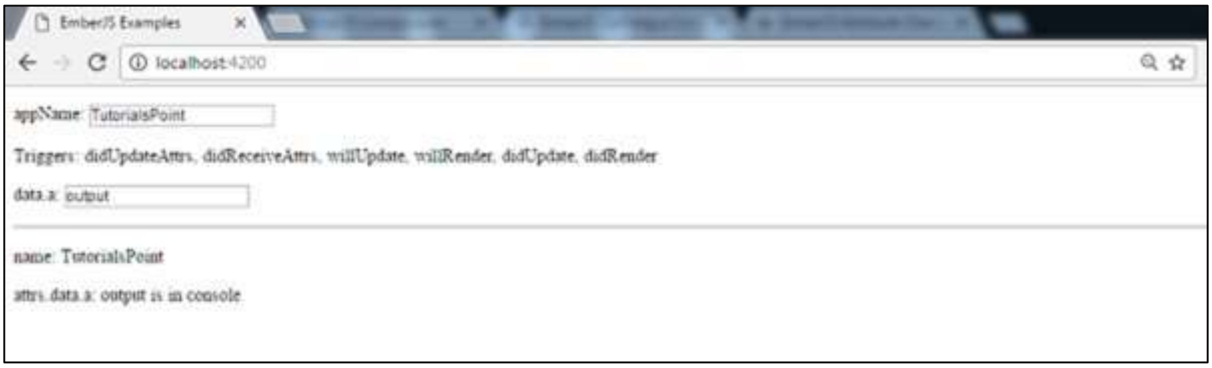
```
<div>
  <p>appName: {{input type="text" value=appName}}</p>
  <p>Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender,
didUpdate, didRender</p>
</div>

<div>
  <p>data.a: {{input type="text" value=data.a}}</p>
</div>
<hr>

{{post-action name=appName data=data}}
{{outlet}}
```

Create an *index.js* file under *app/controller/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  appName:'TutorialsPoint',
  data: {
    a: 'output',
  }
});
```

Output

Run the ember server; you will receive the following output:



appName: TutorialsPoint

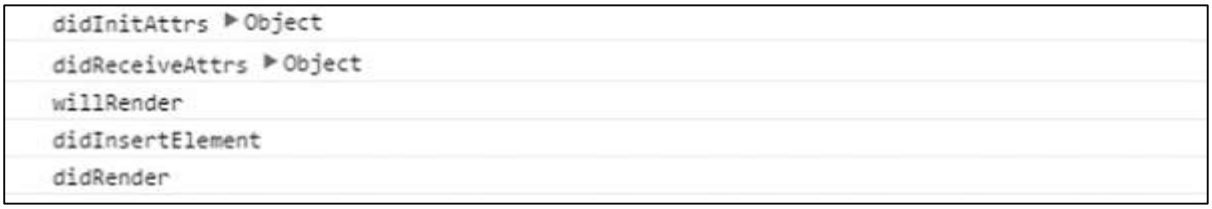
Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender, didUpdate, didRender

data.a: output

name: TutorialsPoint

attrs.data.a: output is in console

Open the console and you will get the output as shown in the screenshot below:



```
didInitAttrs ▶Object
didReceiveAttrs ▶Object
willRender
didInsertElement
didRender
```

EmberJS – Attributes with didReceiveAttrs

The *didReceiveAttrs* hook can be used after the *init* method and called when the component's attributes are updated and it will not run when the re-rendered is initiated internally.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
...
didReceiveAttrs() {
   // code here
},
...
})
```

Example

The example given below describes the use of *didReceiveAttrs* hook to be used when the component's attributes are updated. Create a component with the name *post-action* which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
      didInitAttrs(options) {
    console.log('didInitAttrs', options);
  },


  didUpdateAttrs(options) {
    console.log('didUpdateAttrs', options);
  },

  willUpdate(options) {
    console.log('willUpdate', options);
  },

  didReceiveAttrs(options) {
    console.log('didReceiveAttrs', options);
  },

  willRender() {
    console.log('willRender');
  },

  didRender() {
    console.log('didRender');
  },

  didInsertElement() {
    console.log('didInsertElement');
  },

  didUpdate(options) {
    console.log('didUpdate', options);
  },
});
```

Now open the component template file *post-action.hbs* with the following code:

```
<p>name: {{name}}</p>
<p>attrs.data.a: {{attrs.data.a}} is in console</p>
{{yield}}
```

Open the *index.hbs* file and add the following code:

```
<div>
  <p>appName: {{input type="text" value=appName}}</p>
  <p>Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender,
didUpdate, didRender</p>
```

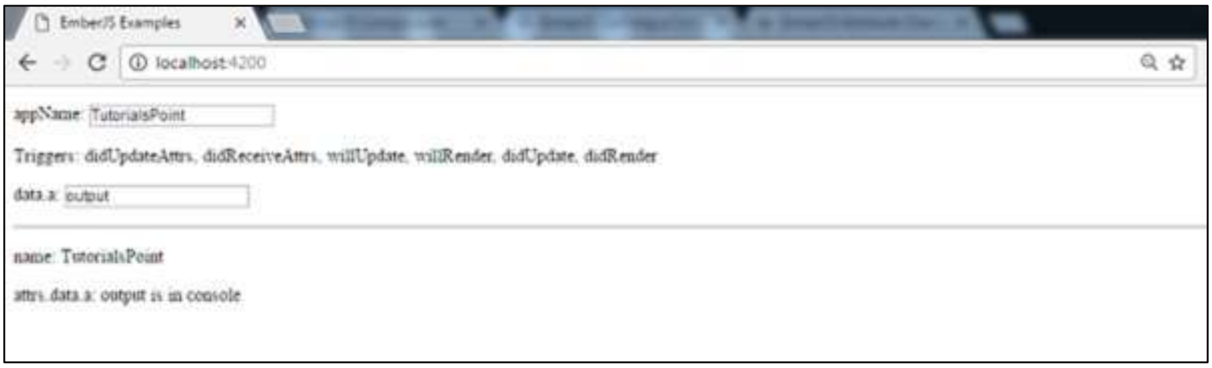
```
</div>

<div>
  <p>data.a: {{input type="text" value=data.a}}</p>
</div>
<hr>

{{post-action name=appName data=data}}
{{outlet}}
```

Create an *index.js* file under *app/controller/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  appName:'TutorialsPoint',
  data: {
    a: 'output',
  }
});
```

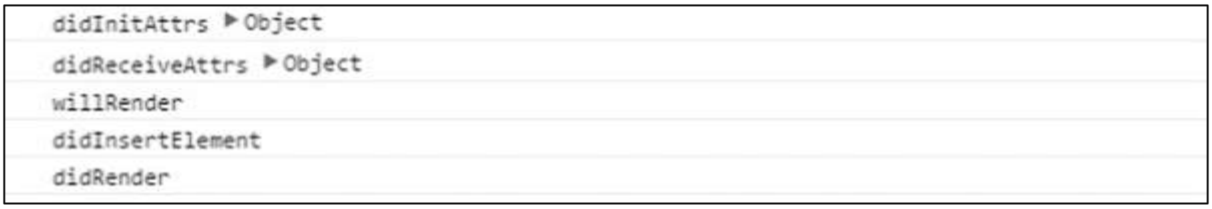
Output

Run the ember server; you will receive the following output:



```
appName: TutorialsPoint
Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender, didUpdate, didRender
data.a: output

name: TutorialsPoint
attrs.data.a: output is in console
```

Open the console and you will get the output as shown in the screenshot below:



```
didInitAttrs ▶Object
didReceiveAttrs ▶Object
willRender
didInsertElement
didRender
```

EmberJS – Third-Party Libraries with didInsertElement

You can initialize and attach the 3rd party libraries into the DOM element by using this *didInsertElement* hook. This can be called when the component's element has been created and inserted into DOM and accessible by using the *s()* method.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
...
didInsertElement() {
   // code here
},
...
})
```

Example

The example given below describes the use of *didInsertElement* hook when integrating with third-party library. Create a component with the name *post-action*, which will get define under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';
var inject = Ember.inject;


export default Ember.Component.extend({
   age: 'Tutorialspoint',
   actions: {
      pressed: function () {
         this.$("#test").fadeIn("slow");
      }
   },
   didInsertElement: function () {
      Ember.run.scheduleOnce('afterRender', this, function () {
         this.$("#test").fadeOut("slow");
      });
   }
});
```

Now open the component template file *post-action.hbs* with the following code:

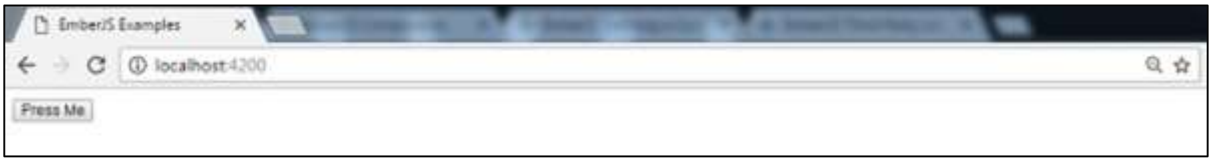
```
<div id="test">This is {{age}}</div>
<button {{action "pressed"}}>
Press Me
</button>
{{yield}}
```

Open the *index.hbs* file and add the following code:

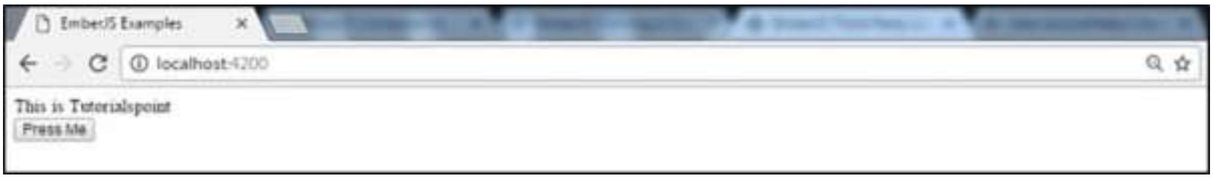
```
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



When you click the button, it will specify the fadeIn and fadeOut effect on the text:



This is Tutorialspoint

EmberJS – Rendered DOM with didRender

The *didRender* hook is called to make update to the DOM when the template has rendered.

Syntax

```
import Ember from 'ember';

export default Ember.Component.extend({
...
didRender() {
   // code here
},
...
})
```

Example

The example given below describes the use of *didRender* hook which makes update to the DOM. Create a component with the name *post-action* which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
      didInitAttrs(options) {
    console.log('didInitAttrs', options);
  },

  didUpdateAttrs(options) {
    console.log('didUpdateAttrs', options);
  },

  willUpdate(options) {
    console.log('willUpdate', options);
  },

  didReceiveAttrs(options) {
    console.log('didReceiveAttrs', options);
  },

  willRender() {
    console.log('willRender');
  },

  didRender() {
    console.log('didRender');
  },

  didInsertElement() {
    console.log('didInsertElement');
  },

  didUpdate(options) {
    console.log('didUpdate', options);
  },
});
```

Now open the component template file *post-action.hbs* with the following code:

```
<p>name: {{name}}</p>
<p>attrs.data.a: {{attrs.data.a}} is in console</p>
{{yield}}
```

Open the *index.hbs* file and add the following code:

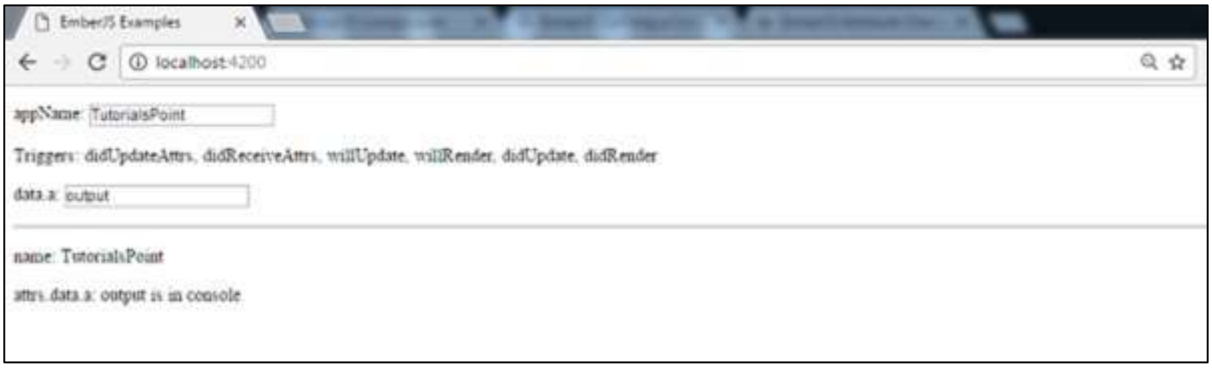
```
<div>
  <p>appName: {{input type="text" value=appName}}</p>
  <p>Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender,
didUpdate, didRender</p>
</div>

<div>
  <p>data.a: {{input type="text" value=data.a}}</p>
</div>
<hr>

{{post-action name=appName data=data}}
{{outlet}}
```

Create an *index.js* file under *app/controller/* with the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  appName:'TutorialsPoint',
  data: {
    a: 'output',
  }
});
```

Output

Run the ember server; you will receive the following output:



appName: TutorialsPoint

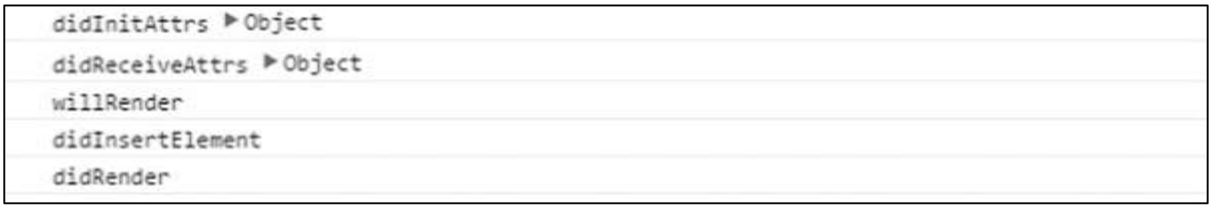
Triggers: didUpdateAttrs, didReceiveAttrs, willUpdate, willRender, didUpdate, didRender

data.a: output

name: TutorialsPoint

attrs.data.a: output is in console

Open the console and you will get the output as shown in the screenshot below:



```
didInitAttrs ▶Object
didReceiveAttrs ▶Object
willRender
didInsertElement
didRender
```

EmberJS – Detaching and Tearing Down with willDestroyElement

You can remove the component elements from the DOM by triggering the *willDestroyElement* hook.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
...
willDestroyElement() {
   // code here
},
...
})
```

Example

The example given below describes the use of *willDestroyElement* hook, which removes the component elements from the DOM. Create a controller with name *index* and open the file from *app/controller/* to add the following code:

```
import Ember from 'ember';


export default Ember.Controller.extend({
  showComponent: true,
  laterCount: 0,


  buttonText: Ember.computed('showComponent', function() {
    let showing = Ember.get(this, 'showComponent');
    if (showing) {
      return 'Remove';
    } else {
      return 'Add';
    }
  }),


  actions: {
    toggleComponent() {
      this.toggleProperty('showComponent');
    },


    updateLaterCount() {
      Ember.set(this, 'laterCount', Ember.get(this, 'laterCount') + 1);
    }
  }
});
```

Create a component with the name *post-action*, which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
     runLater: null,
  didInsertElement() {
    let timeout = Ember.run.later(this, function() {
      Ember.Logger.log('fired this after 1 seconds...');
      this.sendAction();
    }, 500);

    Ember.set(this, 'runLater', timeout);
  },

  willDestroyElement() {
    Ember.run.cancel(Ember.get(this, 'runLater'));
  }
});
```

Now open the component template file *post-action.hbs* with the following code:

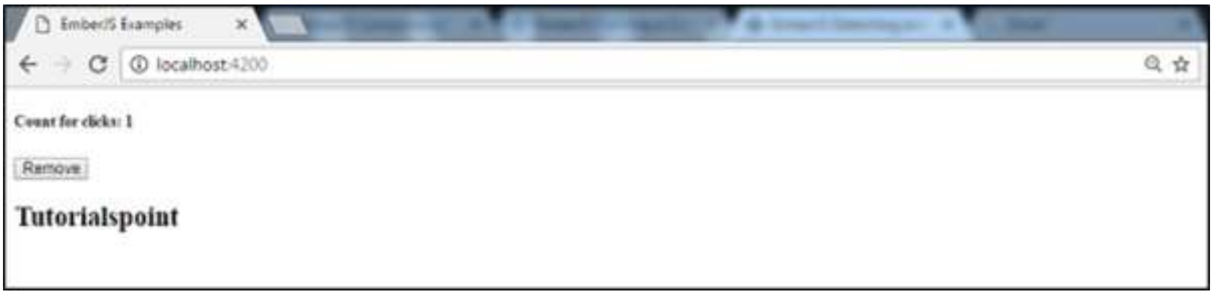
```
<h2>Tutorialspoint</h2>
```

Open the *index.hbs* file and add the following code:

```
<h5>Count for clicks: {{laterCount}}</h5>
<button {{action 'toggleComponent'}}>{{buttonText}}</button>
{{#if showComponent}}
{{post-action action="updateLaterCount"}}
{{/if}}
{{outlet}}
```

Output

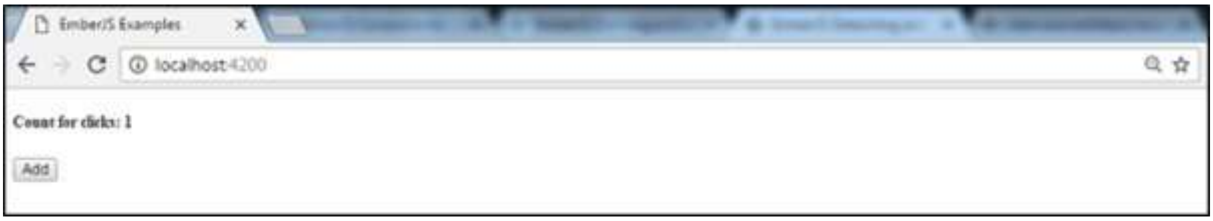
Run the ember server; you will receive the following output:



Count for clicks: 1

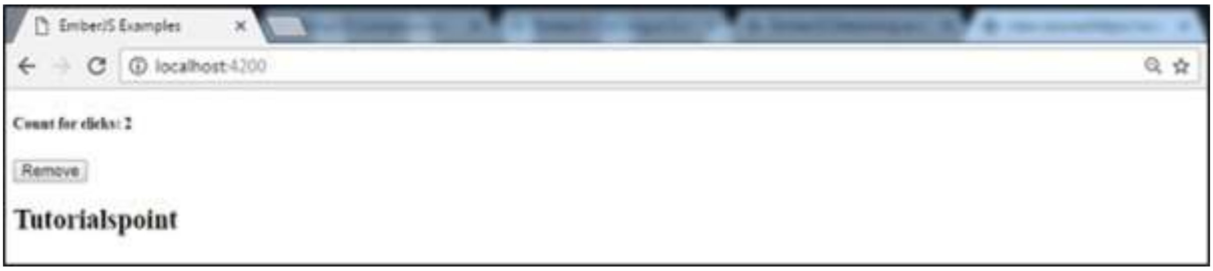
Tutorialspoint

Initially number of clicks will be 1. When you click the *Remove* button, it will remove the text:



Count for clicks: 1

Next, click the *Add* button, it will increment the number of clicks and display the text:



Count for clicks: 2

Tutorialspoint

EmberJS – Passing Properties to a Component

The component doesn't access the property directly in the template scope. Therefore, just declare the property at the time of component deceleration (ex: {{component-name title=title}}). The *title* property in the outer template scope is available inside the component's template.

Syntax

```
{{post-action title=title}}
```

In the above code, the 'post-action' component has the 'title' property and initialized with the same name as of property name ('title').

Example

The example given below describes how to pass properties to a component.Create a route with the name as *post-action* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
    this.route('post-action');
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Now open the component template file *post-action.hbs* with the following code:

```
<p>Enter your data: {{input type="text" value=title}}</p>
<p>The details of the object passed are : {{title}}</p>
{{yield}}
```

Open the *index.hbs* file and add the following code:

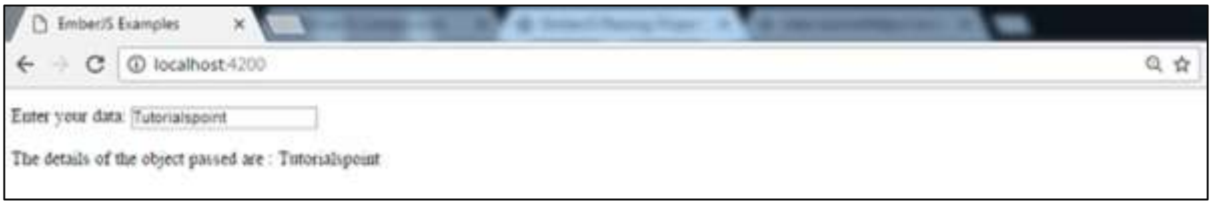
```
{{post-action title=title}}
{{outlet}}
```

Open the file *post-action.js* file created under *app/routes/* with the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function() {
      // assigning the default value for 'title' property
      return {
         title: ""
      };
   }
});
```

Output

Run the ember server; you will receive the following output:



Enter your data: Tutorialspoint

The details of the object passed are : Tutorialspoint

EmberJS – Wrapping Content in a Component

You can wrap the content in a component by using the templates. Consider we have one component called {{my-component}}, which can be wrapped in component by passing properties to it in another template as shown below:

```
{{my-component title=title action="funcName"}}
```

You can share the component data with its wrapped content. For more information, click this link - https://www.tutorialspoint.com/emberjs/comp_share_wrap_cont_comp.htm.

Example

The example given below specifies how to wrap content in a component. Create a component with the name *post-action*, which will get define under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
   actions: {
      compFunc: function () {
         this.set('title', "Tutorialspoint...");
         // This method sends the specified action
         this.sendAction();
      }
   }
});
```

Now open the component template file *post-action.hbs* with the following code:

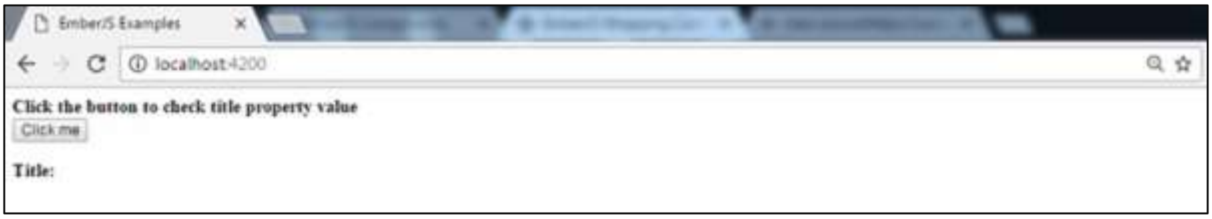
```
<input type="button" value="Click me" {{action "compFunc"}} /><br/>
// wrapping the 'title' property value
<p><b>Title:</b> {{title}}</p>
{{yield}}
```

Open the *index.hbs* file and add the following code:

```
<b>Click the button to check title property value</b>
{{post-action title=title action="compFunc"}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:

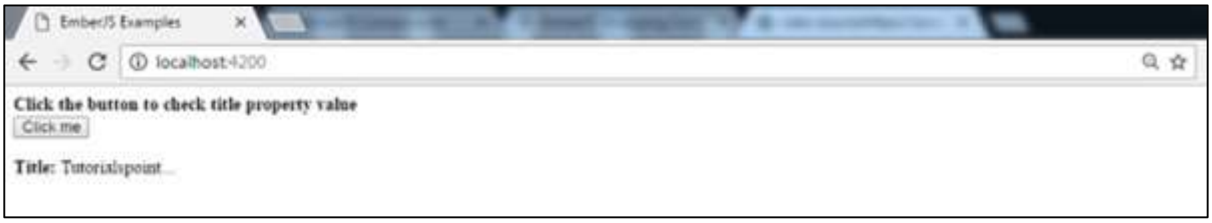


Click the button to check title property value

Click me

Title:

When you click on button, the *compFunc()* function will get trigger and it will further display the following output:



Click the button to check title property value

Click me

Title: Tutorialspoint...

EmberJS – Customizing a Component's Element

You can customize the component's element such as attributes, class names by using subclass of *Ember.Component* in the JavaScript.

The following table lists down the different types of customizing a component's element:

S.NO.	Component Element & Description
1	**Customizing Element** Customize the element by using *Ember.Component* subclass and set the *tagName* property to it.
2	**Customizing Element's Class** Customize the element's class at the invocation time i.e. while calling the clas name.
3	**Customizing Attributes** You can customize the attributes by binding them to a DOM element by using the *attributeBindings* property.

EmberJS – Customizing a Component's Element

Customize the element by using *Ember.Component* subclass and set the *tagName* property to it.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
  tagName: 'tag_name'
});
```

Example

The example given below specifies customizing a component's element by using *tagName* property. Create a component with the name *post-action*, which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
   tagName: 'h1'
});
```

Now open the component template file *post-action.hbs* with the following line of code:

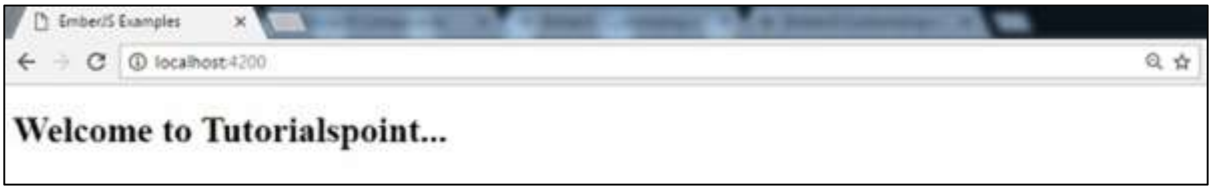
```
Welcome to Tutorialspoint...
{{yield}}
```

Open the *index.hbs* file and add the following line of code:

```
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Welcome to Tutorialspoint...

EmberJS – Customizing the Element's Class

Customize the element's class at the invocation time, i.e., at the time of calling the class name.

Syntax

```
import Ember from 'ember';
export default Ember.Component.extend({
  classNames: ['name_of_class']
});
```

Example

The example given below specifies customizing the element's class at the invocation time. Create a component with the name *post-action*, which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';
export default Ember.Component.extend({
  classNameBindings: ['isUrgent'],
  isUrgent: true,
});
```

Now open the component template file *post-action.hbs* with the following line of code:

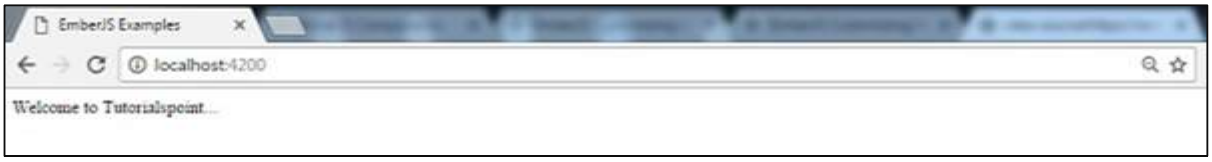
```
<div class="ember-view is-urgent">Welcome to Tutorialspoint...</div>
{{yield}}
```

Open the *index.hbs* file and add the following line of code:

```
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Welcome to Tutorialspoint...

EmberJS – Customizing Attributes

You can customize the attributes by binding them to a DOM element by using the *attributeBindings* property.

Syntax

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'tag_name',
  attributeBindings: ['attr_name'],
  attr_name: 'value'
});
```

Example

The example given below specifies customizing the attributes by binding them to a DOM element by using the *attributeBindings* property. Create a component with the name *post-action*, which will get defined under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'font',
  attributeBindings: ['color'],
  color: "red"
});
```

Now open the component template file *post-action.hbs* with the following line of code:

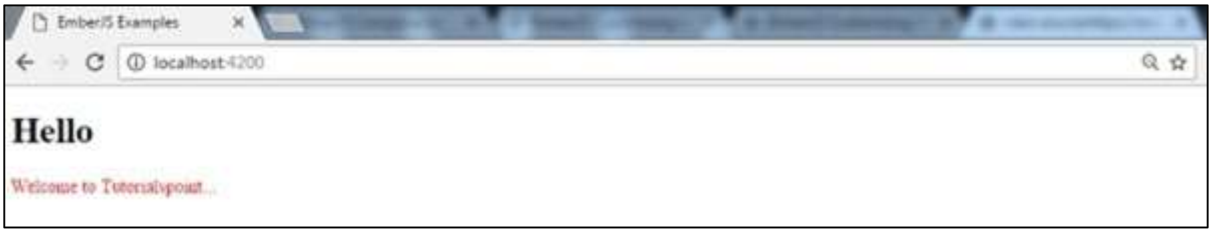
```
<div>Welcome to Tutorialspoint...</div>
{{yield}}
```

Open the *index.hbs* file and add the following line of code:

```
<h1>Hello</h1>
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Hello

Welcome to Tutorialspoint...

EmberJS – Using Block Params

The passed properties in a component can give back the result in a block expression.

The following table lists down the different ways of using block params:

S.NO.	BlockParam Ways & Description
1	**Return values from a component with yield** The values can be returned from a component by using the *yield* option.
2	**Supporting both block and non-block component usage** You can support the usage of block and non block components from single component by using the *hasBlock* property.

EmberJS – Return values from a component with yield

The values can be returned from a component by using the *yield* option.

Syntax

```
{#each myval as |myval1|}}
   {{ yield myval1 }}
{{/each}}
```

Example

The example given below specifies returning values from a component with the *yield* property. Create a route with the name *comp-yield* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config
// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('comp-yield');
});

export default Router;
```

Create the *application.hbs* file and add the following code:

```
// link-to is a handlebar helper used for creating links
{{#link-to 'comp-yield'}}Click Here{{/link-to}}
{{outlet}} //It is a general helper, where content from other pages will appear
inside this section
```

Open the *comp-yield.js* file which is created under *app/routes/*and enter the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function() {
      // an array called 'country' contains objects
      return { country: ['India', 'England', 'Australia'] };
   }
});
```

Create a component with the name *comp-yield* and open the component template file *comp-yield.hbs* created under *app/templates/* with the following code:

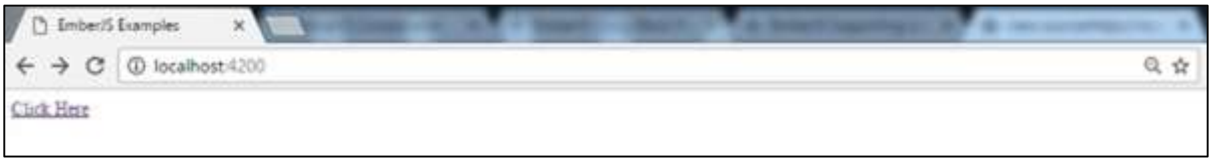
```
{{#comp-yield country=model.country as |myval|}}
   <h3>{{ myval }}</h3>
{{/comp-yield}}
{{outlet}}
```

Open the *comp-yield.hbs* file created under *app/templates/components/*and enter the following code:

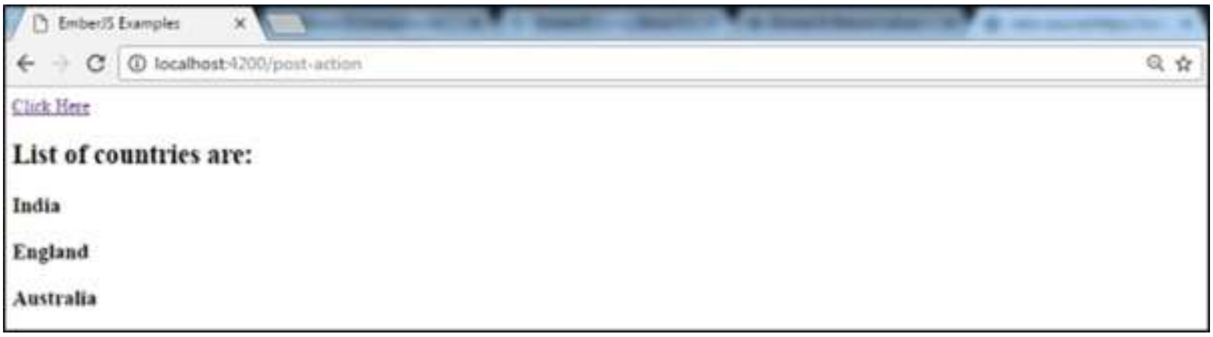
```
<h2>List of countries are:</h2>
// template iterates an array named 'country'
{{#each country as |myval|}}   // each item in an array provided as blobk param 'myval'
   {{ yield myval }}
{{/each}}
```

Output

Run the ember server; you will receive the following output:



When you click on the link, it will display the list of objects from an array as shown in the screenshot below:



List of countries are:

India

England

Australia

EmberJS – Supporting both block and non-block Component Usage

You can support the usage of block and non-block components from single component by using the *hasBlock* property.

Syntax

```
{{#if hasBlock}}
  //code here
{{/if}}
```

Example

The example given below specifies supporting of both block and non-block component usage in one template. Create a route with the name *comp-yield* and open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
```

```
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
Router.map(function() {
  this.route('comp-yield');
});

export default Router;
```

Create the *application.hbs* file and add the following code:

```
// link-to is a handlebar helper used for creating links
{{#link-to 'comp-yield'}}Click Here{{/link-to}}
{{outlet}}
// It is a general helper, where content from other pages will appear inside this section
```

Open the *comp-yield.js* file, which is created under *app/routes/*and enter the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model: function () {
      return {
         title: "Emberjs",
         author: "Tutorialspoint",
         body: "This is introduction"
      };
   }
});
```

Create a component with the name *comp-yield* and open the component template file *comp-yield.hbs* created under *app/templates/* with the following code:

```
{{#comp-yield title=title}}
```

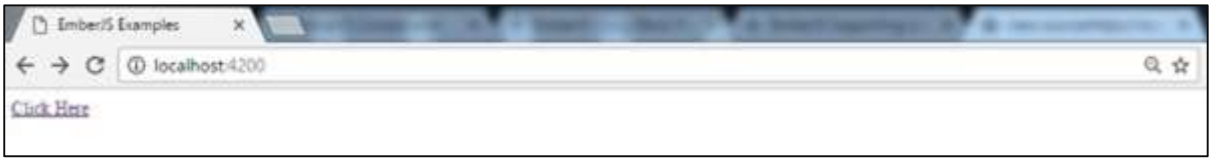
```
   <p class="author">by (blocked name){{author}}</p>
   {{body}}
{{/comp-yield}}
{{comp-yield title=title}}
{{outlet}}
```

Open the *comp-yield.hbs* file created under *app/templates/components/*and enter the following code:

```
{{#if hasBlock}}
   <div class="body">{{yield}}</div>
   {{else}}
   <div class="body">Tutorialspoint data is missing</div>
{{/if}}
{{yield}}
```

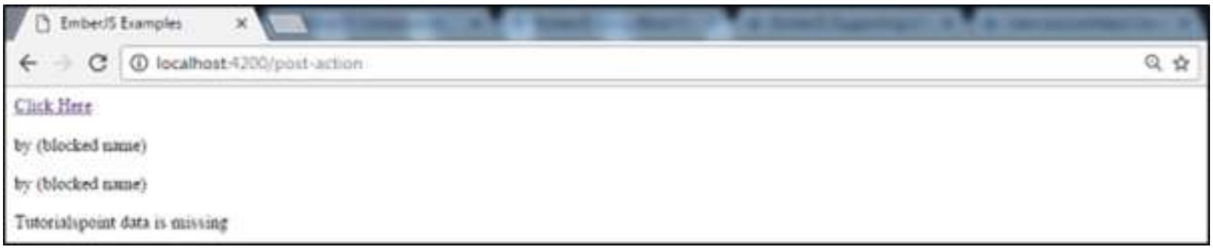
Output

Run the ember server; you will receive the following output:



Click Here

When you click on the link, it will block the names as shown in the screenshot below:



Click Here

by (blocked name)

by (blocked name)

Tutorialspoint data is missing

EmberJS – Handling Events

The user events such as double click, hovering, key press etc can be handled by event handlers. To do this, apply the event name as a method on the component.

For instance, consider we have a template as given below:

```
{{#double-clickable}}
```

```
   // code here
{{/double-clickable}}
```

When you double-click on the element, it will display the message as shown below:

```
import Ember from 'ember';


export default Ember.Component.extend({
  doubleClick() {
    document.write("The double click event has occurred!");
  }
});
```

Event Names

Ember.js contains following built-in events such as touch, keyboard, mouse, form, drag and drop events.

Touch Events

- touchStart
- touchMove
- touchEnd
- touchCancel

Keyboard Events

- keyDown
- keyUp
- keyPress

Mouse Events

- mouseDown
- mouseUp
- contextMenu
- click
- doubleClick
- mouseMove
- focusIn
- focusOut

- mouseEnte
- mouseLeave

Form Events

- submit
- change
- focusIn
- focusOut
- input

HTML5 Drag and Drop Events

- dragStart
- drag
- dragEnter
- dragLeave
- dragOver
- dragEnd
- drop

You can use event handlers to send actions from component to your application. For more information on sending actions, check out the following section.

EmberJS – Sending Actions

You can use event handlers to send actions from component to your application.

Syntax

```
{{comp_name action="name_of_action"}}
```

Example

The example given below specifies sending actions from components to your application. Create a component with the name *comp-yield* and open the component template file *comp-yield.js* created under *app/components/* with the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
   actions: {
      compFunc: function () {
         this.set('title', "Hello...Welcome To Tutorialspoint...");
         // sendAction() method sends the specified action when the component
is used in a template
         this.sendAction();
      }
   }
});
```

Open the *comp-yield.hbs* file created under *app/templates/components/* and enter the following code:

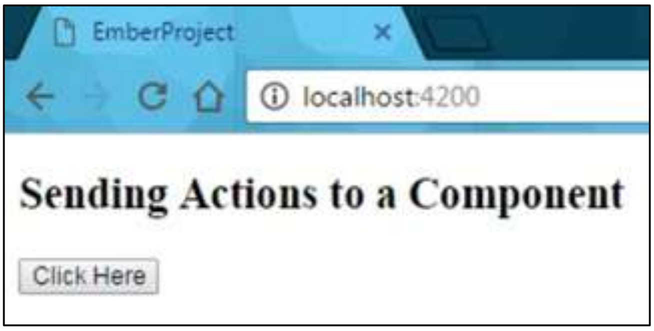
```
<h2>Sending Actions to a Component</h2>
<input type="button" value="Click Here" {{action "compFunc"}} /><br/>
<p><b>{{title}}</b></p>
{{yield}}
```

Create the *application.hbs* file and add the following code:

```
{{comp-yield title=title action="compFunc"}}
{{outlet}}
```

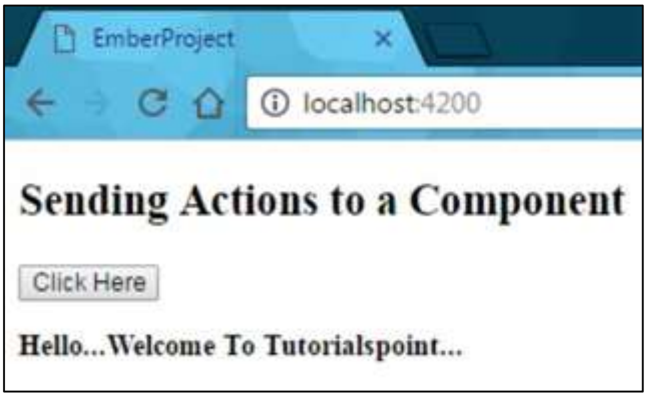
Output

Run the ember server; you will receive the following output:



Sending Actions to a Component

When you click on the button, it will display the text as shown in the screenshot below:



Sending Actions to a Component

Click Here

Hello...Welcome To Tutorialspoint...

EmberJS – Triggering Changes with Actions

Components can trigger the changes and communicate with events by using the actions.

The following table lists down the different ways of triggering the changes with actions:

S.NO.	Triggering Changes Ways & Description
1	**Creating the Component** The component can be created easily in Ember.js and each component must have a dash in their name.
2	**Implementing Action and Designing Child Component** You can implement the action on the parent component by calling its specified action method and create a logic in the child component for the specified action method.
3	**Handling Action Completion and Passing Arguments** The component can handle action's completion by returning a promise and arguments can be passed to a component by using an action helper.
4	**Invoking Actions on Component Collaborators** You can invoke actions on component collaborators directly from template.

EmberJS – Defining a Component

You can easily define the component in Ember.js and each component must have a dash in their name (ex: my-component). Ember.js has the power of defining the component subclasses by using an *Ember.Component* class.

The component can be created by using the following command:

```
ember generate component component-name
```

Example

The example given below describes how to define a component in Ember.js. Create a component with the name *post-action*, which will get define under *app/components/*.

Open the *post-action.js* file and add the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
      toggleBody:['Welcome to Tutorialspoint!!!']
});
```

Now open the component template file *post-action.hbs* with the following code:

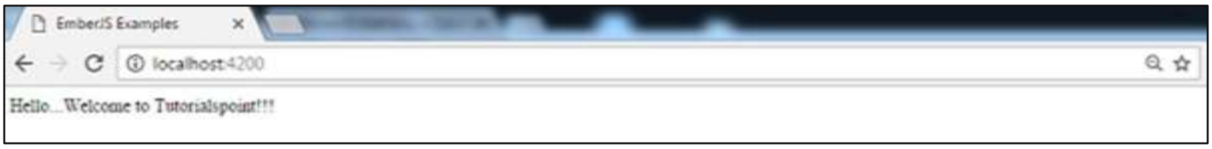
```
{{#each toggleBody as |body|}}
Hello...{{body}}
{{/each}}
{{yield}}
```

Open the *index.hbs* file and add the following code:

```
{{post-action}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Hello...Welcome to Tutorialspoint!!!

EmberJS – Implementing Action and Designing Child Component

You can implement the action on the parent component by calling its specified action method and create a logic in the child component for the specified action method.

Syntax

The action can be implemented as given below:

```
import Ember from 'ember';


export default Ember.Component.extend({
   actions: {
      action_name() {
         //code here
      }
   }
});
```

The child component can be implemented as in the following line of code:

```
<tag_name {{action "action_name"}}>{{Text Here}}</end_of_tag>
```

Example

The example given below specifies implementing action and designing child component in your application. Create a component with the name *ember-actions* and open the component template file *ember-actions.js* created under *app/components/* with the following code:

```
import Ember from 'ember';


export default Ember.Component.extend({
   actions: {
      toggleBody() {
         this.decrementProperty('isShowingBody');
      },
      cancelBody() {
         this.incrementProperty('isShowingBody');
      }
   }
});
```

Open the *ember-actions.hbs* file created under *app/templates/components/*and enter the following code:

```
<button {{action "toggleBody"}}>{{title}}Show (Display Text)</button>
{{#if isShowingBody }}
```

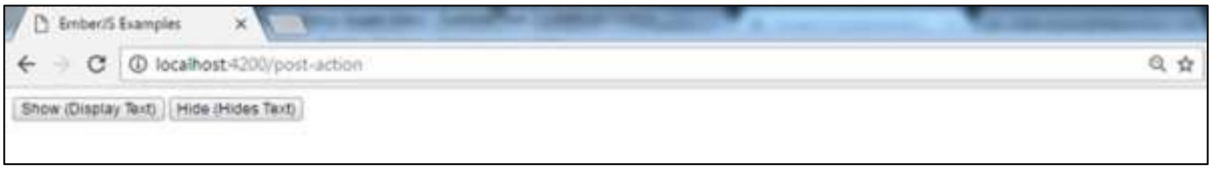
```
  <h2>Welcome to Tutorialspoint...</h2>
{{/if}}
<button class="confirm-cancel" {{action "cancelBody"}}>{{title}}Hide (Hides Text)
</button>
{{yield}}
```

Create the *application.hbs* file and add the following code:

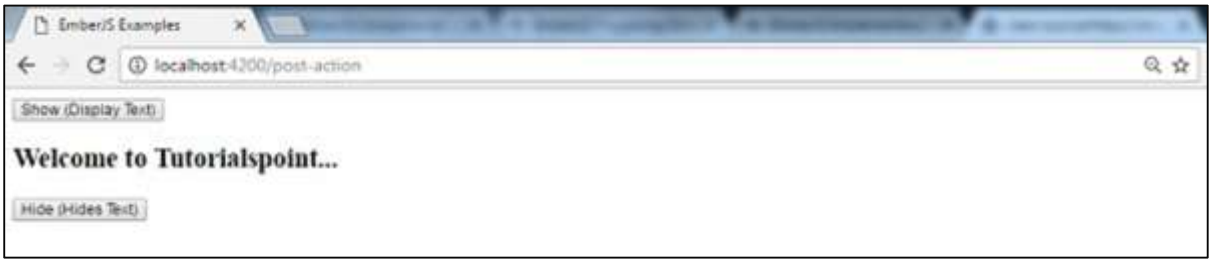
```
{{ember-actions}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



When you click on the 'Show' button, it will display the text and hides the text on clicking the 'Hide' button:



Welcome to Tutorialspoint...

EmberJS – Handling Action Completion and Passing Arguments

The component can handle action's completion by returning a promise and arguments can be passed to a component by using an action helper.

Syntax

The action can be implemented as:

```
import Ember from 'ember';

export default Ember.Component.extend({
   actions: {
      action_name() {
         // code here
      }
   }
});
```

The arguments can be passed to a component as:

```
{{component_name text="text-here" action-helper=(action "action_name" "args")}}
```

Example

The example given below specifies handling action completion and passing arguments in your application. Create a component with the name *ember-actions* and open the component template file *ember-actions.js* created under *app/components/* with the following code:

```
import Ember from 'ember';

export default Ember.Component.extend({
     doubleClick: function() {
    this.toggleProperty('isEditing');
  },
  isEditing: false

});
```

Open the *ember-actions.hbs* file created under *app/templates/components/*and enter the following code:

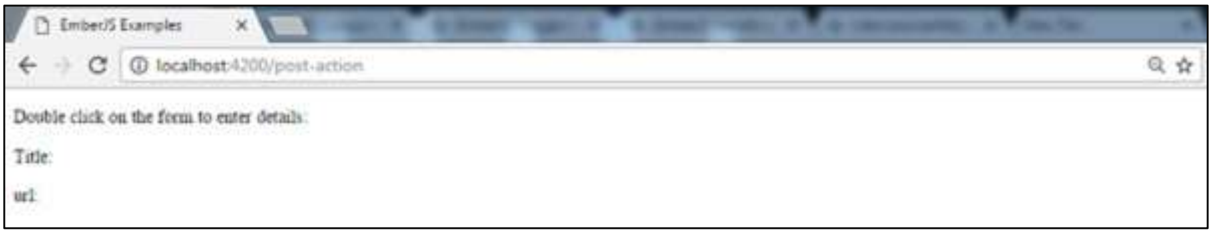
```
{{#if isEditing}}
   <p>Title: {{input value=title}}</p>
   <p>url: {{input value=url}}</p>
   <p>Double click on the save button to save information.</p>
   <button>Save</button>
{{else}}
   <p>Double click on the form to enter details:</p>
   <p>Title: {{title}}</p>
   <p>url: {{url}}</p>
{{/if}}
{{yield}}
```

Create the *application.hbs* file and add the following code:

```
{{ember-actions}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:

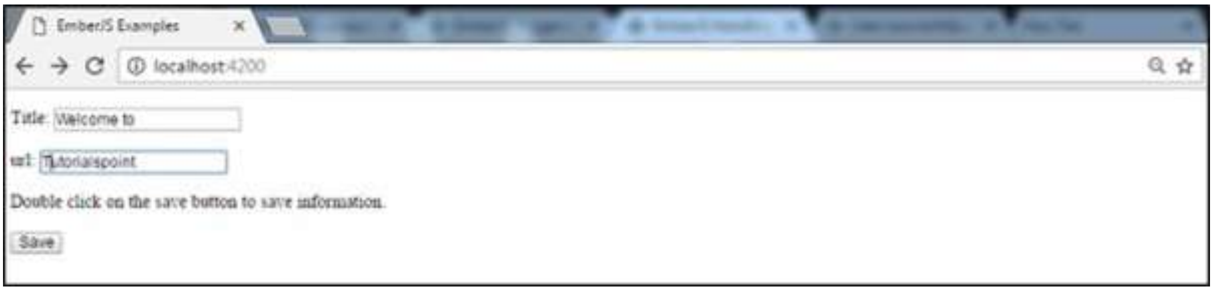


Double click on the form to enter details:

Title:

url:

After double clicking on the form, it will display the form and enter the details in it. Next double-click on the Save button to save the details:



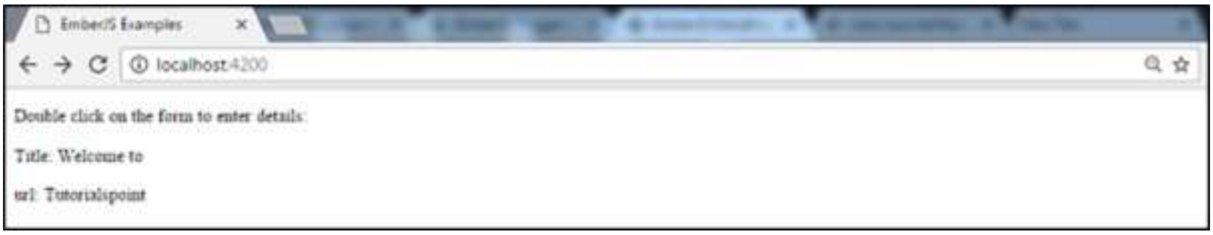
Title: Welcome to

url: Tutorialspoint

Double click on the save button to save information.

Save

Now you will see the saved details as shown in the screenshot below:



Double click on the form to enter details:

Title: Welcome to

url: Tutorialspoint

EmberJS – Invoking Actions on Component Collaborators

You can invoke actions on component collaborators directly from template.

Syntax

```
import Ember from 'ember';


export default Ember.Component.extend({
  target_attribute: Ember.inject.service(),
  // code for component implementation
});
```

Example

The example given below specifies invoking actions on component collaborators directly from template in your application. Create a component with the name *ember-actions* and open the component template file *ember-actions.js* created under *app/components/* with the following code:

```
import Ember from 'ember';
var inject = Ember.inject;


export default Ember.Component.extend({
   message: inject.service(),
   text: 'This is test file',
   actions: {
      pressMe: function () {
         var testText = this.get('start').thisistest();
         this.set('text', testText);
      }
   }
});
```

Create a service, which can be made available in the different parts of the application. Use the following command to create the service:

```
ember generate service emberactionmessage
```

Now open the *emberactionmessage.js* service file, which is created under *app/services/* with the following code:

```
import Ember from 'ember';
```

```
export default Ember.Service.extend({
   isAuthenticated: true,
   thisistest: function () {
      return "Welcome to Tutorialspoint";
   }
});
```

Next create an initializer, which configures the application as it boots. The initializer can be created by using the following command:

```
ember generate initializer init
```

Open the *init.js* initializer file, which is created under *app/initializers/* with the following code:

```
export function initialize(app) {
   app.inject('component', 'start', 'service:emberactionmessage');
}

export default {
   name: 'init',
   initialize: initialize
};
```

Open the *ember-actions.hbs* file created under *app/templates/components/*and enter the following code:

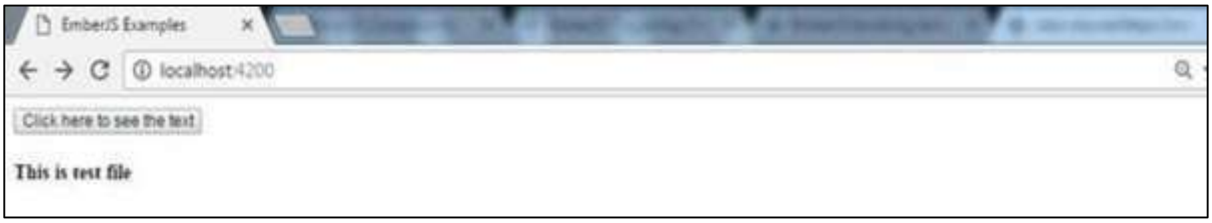
```
<button {{action "pressMe"}}>Click here to see the text</button><br>
<h4>{{text}}</h4>
{{yield}}
```

Create the *application.hbs* file and add the following code:

```
{{ember-actions}}
{{outlet}}
```

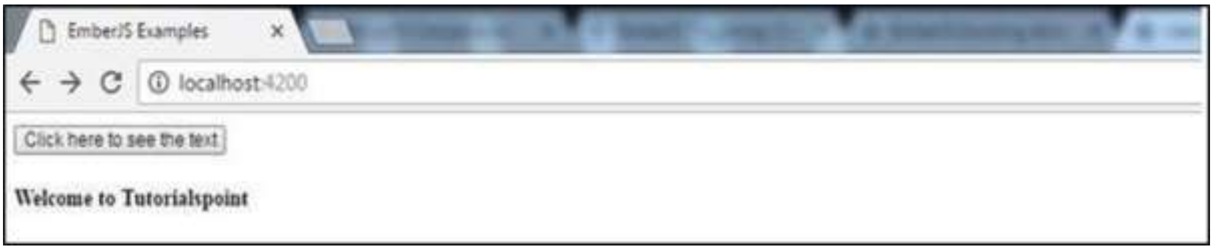
Output

Run the ember server; you will receive the following output:



This is test file

Next click on the button, it will display the text from the service page as shown in the screenshot below:



Welcome to Tutorialspoint

9. EmberJS – Models

Model is a class that extends the functionality of the Ember Data. When a user refreshes the page, the contents of page should be represented by a model. In Ember.js, every route has an associated model. The model helps to improve the performance of application. The Ember Data manipulates the stored data in the server and also works easily with streaming APIs like socket.io and Firebase or WebSockets.

Core Components

- Store
- Models
- Records
- Adapter
- Caching

Store

The store is a central repository and cache of all records available in an application. The route and controllers can access the stored data of your application. The DS.Store is created automatically to share the data among the entire object.

```
import Ember from 'ember';


export default Ember.Route.extend({
  model() {
     return this.store.find();
  }
});
```

Models

Model is a class that extends the functionality of the Ember Data, which specifies relationships with other objects. When a user refreshes the page, the contents of page should be represented by a model.

```
import DS from 'ember-data';


export default DS.Model.extend({
  owner: DS.attr(),
  city: DS.attr()
});
```

Records

A record is an instance of a model that includes the information, which is loaded from a server and you can identify the record by its model *type* and *ID*.

```
// It finds the record of type 'person' and an 'ID' of 1
this.get('store').findRecord('person', 1); // => { id: 1, name: 'steve-buscemi'
}
```

Adapter

An adapter is an object that is responsible for translating requested records from Ember into appropriate calls to particular server backend. For instance, if you want to find a person with ID of 1, then Ember will load the URL by using HTTP as */person/1*.

Caching

The records can be cached automatically by the store and returns the same object instance when you load the records from the server for the second time. This improves the performance of your application and displays the application UI to the user as fast as possible.

The following table lists down the details about models:

S.NO.	Model Ways & Description
1	**Defining Models** Model is a simple class that extends the functionality of the Ember Data.
2	**Finding Records** You can retrieve the records by using the Ember data store.
3	**Creating and Deleting Records** You can create and delete the records on the instance of model.
4	**Relationships** Ember.js provides relationship types to specify how the models are related to each other.
5	**Pushing Records into the Store** You can push the records into the store's cache without requesting the records from an application.
6	**Handling Metadata** Metadata is a data that is used for specific model or type instead of using record.

7	**Customizing Adapters** Ember.js Adapter specifies how data is kept on at the backend data store such as URL format and REST API headers.

EmberJS – Defining Models

Model is a class that extends the functionality of the Ember Data which specifies relationships with other objects. In Emberjs, every route has an associated model and when a user refreshes the page, the contents of page should be represented by a model.

The model can be created by using the following command:

```
ember generate model model_name
```

It creates the file structure under *app/models/model_name.js* as shown below:

```
import DS from 'ember-data';
export default DS.Model.extend({
});
```

Defining Attributes

The *DS.attr* is used to specify attributes for a model. This also takes an optional second parameter as hash.

For instance:

```
import DS from 'ember-data';


export default DS.Model.extend({
   bookName: DS.attr(),
   authorName: DS.attr()
});
```

For more about defining models and attributes along with an example, see the following section.

EmberJS – Router Dynamic Models

You can define multiple models by using *RSVP.hash*, which uses the objects to return the promises.

Syntax

```
Ember.Route.extend({
   model() {
      return Ember.RSVP.hash({
      //code here
       })
   }
});
```

Example

The example below shows how to specify multiple models for displaying data by using *RSVP.hash*. Create a new model as specified in the previous chapters. Here we have created two models as *rental* and *review* under *app/models/*.

Now open *rental.js* file and provide its attributes:

```
import DS from 'ember-data';

export default DS.Model.extend({
  owner: DS.attr(),
  city: DS.attr()
});
```

Now open *review.js* file and provide its attributes:

```
import DS from 'ember-data';

export default DS.Model.extend({
  author: DS.attr(),
  rating: DS.attr(),
  content: DS.attr()
});
```

You can return both the rentals and review models in one model hook and display them on index page(app/routes/index.js) by using the code given below:

```
import Ember from 'ember';


export default Ember.Route.extend({
  model() {
    // The RSVP.hash methos is built with RSVP.js library that allows to load
multiple JavaScript promises
    return Ember.RSVP.hash({
       // Find the records for the given type and returns all the records of
this type present in the store
      rentals: this.store.findAll('rental'),
      reviews: this.store.findAll('review')
    });
  },
});
```

Now you can access the model data in RSVP hash which are referenced in index template, i.e., in the *app/templates/index.hbs* file:

```
<h3>Members - City </h3>
<ul>
  {{#each model.rentals as |rental|}}
    <li>{{rental.owner}} - {{rental.city}}</li>
  {{/each}}
</ul>


<h3>Member Reviews </h3>
<ul>
  {{#each model.reviews as |review|}}
<li>{{review.rating}} - {{review.content}} - by {{review.author}}</li>
  {{/each}}
</ul>
```

The code displays data from *Firebase database* which is a cloud database that stores information in JSON format. Therefore to make use of this database, create an account by using the Firebase's website.

Install the EmberFire to interface with the Firebase by Ember data.

```
ember install emberfire
```

It adds the EmberFire to *package.json* and firebase to *bower.json*.

Configuring Firebase

Login to the Firebase account and click on the *CREATE NEW PROJECT* button. Provide the same name to Firebase project which is given to the Ember application.

Open the *config/environment.js* file to add the configuration info for the Ember application from the project which has been created on Firebase's website.

```
module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'super-rentals',
    environment: environment,
    rootURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember canary build
        // e.g. 'with-controller': true
      }
    },

    firebase: {
      apiKey: "AIzaSyAqxzlKErYeg64iN_uROKA5eN40locJSXY",
      authDomain: "multiple-models.firebaseapp.com",
      databaseURL: "https://multiple-models.firebaseio.com",
      storageBucket: "multiple-models.appspot.com"
    },

    APP: {
      // Here you can pass flags/options to your application instance
      // when it is created
    }
  };
```

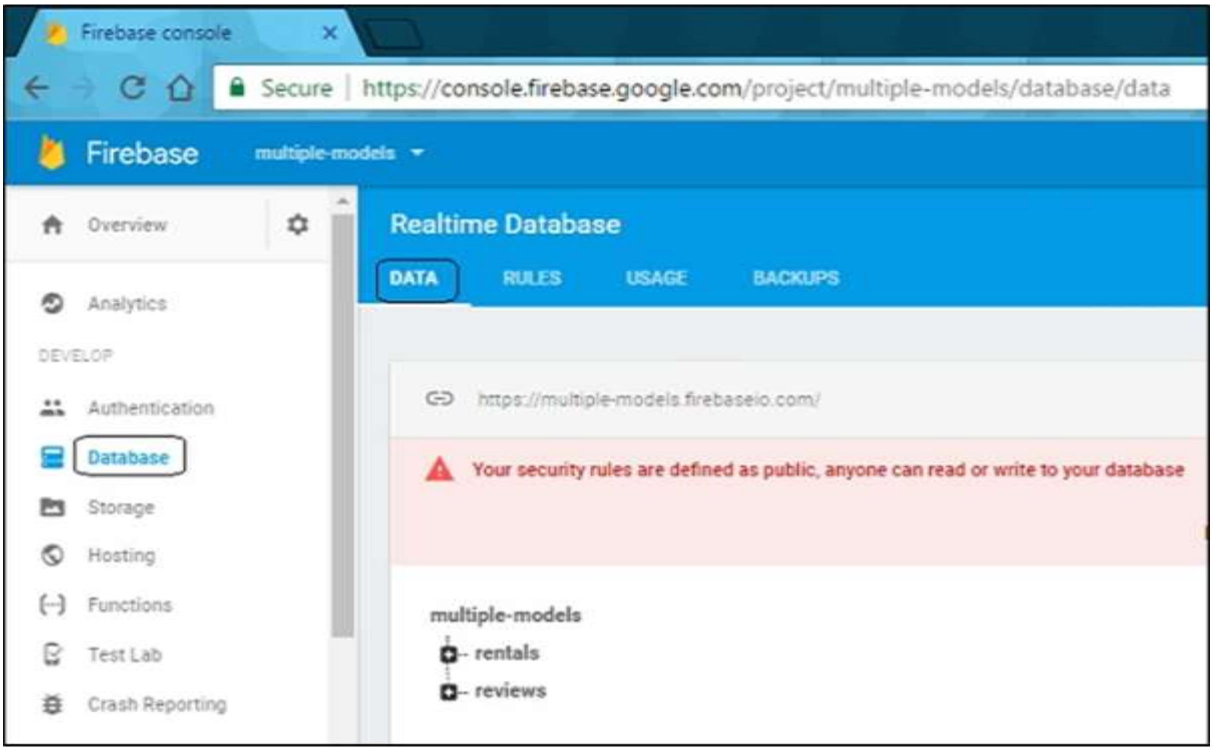
```
   //other code here
```

You need to change the firebase section defined under the *ENV* section. Click on the Firebase project and click the *Add Firebase to your web app* button to include *apiKey*, *authDomain*, *databaseURL* and *storageBucket* fields from the firebase project to firebase section provided in the *environment.js* file. After configuring the EmberFire, restart the server for applying changes.

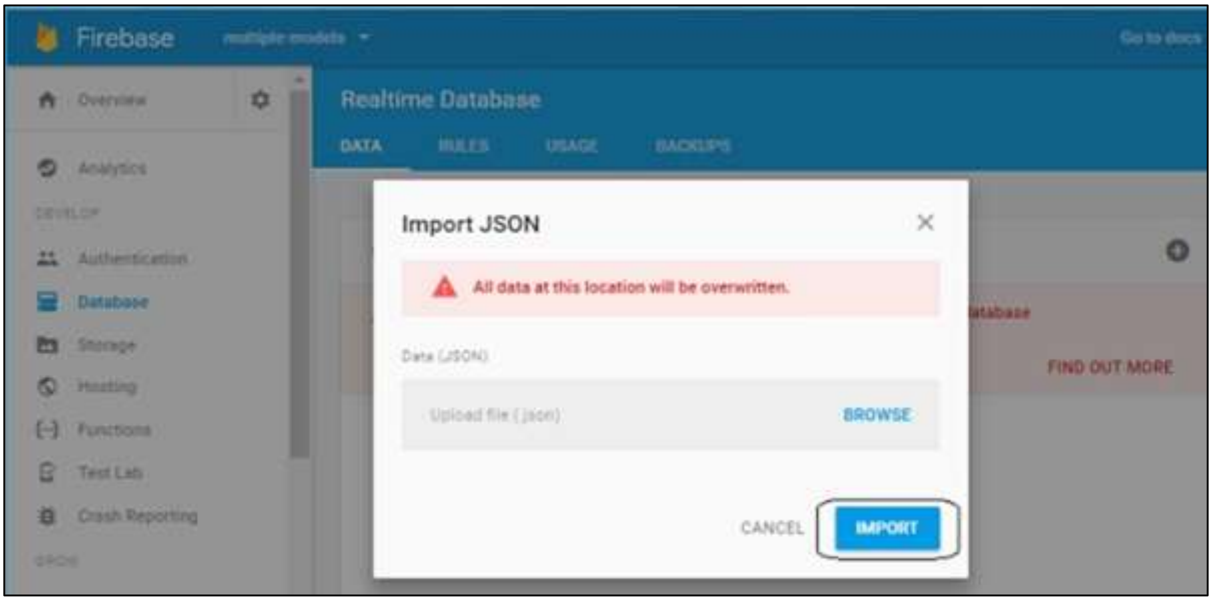
Now import the data to Firebase by using the json file. In this app, we have created a file called *rentals.json* which contains the data in JSON format.

```
{ "rentals": [{
    "owner": "Will Smith",
    "city": "San Francisco"
  }, {
    "owner": "John Davidson",
    "city": "Seattle"
  }, {
    "owner": "Shane Watson",
    "city": "Portland"
  }],
  "reviews": [{
    "author": "Will Smith",
    "rating": 4,
    "content": "Good Product"
  }, {
    "author": "John Davidson",
    "rating": 5,
    "content": "Nice Product"
  }]
}
```

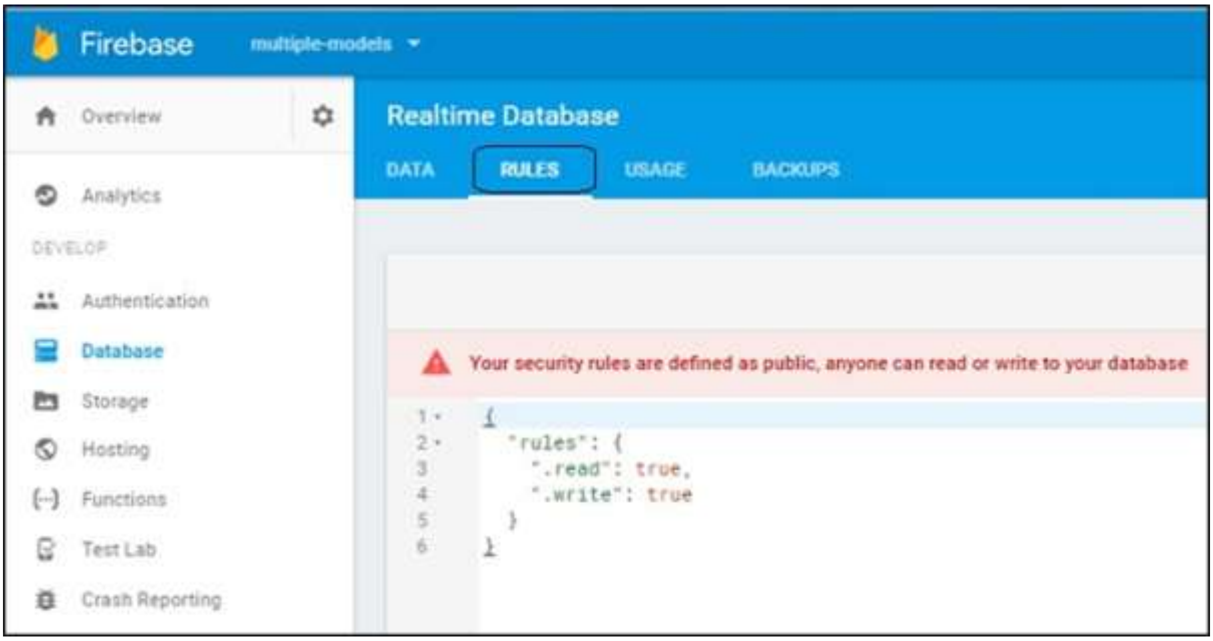
Go to the Firebase console, click on the *Database* section and select the *Data* tab.



Click on the three dots on the right hand side and select *Import JSON* option. Next, browse the json file which you have created and click on the *IMPORT* button.



Now set the Firebase permissions to the new database. Go to the *Rules* tab and click on the *PUBLISH* to update the json.



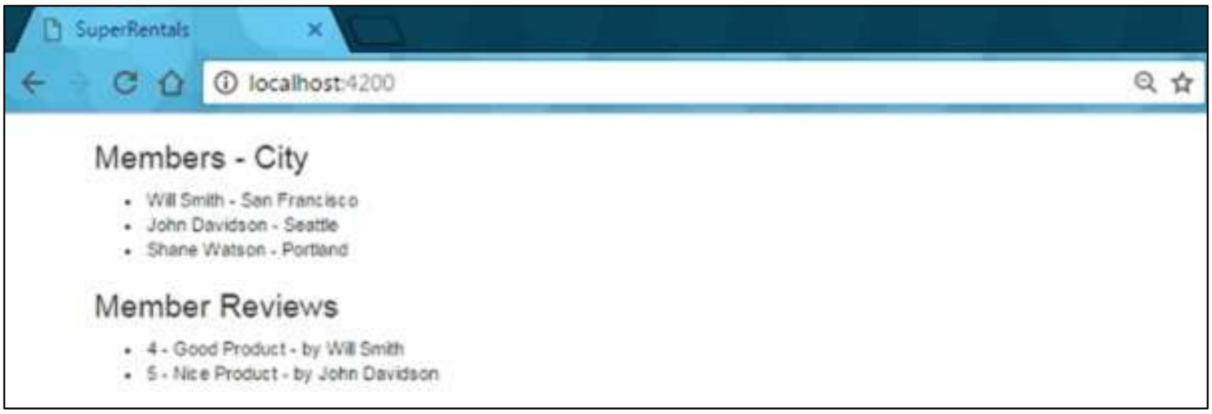
Your security rules are defined as public, anyone can read or write to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

By changing the rules, anyone can read or write to your database.

Output

Run the ember server; you will receive the following output:



Members - City

- Will Smith - San Francisco
- John Davidson - Seattle
- Shane Watson - Portland

Member Reviews

- 4 - Good Product - by Will Smith
- 5 - Nice Product - by John Davidson

EmberJS – Finding Records

You can retrieve the records by using the Ember data store which uses store object's built-in method for finding the records based on the arguments.

The records can be retrieved of a given type by using the following method:

```
store.findAll()
```

Syntax

```
import Ember from 'ember';


export default Ember.Route.extend({
  model() {
    return this.store.findAll('model_name');
  }
});
```

For more about finding records along with an example, see the following section.

EmberJS – Router Dynamic Models

You can define the multiple models by using *RSVP.hash*, which uses the objects to return the promises.

Syntax

```
Ember.Route.extend({
   model() {
      return Ember.RSVP.hash({
      // code here
      })
   }
});
```

Example

The example below shows how to specify multiple models for displaying data by using *RSVP.hash*. Create a new model as specified in the previous chapters. Here we have created two models as *rental* and *review* under *app/models/*.

Now open *rental.js* file and provide its attributes:

```
import DS from 'ember-data';


export default DS.Model.extend({
  owner: DS.attr(),
  city: DS.attr()
});
```

Now open *review.js* file and provide its attributes:

```
import DS from 'ember-data';


export default DS.Model.extend({
  author: DS.attr(),
  rating: DS.attr(),
  content: DS.attr()
});
```

You can return both the rentals and review models in one model hook and display them on index page(app/routes/index.js) by using the code given below:

```
import Ember from 'ember';


export default Ember.Route.extend({
  model() {
    // The RSVP.hash methos is built with RSVP.js library that allows to load
multiple JavaScript promises
    return Ember.RSVP.hash({
       // Find the records for the given type and returns all the records of
this type present in the store
      rentals: this.store.findAll('rental'),
      reviews: this.store.findAll('review')
    });
  },
});
```

Now you can access the model data in RSVP hash which are referenced in index template, i.e., in the *app/templates/index.hbs* file:

```
<h3>Members - City </h3>
<ul>
  {{#each model.rentals as |rental|}}
    <li>{{rental.owner}} - {{rental.city}}</li>
  {{/each}}
</ul>


<h3>Member Reviews </h3>
<ul>
  {{#each model.reviews as |review|}}
```

```
<li>{{review.rating}} - {{review.content}} - by {{review.author}}</li>
  {{/each}}
</ul>
```

The code displays data from *Firebase database* which is a cloud database that stores information in JSON format. Therefore to make use of this database, create an account by using the Firebase's website.

Install the EmberFire to interface with the Firebase by Ember data.

```
ember install emberfire
```

It adds the EmberFire to *package.json* and firebase to *bower.json*.

Configuring Firebase

Login to the Firebase account and click on the *CREATE NEW PROJECT* button. Provide the same name to Firebase project which is given to Ember application.

Open the *config/environment.js* file to add the configuration info for the Ember application from the project which has been created on Firebase's website.

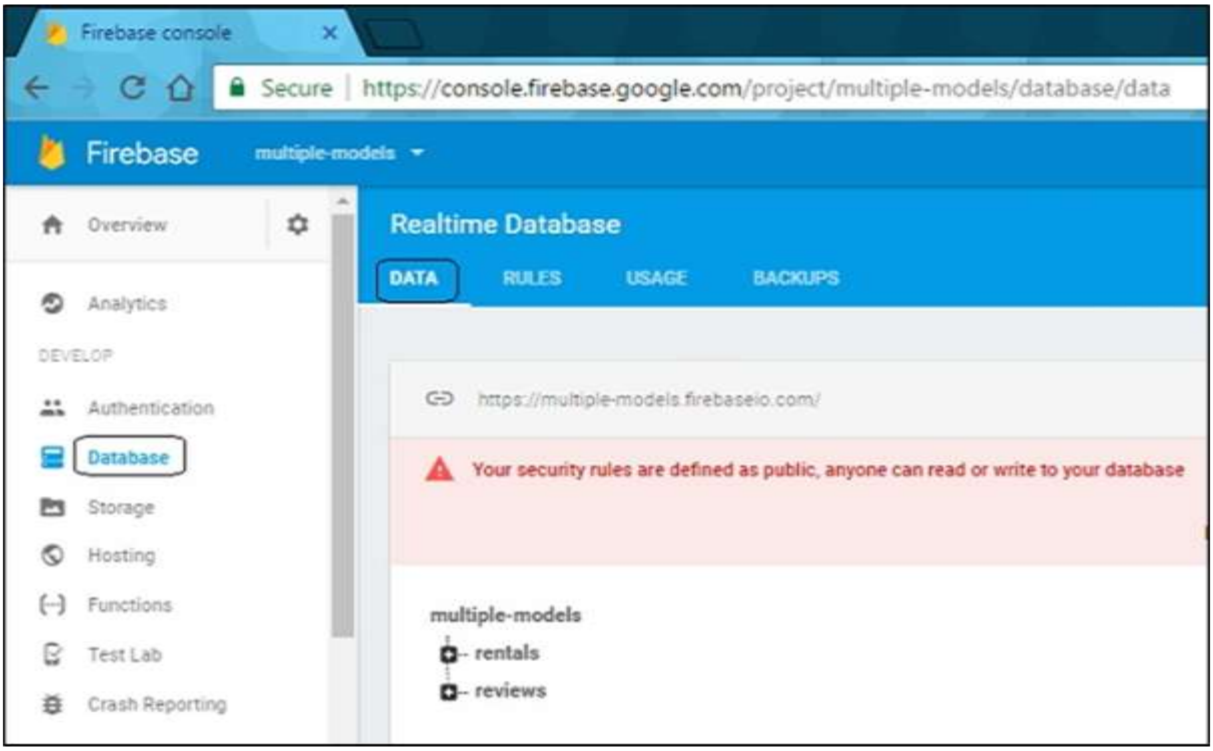
```
module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'super-rentals',
    environment: environment,
    rootURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember canary build
        // e.g. 'with-controller': true
      }
    },

    firebase: {
      apiKey: "AIzaSyAqxzlKErYeg64iN_uROKA5eN40locJSXY",
      authDomain: "multiple-models.firebaseapp.com",
      databaseURL: "https://multiple-models.firebaseio.com",
      storageBucket: "multiple-models.appspot.com"
    },

    APP: {
      // Here you can pass flags/options to your application instance
      // when it is created
    }
  };

  //other code here
```

You need to change the firebase section defined under the *ENV* section. Click on the Firebase project and click the *Add Firebase to your web app* button to include *apiKey*, *authDomain*, *databaseURL* and *storageBucket* fields from the firebase project to firebase section provided in the *environment.js* file. After configuring the EmberFire, restart the server to apply changes.

Now import the data to Firebase by using the json file. In this app, we have created a file called *rentals.json* which contains the data in JSON format.

```
{ "rentals": [{
    "owner": "Will Smith",
    "city": "San Francisco"
  }, {
    "owner": "John Davidson",
    "city": "Seattle"
  }, {
    "owner": "Shane Watson",
    "city": "Portland"
  }],
  "reviews": [{
    "author": "Will Smith",
    "rating": 4,
    "content": "Good Product"
  }, {
    "author": "John Davidson",
    "rating": 5,
    "content": "Nice Product"
  }]
}
```

Go to Firebase console, click on the *Database* section and select the *Data* tab.



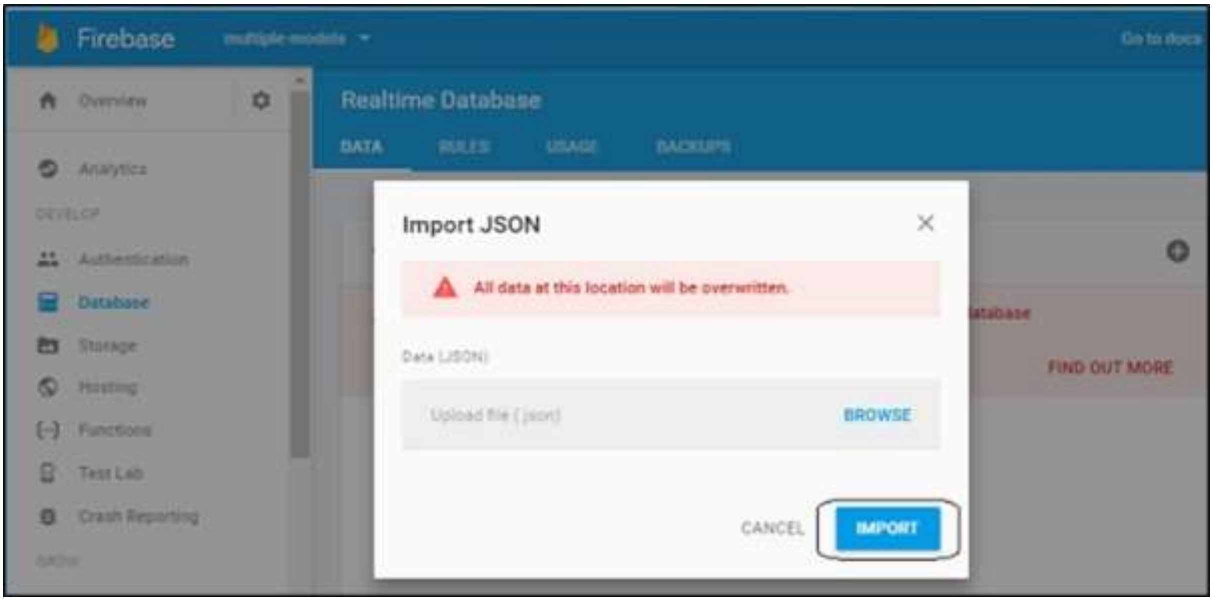
https://console.firebase.google.com/project/multiple-models/database/data

https://multiple-models.firebaseio.com/

Your security rules are defined as public, anyone can read or write to your database

```
multiple-models
  rentals
  reviews
```

Click on the three dots on the right hand side and select the *Import JSON* option. Next browse the json file which you have created and click on the *IMPORT* button.



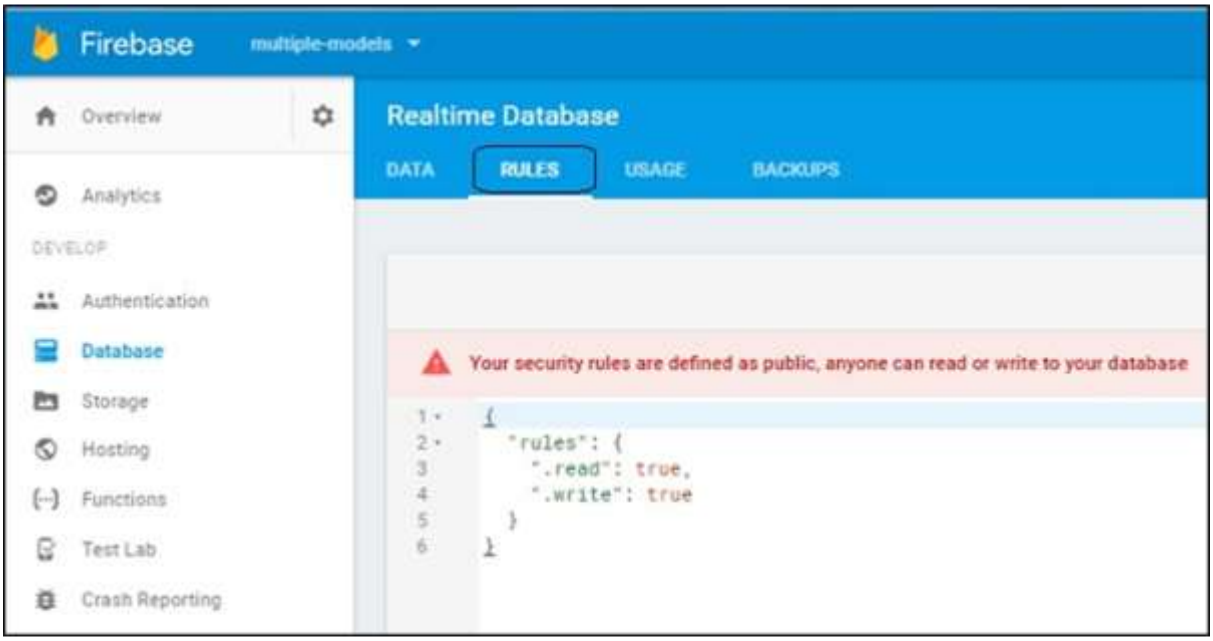
Import JSON

All data at this location will be overwritten.

Data (JSON)

Upload file (.json)

Now set the Firebase permissions to the new database. Go to the *Rules* tab and click on the *PUBLISH* button to update the json.



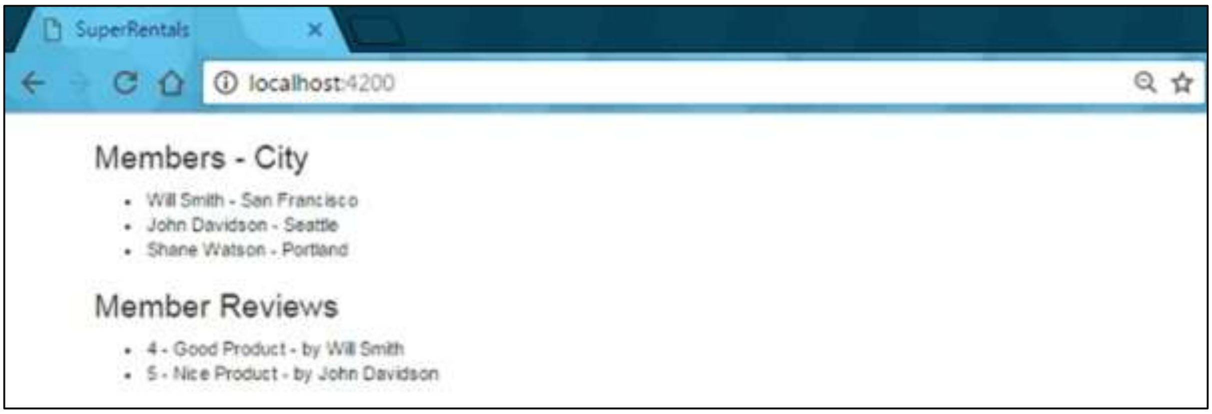
Your security rules are defined as public, anyone can read or write to your database

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

By changing the rules, anyone can read or write to your database.

Output

Run the ember server; you will receive the following output:



Members - City

- Will Smith - San Francisco
- John Davidson - Seattle
- Shane Watson - Portland

Member Reviews

- 4 - Good Product - by Will Smith
- 5 - Nice Product - by John Davidson

EmberJS – Creating and Deleting Records

You can create and delete the records on the instance of model.

Syntax

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    // code here
  },
  actions:{
     addNewCategory(id, name) {
      this.controller.get('model').pushObject({ var1,va2});
    },

    deleteCategory(category) {
      this.controller.get('model').removeObject(model_name);
    }
  }
});
```

Example

The example given below shows creation and deletion of records. Create a new route with the name *record_demo* and create one more route within this route and name it as *categories*. Now open the *router.js* file to define the URL mappings:

```
import Ember from 'ember';      // Access to Ember.js library as variable Ember
import config from './config/environment';
// It provides access to app's configuration data as variable config

// The const declares read only variable
const Router = Ember.Router.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

// Defines URL mappings that takes parameter as an object to create the routes
```

```
Router.map(function() {
  this.route('record_demo', function() {
    this.route('categories');
  });
});

// It specifies Router variable available to other parts of the app
export default Router;
```

Open the *application.hbs* file created under *app/templates/* with the below code:

```
{{#link-to 'record_demo'}}Go to Records demo page{{/link-to}}
{{outlet}}
```

When you click the above link, it will open the record_demo template page, which is created under *app/templates/*. The *record_demo.hbs* file contains the fllowing code:

```
<h2>Welcome...Click the below link for Categories page</h2>
{{#link-to 'record_demo.categories'}}Go to Categories page{{/link-to}}
{{outlet}}
```

The above template page opens the *categories.hbs* file, which is created under *app/templates/record_demo* and contains the following code:

```
<h2>Categories Page</h2>
<form>
  <label>ID:</label>
    {{input value=newCategoryId}}
  <label>NAME:</label>
    {{input value=newCategoryName}}
  // when user adds records, the 'addNewCategory' function fires and adds the
records to model
  <button type="submit" {{action 'addNewCategory' newCategoryId
newCategoryName}}>Add to list</button>
</form>

<ul>
  {{#each model as |category|}}
    <li>
      Id: {{category.id}}, Name: {{category.name}}
      // when user delete records, the ‘deleteCategory’ function fires and
remove the records from model
```

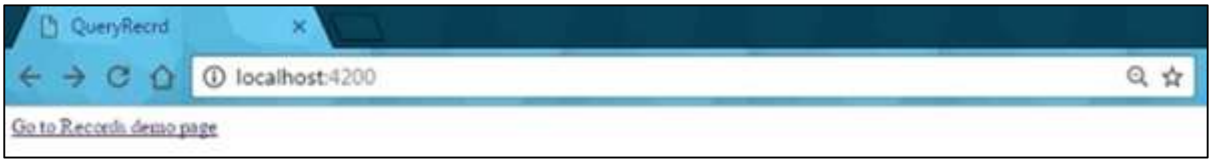
```
      <button {{action 'deleteCategory' category}}>Delete</button>
    </li>
  {{/each}}
</ul>
// it counts the number of added records and removed records from the model
<strong>Category Counter: {{model.length}}</strong>
{{outlet}}
```

Now open the *categories.js* file created under *app/routes/record_demo* with the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model() {
      //model will display these records when you execute the code
      return [{
         id: 1,
         name: 'Category One'
      }, {
         id: 2,
         name: 'Category Two'
      }];
   },

   actions: {
      // it adds records to model
      addNewCategory(id, name) {
         this.controller.get('model').pushObject({id,name});
      },
      // it removes the records from model
      deleteCategory(category) {
         this.controller.get('model').removeObject(category);
      }
   }
});
```

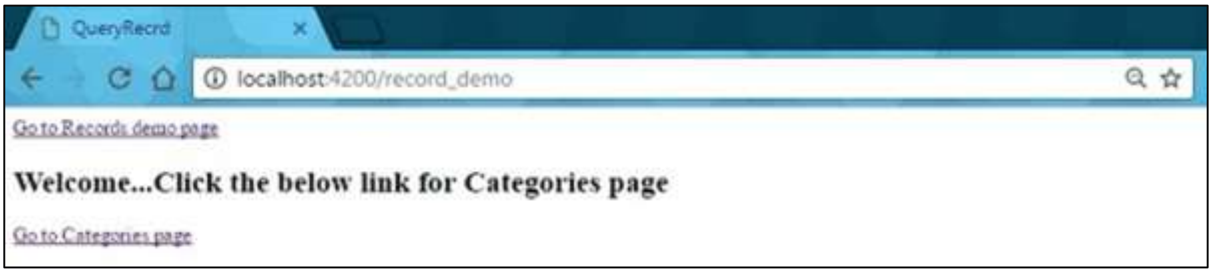
Output

Run the ember server; you will receive the following output:



Go to Records demo page

When you click on the link, it will open the records_demo page with the categories page link:

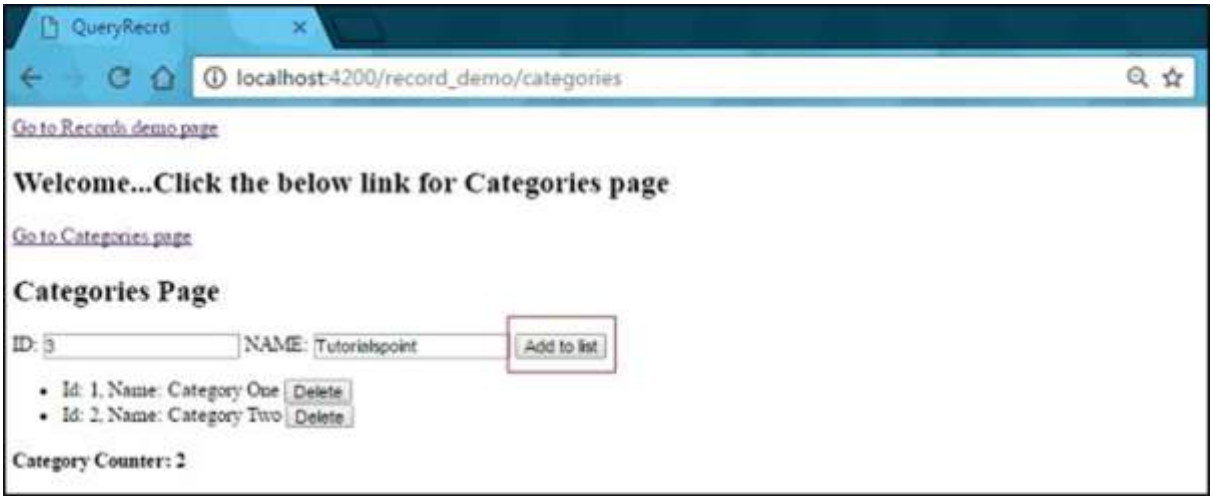


Go to Records demo page

Welcome...Click the below link for Categories page

Go to Categories page

Next, the categories template page will open. Enter the id and name in the input box and click the *Add to list* button as shown in the screenshot below:



Go to Records demo page

Welcome...Click the below link for Categories page

Go to Categories page

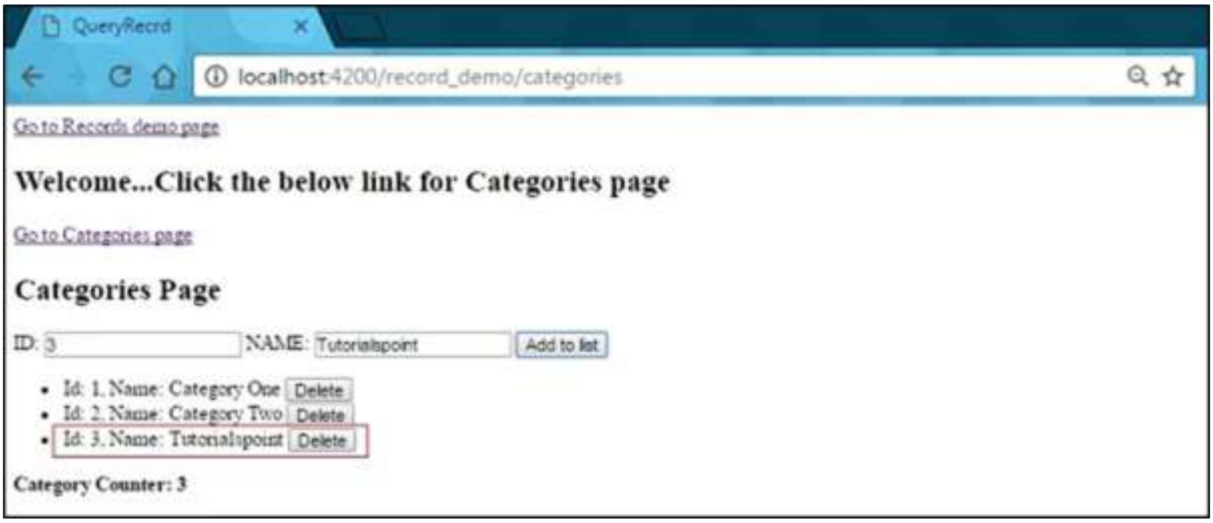
Categories Page

ID: 3 NAME: Tutorialspoint

- Id: 1, Name: Category One
- Id: 2, Name: Category Two

Category Counter: 2

Next, click on the add button; you will see the added records in the list and number of count will get incremented:



Go to Records demo page

Welcome...Click the below link for Categories page

Go to Categories page

Categories Page

ID: 3 NAME: Tutorialspoint

- Id: 1. Name: Category One
- Id: 2. Name: Category Two
- Id: 3. Name: Tutorialspoint

Category Counter: 3

If you want to remove the records from the list, then click the *Delete* button.

EmberJS – Relationships

Ember.js provides relationship types to specify how the models are related to each other. There are different relationship types such as One-to-One relationship can be used with *DS.belongsTo*, One-to-Many relationship can be used with *DS.hasMany* along with *DS.belongsTo* and Many-to-Many relationship can be used with *DS.hasMany*.

Syntax

```
import DS from 'ember-data';


export default DS.Model.extend({
  var_name1: DS.belongsTo('model_name1'),
  var_name2: DS.hasMany('model_name2')
});
```

Example

The example given below shows the use of relationship types. Create two adapters with the names *account* and *staff* by using the following command:

```
ember generate adapter adapter_name
```

Now open the *app/adapters/account.js* file and add the following code:

```
import ApplicationAdapter from './application';
// created an "account" array to store relationship data
const account = {
   "data": {
      "type": "account",
      "id": "100",

      "relationships": {
         "secondVal": {
            "data": {
               "type": "staff",
               "id": "2"
            }
         },
         "firstVal": {
            "data": {
               "type": "staff",
               "id": "1"
            }
         }
      }
   }
};

export default ApplicationAdapter.extend({
   // this method fetches data from 'staff' adapter
   findRecord() {
      //returns the data from array
      return account;
   }
});
```

Open the *app/adapters/staff.js* file and add the following code:

```
import ApplicationAdapter from './application';
import Ember from 'ember';

// values given for type and id
const relval1 = {
   data: {
      type: "staff",
      id: "1",
      attributes: {
         name: 'JavaScript'
      }
   }
};
const relval2 = {
   data: {
      type: "staff",
      id: "2",
      attributes: {
         name: 'jQuery'
      }
   }
};

// the variable 'relval3' pushes the data to 'relval1' and 'relval2'
const relval3 = Ember.A();
relval3.pushObject(relval1);
relval3.pushObject(relval2);

export default ApplicationAdapter.extend({
   findRecord(store, type, id) {
      // finds the item id and returns to 'relval3' variable
      let valret = relval3.find(function (item) {
         return id === Ember.get(item, 'data.id');
      });
      //the searched item will passed to 'relval3' from 'valret' variable
      return valret;
   }
});
```

Create two models with the names *account* and *staff*. Open the *app/models/account.js* file and include the following code:

```
import DS from 'ember-data';
import Model from "ember-data/model";
import attr from "ember-data/attr";
// defines one-to-one and one-to-many relationship between models
import { belongsTo, hasMany } from "ember-data/relationships";

export default DS.Model.extend({
  // when async is 'true', it will fetch related entries
  firstVal: belongsTo('staff', {async: true}),
  secondVal: belongsTo('staff', {async: true})
});
```

Now open the *app/models/staff.js* file and include the following code:

```
import DS from 'ember-data';
import Model from "ember-data/model";
import attr from "ember-data/attr";
import { belongsTo, hasMany } from "ember-data/relationships";

export default DS.Model.extend({
   // specifying attributes using 'attr()' method
   name: attr()
});
```

Next, create a route and name it as *application.js*. Open this file, which is created under *app/routes/* and add the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({
   model(){
      // returns the value of model() hook
      return this.get('store').findRecord('account', 100);
   //retrieve a record for specific id
   }
});
```

Create a serializer with the name *application* by using the following command:

```
ember generate serializer serializer_name
```

Open the *app/serializers/application.js* file and add the following code:

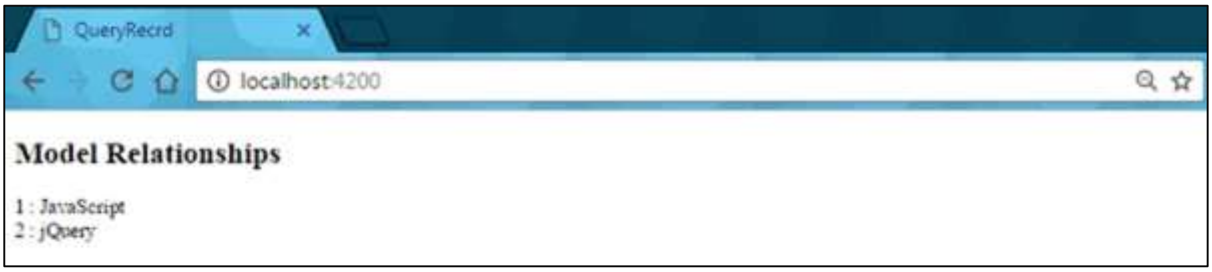
```
import DS from 'ember-data';

// it is the default serializer and works with JSON API backends
export default DS.JSONAPISerializer.extend({
   // keyForRelationship() method overwrites the naming conventions
   keyForRelationship: function(key, relationship, method) {
      return Ember.String.camelize(key);  //returns the lowerCamelCase form of a string
   }
});
```

Open the *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Model Relationships</h2>
// display the id along with the name
{{model.firstVal.id}} : {{model.firstVal.name}}
<br>
{{model.secondVal.id}} : {{model.secondVal.name}}
{{outlet}}
```

Output

Run the ember server; you will receive the following output:



Model Relationships

1 : JavaScript
2 : jQuery

EmberJS – Pushing Records

You can push the records into the store's cache without requesting the records from an application. The store has the ability to return the record, if it is asked by a route or controller only when record is in the cache.

Example

The example given below shows the pushing of records into the ember firebase. Open the *application.hbs* file created under *app/templates/* with the following code:

```
<h2>Pushing Record into Store</h2>
<form>
   {{input type="name" value=nameAddress placeholder="Enter the text"
autofocus="autofocus"}}
   // when user clicks the send button, the 'saveInvitation' action will get triggered
   <button {{action 'saveInvitation'}} >Send</button>
</form>

{{#if responseMessage}}
   // display the response sessage after sending the text successfully
   {{responseMessage}}
{{/if}}
{{outlet}}
```

Create a model with the name *invitation*, which will get created under *app/models/*. Open the file and include the following code:

```
import DS from 'ember-data';

export default DS.Model.extend({
  // specifying attribute using 'attr()' method
  name: DS.attr('string')
});
```

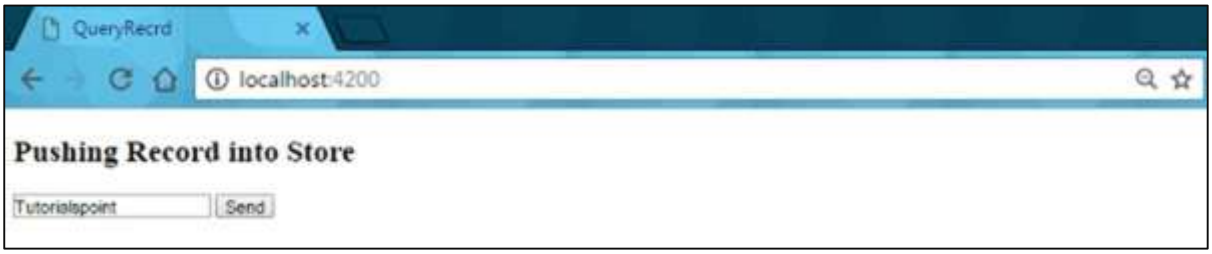
Next, create a controller with the name *application*, which will be created under *app/controllers/*. Open the file and add the following code:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  headerMessage: 'Coming Soon',
  // displays the response message after sending record to store
  responseMessage: '',
  nameAddress: '',

  actions: {
    // this action name which fires when user clicks send button
    saveInvitation() {
      const name = this.get('nameAddress');
      // create the records on the store by calling createRecord() method
      const newInvitation = this.store.createRecord('invitation', { name: name });
      newInvitation.save();  //call the save() method to persist the record to the backend
      this.set('responseMessage', `Thank you! We have saved your Name: 
${this.get('nameAddress')}`);
      this.set('nameAddress', '');
    }
  }
});
```

You can store the information in JSON format on Ember Firebase. To do this, you need to create account by using the [Firebase's website](). For more information about how to create and configure the Firebase in your application, click this link - [https://www.tutorialspoint.com/emberjs/route_multiple_models.htm]().

Output

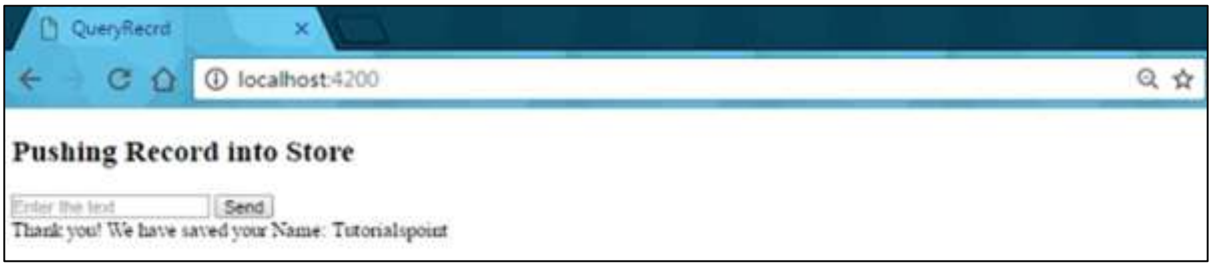
Run the ember server and you will get the input box to enter the value as shown in the screenshot below:



Pushing Record into Store

Tutorialspoint

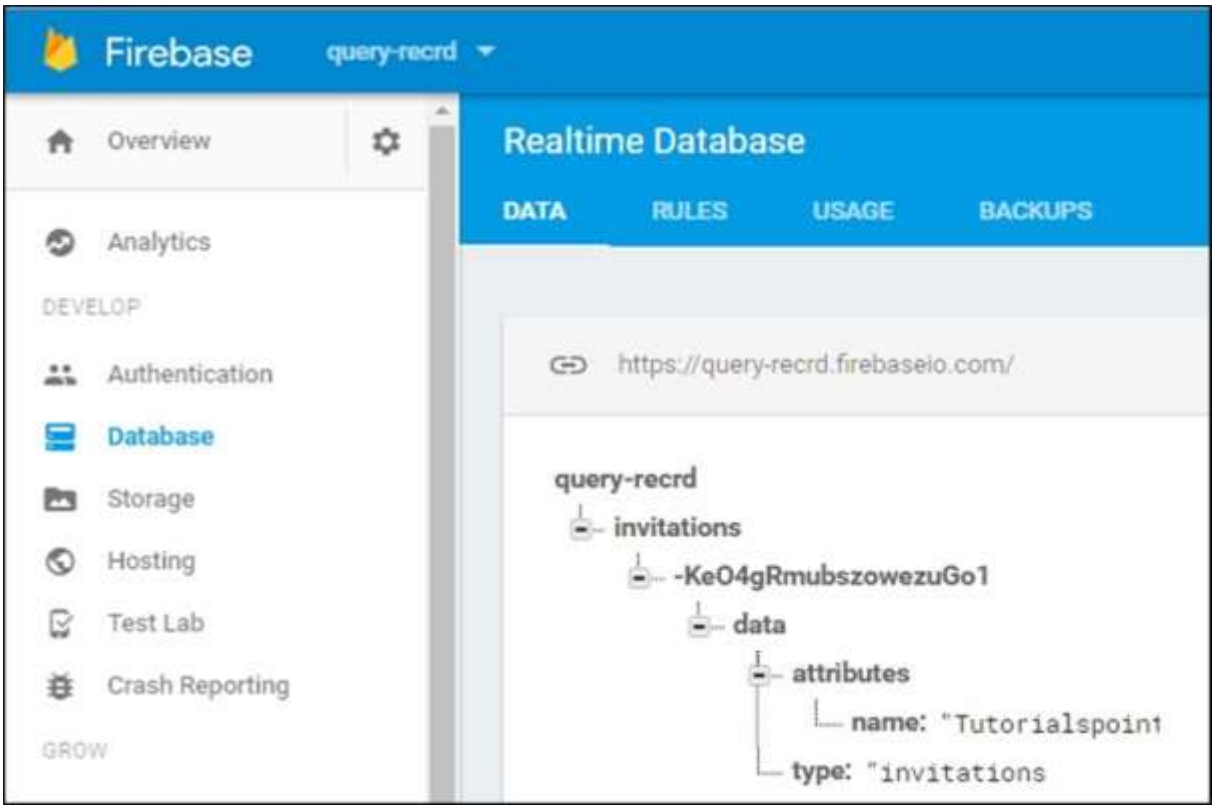
After clicking the send button, it will display the text entered by the user:



Pushing Record into Store

Thank you! We have saved your Name: Tutorialspoint

Now open your firebase database, you will see the stored value under the *Database* section:



```
query-recrd
 invitations
   -KeO4gRmubszowezuGo1
     data
       attributes
         name: "Tutorialspoint
       type: "invitations
```

EmberJS – Handling Metadata

Metadata is a data that is used for specific model or type instead of using record. The total number of records of the server will be stored in the metadata.

```
{
   "post": {
      "id": 1,
      "type": "type_name",
      "attributes": {
         "name": "group_name",
         "city": "city_name"
      }
      // ...
   },
   "meta": {
      "total": 100
   }
}
```

In the above code, *meta* represents the number of records in the store. The metadata can be accessed by using the following method:

```
store.query('post').then((myresult) => {
  let meta = myresult.get('meta');
})
```

The above process can be done by calling the *store.query()* method on the *myresult*. The total number of pages can be calculated by using *meta.total*.

EmberJS – Customizing Adapters

Ember.js Adapter specifies how data is kept at the backend data store such as the URL format and the REST API headers. The default Adapter of Ember includes some built-in assumptions for REST API. These assumptions help to build a web application much easier and better.

The Adapter can be created by using the following command:

```
ember generate adapter adapter-name
```

When you run the above command, it will display the following lines:

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  // code goes here
});
```

Ember data has the following built-in adapters:

- **DS.Adapter**: It is a basic adapter with no functionality in Ember.js.
- **DS.JSONAPIAdapter**: It is a default adapter that interfaces with HTTP server and follows JSON API conventions by transferring JSON via XHR.
- **DS.RESTAdapter**: It is used to communicate with HTTP server by using your store which transfers the JSON via XHR.

JSONAPIAdapter URL Conventions

The JSONAPIAdapter specifies URLs based on the model name.

For instance:

```
store.findRecord('mypost', 1).then(function(myfunc) {
});
```

The JSONAPIAdapter will send the GET request to */myposts/1*, if you ask for *MyPost* by ID. The following actions can be used on records in JSONAPIAdapter:

S.NO.	Action	HTTP Verb	URL
1	Find	GET	/myposts/123
2	Find All	GET	/myposts
3	Update	PATCH	/myposts/123
4	Create	POST	/myposts
5	Delete	DELETE	/myposts/123

Endpoint Path Customization

The endpoint path can be customized by using the *namespace* property with specific url namespace.

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  namespace: 'api/1'
});
```

If you request for the *myval* model, then it will display the url as *http://emberjs.com/api/1/myval/1*.

Host Customization

You can specify the new domain by using the *host* property on the adapter:

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  host: 'https://api.mysite.com'
});
```

If you request for the *myval* model, then it will display the url as *http://api.mysite.com/myval/1*.

Path Customization

The JSONAPIAdapter generates the path name by pluralizing and dasherizing the model name. You can override the *pathForType* method, if this behavior does not confirm to backend.

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  pathForType: function(type) {
    return Ember.String.underscore(type);
  }
});
```

Headers Customization

The headers can be customized by providing the key/value pairs on the JSONAPIAdapter's headers object and the Ember data will send key/value pair along with each ajax request.

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  headers: {
    'API_KEY': 'secret key',
    'ANOTHER_HEADER': 'header value'
  }
});
```

Authoring Adapters

The serializer can be specified by using the *defaultSerializer* adapter which is used only when specific serializer or *serializer:application* is not defined. It can be written as:

```
import DS from 'ember-data';


export default DS.JSONAPIAdapter.extend({
  defaultSerializer: '-default
});
```

10. EmberJS – Managing Dependencies

Ember uses NPM and Bower for managing dependencies which are defined in *package.json* for NPM and *bower.json* for Bower. For instance, you may require installing SASS for your style sheets which is not installed by Ember while developing Ember app. To accomplish this, use the *Ember Addons* for sharing the reusable libraries. If you want to install any CSS framework or JavaScript datepicker dependencies, then use the Bower package manager.

Addons

The *Ember CLI* can be used to install the Ember Addons by using the following command:

```
ember install ember-cli-sass
```

The *ember install* command will save all the dependencies to the respective configuration file.

Bower

It is a package manager for the web which manages the components of HTML, CSS, JavaScript or image files. It basically maintains and monitors all packages and examines new updates. It uses the configuration file *bower.json* to keep track of applications placed at the root of the Ember CLI project.

You can install the project dependencies by using the following command:

```
bower install <dependencies> --save
```

Assets

You can place the third-party JavaScript in the *vendor/* folder of your project which are not available as an Addon or Bower package and place the own assets such as robots.txt, favicon, etc. in the *public/* folder of your project. The dependencies which are not installed by Ember while developing the Ember app, should be included by using the manifest file *ember-cli-build.js*.

AMD JavaScript modules

You can give the asset path as the first argument and the list of modules and exports as the second argument. You can include these assets in the *ember-cli-build.js* manifest file as:

```
app.import('bower_components/ic-ajax/dist/named-amd/main.js', {
  exports: {
    'ic-ajax': [
      'default',
      'defineFixture',
      'lookupFixture',
      'raw',
      'request'
    ]
  }
});
```

Environment Specific Assets

The different assets can be used in different environments by defining object as first parameter which is an environment name and the value of an object should be used as asset in that environment. In the *ember-cli-build.js* manifest file, you can include as:

```
app.import({
  development: 'bower_components/ember/ember.js',
  production:  'bower_components/ember/ember.prod.js'
});
```

Other Assets

Once all the assets are placed in the *public/* folder, they will get copied into the *dist/* directory. For instance, if you copy a favicon placed at the *public/images/favicon.ico* folder, this will get copied into the *dist/images/favicon.ico* directory. The third-party assets can be added manually in the *vendor/* folder or by using the Bower package manager via the *import()* option. The assets which are not added by using the *import()* option, will not be present in the final build.

For instance, consider the following line of code which imports the assets into the *dist/* folder.

```
app.import('bower_components/font-awesome/fonts/fontawesome-webfont.ttf');
```

The above line of code creates a font file in *dist/font-awesome/fonts/fontawesome-webfont.ttf*. You can also place the above file at a different path as shown below:

```
app.import('bower_components/font-awesome/fonts/fontawesome-webfont.ttf', {
  destDir: 'assets'
});
```

It will copy the font file in *dist/assets/fontawesome-webfont.ttf*.

11. EmberJS – Application Concerns

The Ember application can be extended by using the *Ember.Application* class which declares and configures the objects that are helpful in building your application.

Application creates the *Ember.ApplicationInstance* class while running which is used for managing its aspects and it acts as owner for instantiated objects. In short, the *Ember.Application* class defines the application and the *Ember.ApplicationInstance* class manages its state.

The following table lists down more details about models:

S.NO.	Model Ways & Description
1	**Dependency Injection** It is a process of supplying dependencies of one object to another and used by an Ember application to declare and instantiates the objects and dependencies classes between them.
2	**Initializers** Initializers are used to configure an application as it boots.
3	**Services** Service is an Ember object which can be made available in the different parts of the application.
4	**The Run Loop** It is a region where most of the application's internal code takes place.

EmberJS – Dependency Injection

It is a process of supplying dependencies of one object to another and used by an Ember application to declare and instantiate the objects and dependency classes between them. The *Ember.Application* and *Ember.ApplicationInstance* classes play an important role in the Ember's Dependency Injection implementation.

The *Ember.Application* class declares and configures the objects and used as 'registry' for dependency declarations, where as the *Ember.ApplicationInstance* class acts as 'owner' for instantiated objects. However, the *Ember.Application* class acts as primary registry for an application and each *Ember.ApplicationInstance* class serves as registry.

Factory Registrations

A factory specifies an application part such as route, template, etc. and is registered with a particular key. For instance, index template is defined with *template:index* and application route is defined with *route:application*.

Registration key includes two parts; one is factory type and second is the name of factory and both segments split by colon(:). For instance, you can initialize the application by registering the *Logger* factory with the *logger:main* key.

```
application.register('mylog:logmsg', MyLogger);
```

Factory Injections

The factory can be injected, once it is registered. For instance, consider the following code:

```
application.inject('route', 'mylog', 'mylog:logmsg');
```

All the type of route factories will be represented with the *mylog* property and the value for this property will come from the *mylog:logmsg* factory. You can also inject on specific factory by using the full key as:

```
application.inject('route:index', 'mylog', 'mylog:logmsg');
```

Here only *mylog* property will get injected on the index route.

Factory Instance Lookups

You can use the factory instance's *lookup* method on an application instance to get an instantiated factory from the running application. It uses a string to determine the factory and returns an object.

For instance, you can call the *lookup* method on an application instance to fetch an instantiated factory as shown below:

```
applicationInstance.lookup('factory-type:factory-name');
```

Example

The example given below shows the use of factory registration, injection and instance lookups in the Ember application. Create a component with the name *dependency-inject* which will get define under *app/components/*. Open the *dependency-inject.js* file and add the following code:

```
import Ember from 'ember';
var inject = Ember.inject;

export default Ember.Component.extend({
   // load the service in the file /app/services/message.js
   message: inject.service(),
   message: 'Click the above button to change text!!!',
   actions: {
      pressMe: function () {
         // after clicking button, above message will get display at console
```

```
          var testText = this.get('start').thisistest();
          this.set('message', testText);
          // after clicking button, it will enter in the component page
          this.get('logger').log('Entered in component!');
       },
       scheduleTasks: function () {
          // scheduling work on specific queues like "sync" or "afterRender"
          Ember.run.schedule('afterRender', this, function () {
             console.log("CUSTOM: I'm in afterRender");
             Ember.run.schedule('sync', this, function () {
                console.log("CUSTOM: I'm back in sync");
             });
          });
       }
    }
});
```

Now open the component template file *app/templates/components/dependency-inject.hbs* and enter the following code:

```
<button {{action "pressMe"}}>Click Here</button><br>
<h2>{{message}}</h2>
<button {{action "scheduleTasks"}}>Schedule Tasks!</button>

{{yield}}
```

Open the *application.hbs* file and add the following line of code:

```
{{dependency-inject}}
{{outlet}}
```

We need to create an initializer to configure an application by using the following command:

```
ember generate initializer init
```

Open the *init.js* file which is created under *app/initializers/* and add the following code:

```
export function initialize(app) {
   // injecting the 'start' property into the component
   app.inject('component', 'start', 'service:message');
}

export default {
   //initializer name
   name: 'init',
   initialize: initialize
};
```

Create a service which can be made available in the different parts of the application. Use the following command to create the service:

```
ember generate service message
```

Now open the *message.js* service file which is created under *app/services/* with the following code:

```
import Ember from 'ember';

export default Ember.Service.extend({
   isAuthenticated: true,
   // after clicking the button, 'thisistest()' triggers and display the following text
   thisistest: function () {
      return "Welcome to Tutorialspoint!!!";
   }
});
```

Next, create an initializer which configures the application as it boots. The initializer can be created by using the following command:

```
ember generate initializer logger
```

Open the *logger.js* initializer file which is created under *app/initializers/* with the following code:

```
import Ember from 'ember';

// it is an application initializer that run as your application boots
export function initialize(application) {
   var Logger = Ember.Object.extend({
      log(m) {
         console.log(m);
      }
   });

   // Registration key includes two parts; one is factory type and second is
name of factory
   application.register('logger:main', Logger);

   // Once a factory is registered, it can be injected by using
'application.inject' along with 'logger' property
   // and value for this property will come from 'logger:main'factory
   application.inject('component:dependency-inject', 'logger', 'logger:main');
}

export default {
   name: 'logger',
   initialize: initialize
};
```

Next, create the instance initializer for an application by using the following command:

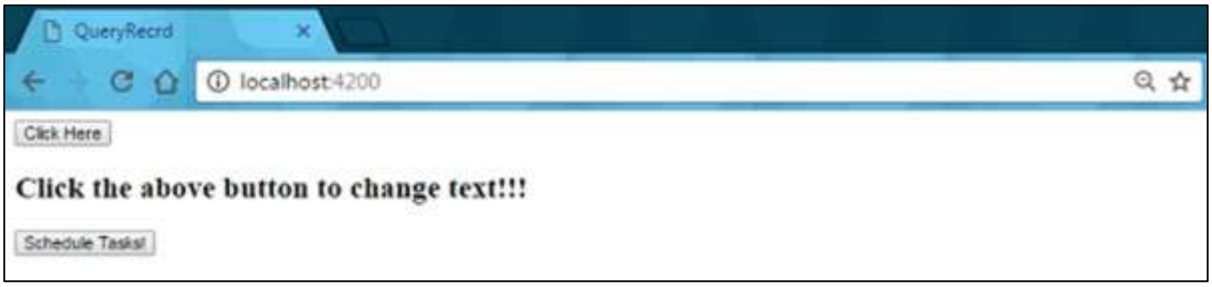
```
ember generate instance-initializer logger
```

Open the *logger.js* initializer file which is created under *app/instance-initializers/* with the following code:

```
// Application instance initializers run as an application instance is loaded
export function initialize(applicationInstance) {
   var logger = applicationInstance.lookup('logger:main');
   // it indicates that instance has booted at console log
   logger.log('Hello...This message is from an instance-initializer!');
}
export default {
   // it is an instance initializer name
   name: 'logger',
   initialize: initialize
};
```

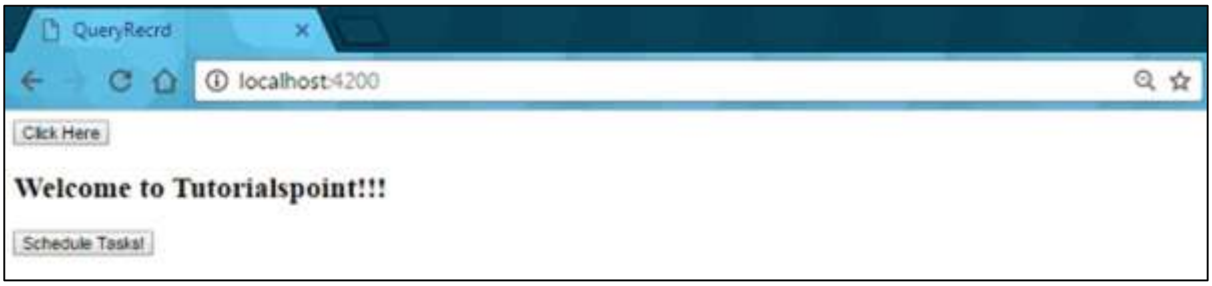
Output

Run the ember server; you will receive the following output:



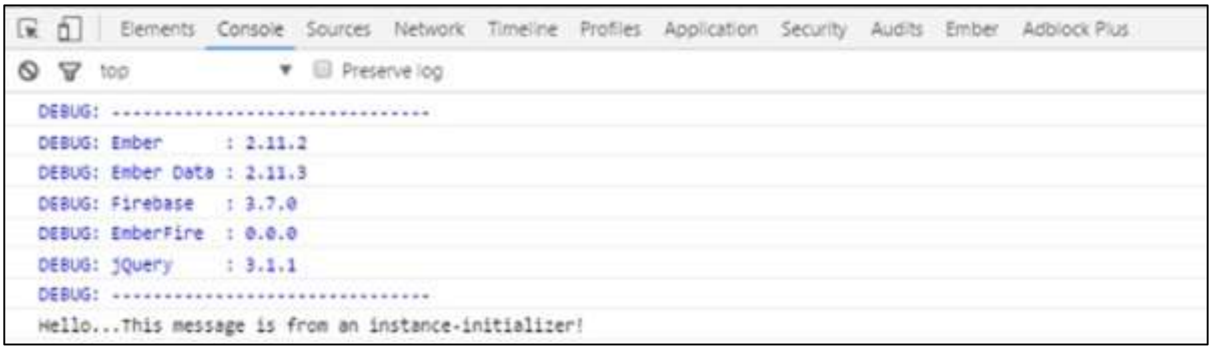
Click the above button to change text!!!

Next, click on the *Click Here* button, it will display the text from the service page as shown in the screenshot below:



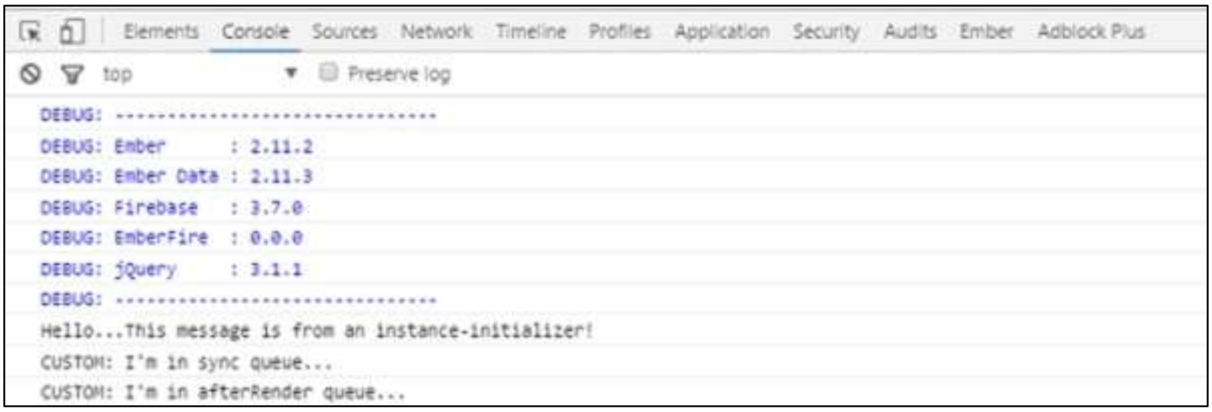
Welcome to Tutorialspoint!!!

Now go to console and check for log messages which are displayed from the instance initializers after the text is displayed as in the abve screenshot:



```
DEBUG: -------------------------------
DEBUG: Ember      : 2.11.2
DEBUG: Ember Data : 2.11.3
DEBUG: Firebase   : 3.7.0
DEBUG: EmberFire  : 0.0.0
DEBUG: jQuery     : 3.1.1
DEBUG: -------------------------------
Hello...This message is from an instance-initializer!
```

Next, click on the *schedule Tasks* button to schedule the work on queues that are processed in the priority order:



```
DEBUG: -------------------------------
DEBUG: Ember      : 2.11.2
DEBUG: Ember Data : 2.11.3
DEBUG: Firebase   : 3.7.0
DEBUG: EmberFire  : 0.0.0
DEBUG: jQuery     : 3.1.1
DEBUG: -------------------------------
Hello...This message is from an instance-initializer!
CUSTOM: I'm in sync queue...
CUSTOM: I'm in afterRender queue...
```

EmberJS – Initializers

Initializers are used to configure an application as it boots. Initializers contain two types:

- **Application Initializers**: An application initializer runs as your application boots and configures the dependency injection in your application.
- **Application Instance Initializers**: An application instance initializers run when an application instance is loaded and configures the initial state of an application.

Application Initializers

Application initializers can be created by using the following command:

```
ember generate initializer initializer-name
```

When you create an initializer, it will display the following code format:

```
export function initialize(/* application */) {
  //application.inject('route', 'foo', 'service:foo');
}


export default {
  //'logger' is an application initializer name
  name: 'logger',
  initialize
};
```

Application Instance Initializers

An instance initializer for an application can be created by using the following command:

```
ember generate instance-initializer instance-initializer-name
```

When you run the above command, it will display the the following code structure:

```
export function initialize(/* appInstance */) {
  // appInstance.inject('route', 'foo', 'service:foo');
}
export default {
   //'logger' is an application instance initializer name
  name: 'logger',
  initialize
};
```

For more about these two initializers along with an example, see this link – https://www.tutorialspoint.com/emberjs/app_concrn_dependency_injection.htm.

EmberJS – Services

Service is an Ember object which can be made available in the different parts of the application.

Services can be used in different kinds of areas:

- Geolocation
- Third-party APIs
- Events or notifications sent by server
- User or session authentication
- Logging
- WebSockets

You can create the service by using the following command:

```
ember generate service service-name
```

When you run the above command, it will display the following code format:

```
import Ember from 'ember';
// services must extend the Ember.Service base class
export default Ember.Service.extend({
});
```

For more about services along with an example, see this link – https://www.tutorialspoint.com/emberjs/app_concrn_dependency_injection.htm.

EmberJS – Run Loop

It is an area where most of the application's internal code takes place. This is used to batch and is a way of ordering or re-ordering the work to check if it is effective and efficient. It schedules the work based on specific queues to complete the work in priority order.

Integrating the run loop with non-Ember API leads to some asynchronous callback. For instance:

- setTimeout and setInterval callbacks
- AJAX callbacks
- postMessage and messageChannel event handlers
- Websocket callbacks
- DOM update and event callbacks

The run loop works in Ember based on queues specified priority-wise:

```
Ember.run.queues
=> ["sync", "actions", "routerTransitions", "render", "afterRender", "destroy"]
```

- **sync**: It is a higher priority queue that includes binding synchronization jobs.
- **actions**: It is a general work queue that includes scheduled tasks.
- **routerTransitions**: It specifies the transition jobs in router.
- **render**: It is used for rendering the jobs which update the DOM.
- **afterRender**: It runs the jobs after completing the scheduled tasks.
- **destroy**: It is a lower priority queue that terminates the jobs which are scheduled to destroy.

Execution of Jobs based on Queues

Follow these steps for the execution of Jobs based on Queues:

Step1: In this step, pending jobs of highest priority queue will be checked in *CURRENT_QUEUE*. The run loop will be completed, if there are no pending jobs.

Step 2: Specify the new temporary queue as *WORK_QUEUE*.

Step 3: Transfer the jobs from *CURRENT_QUEUE* to *WORK_QUEUE*.

Step 4: Successively process the jobs in *WORK_QUEUE*.

Step 5: Repeat from Step 1.

Behavior of Run Loop when Testing

If we try to schedule the work without run loop, then Ember will throw an error when the application is in testing mode. Consider the following reasons to understand why Autoruns are disabled:

- If you fail to open the run loop before scheduling callbacks on it, then Autoruns will not make any mistake in the production.
- Disabling the autoruns identifies the incorrect test failures which occur when an application runs outside a run loop and helps in the testing of your application.

For more about these run loop along with an example, see this link – https://www.tutorialspoint.com/emberjs/app_concrn_dependency_injection.htm.

12. EmberJS – Configuring Ember.js

The Ember.js can be configured for managing the application's environment. The configuring Ember.js includes the following topics:

S.NO.	Configuring Ways & Description
1	**Configuring App and Ember CLI** You can configure the Ember App and CLI for managing the application's environment.
2	**Disabling Prototype Extensions and Specifying URL Type** The prototype extensions can be disabled by setting the *EXTEND_PROTOTYPES* flag to false and specifying the URL type by using the Ember router options.
3	**Embedding Applications and Feature Flags** You can Embed an application into an existing page by changing the root element and feature flags can be enabled based on the project's configuration.

EmberJS – Configuring App and Ember CLI

You can configure the Ember App and CLI for managing the application's environment. The environment config file will be present at *config/environment.js*. It contains the following code structure:

```
module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'query-params', //it is the name of application
    environment: environment,
    baseURL: '/',
    locationType: 'auto',
    EmberENV: {
      FEATURES: {
        // Here you can enable experimental features on an ember canary build
        // e.g. 'with-controller': true
      }
    },

    APP: {
      // Here you can pass flags/options to your application instance
      // when it is created
      API_HOST: 'http://localhost:3000'
    }
  };

  if (environment === 'development') {
    //code here
  }

  if (environment === 'test') {
    //code here
  }

  if (environment === 'production') {

  }
  return ENV;
};
```

The *ENV* object includes the following three keys:

- **EmberENV**: It provides the Ember feature flags.
- **APP**: It is used to pass flags/options to your application instance.
- **environment** : It provides the current environment names such as *development*, *production* and *test*.

Configuring Ember CLI

You can configure the Ember CLI by adding the configurations to the *.ember-cli* file which is present in your application's root.

For instance, you can pass the port number by using the command *ember server --port 8080* from the command line. This configuration can be added in the *.ember-cli* file as shown below:

```
{
   "port" : 8080
}
```

EmberJS – Disabling Prototype Extensions and Specifying URL Type

The prototype extensions can be disabled by setting the *EXTEND_PROTOTYPES* flag to false. Open the *config/environment.js* file and set the flag in the ENV object:

```
ENV = {
  EmberENV: {
    EXTEND_PROTOTYPES: false
  }
}
```

The prototypes of JavaScript objects can be extended by Ember.js in the following ways:

- **Array**: It is used to implement the *Ember.Enumerable*, *Ember.MutableEnumerable*, *Ember.MutableArray* and *Ember.Array* interfaces.
- **String**: It adds some string helper methods such as *camelize()* (specifies the lowerCamelCase form), *w()* (divides string into separate units), etc.
- **Function**: It is used to explain the functions as computed properties by using the *property()* method.

The above prototype extensions can be used in the application's configuration file as shown below:

```
ENV = {
  EmberENV: {
    EXTEND_PROTOTYPES: {
      String: false,
      Array: true
    }
  }
}
```

Specifying URL Type

The application's URL type can be specified by using Ember router's four options:

- **history**
- **hash**
- **none**
- **auto**

History

This option uses HTML5 browser's API to create the URLs. For instance, create a router called *myroute1* under another router *myroute* which will navigate to the *myroute.myroute1* route.

```
Router.map(function() {
  this.route('myroute', function() {
    this.route('myroute1');
  });
});
```

The above code is created under the *app/router.js* file to define the URL mappings that takes parameter as an object to create the route.

Hash

This option specifies the starting state of an application by using the anchor based URL's which will put in sync as we move around. For instance, the above route path */#/myroute/myroute1* will navigate to the *myroute.myroute1* route.

None

This option does not update the URL and set the *ENV.locationType* flag to *none* to disable the location API which does not allow browser's URL to interact with your application.

EmberJS – Embedding Applications and Feature Flags

You can embed an application into an existing page by changing the root element. When you create an application, by default the application template will be rendered by the application and attached to the body element. It is possible to include the application template to different element by using the *rootElement* property.

```
import Ember from 'ember';


export default Ember.Application.extend({
  rootElement: '#app'
});
```

The URL can be disabled by setting the router's *locationType* flag to *none*. This property can be added in the *config/environment.js* file.

```
let ENV = {
  locationType: 'none'
};
```

The root URL can be specified in the Ember application, if it is served from the same domain. You also need to specify what the root URL of your Ember application is.

For instance, you can include the blogging application from *http://emberjs.com/myblog/*, and specify the root URL of *myblog*. This can be done by using the *rootURL* property router:

```
Ember.Router.extend({
  rootURL: '/myblog/'
});
```

Feature Flags

The flagging details of feature flags will be specified in the *features.json* file. The code of feature flags can be enabled based on the project's configuration. The newly developed feature flag is available only in canary builds. It can be enabled by using the project's configuration file when the Ember.js community considers that it is ready for production use.

A feature can have any of the following three flags:

- **true**: It specifies that the flag is present and enabled; the code must be enabled in the generated build.
- **null**: It specifies that the flag is present, but disabled in the build output and can be enabled at runtime.
- **false**: It specifies that the flag disabled and the code is not available in the generated build.

Developers include the entry of new feature in the *FEATURES.md* file along with the explanation of feature. They also add a new feature to the master branch on the github.

The feature can be enabled at run time by setting the *link-to* flag value to *true* before application boots. Open the *config/environment.js* file and set the flag as shown below:

```
let ENV = {
  EmberENV: {
    FEATURES: {
      'link-to': true
    }
  }
};
```

13. EmberJS – Ember Inspector

Ember inspector is a browser add-on which is used to debug the Ember applications. The Ember inspector includes the following topics:

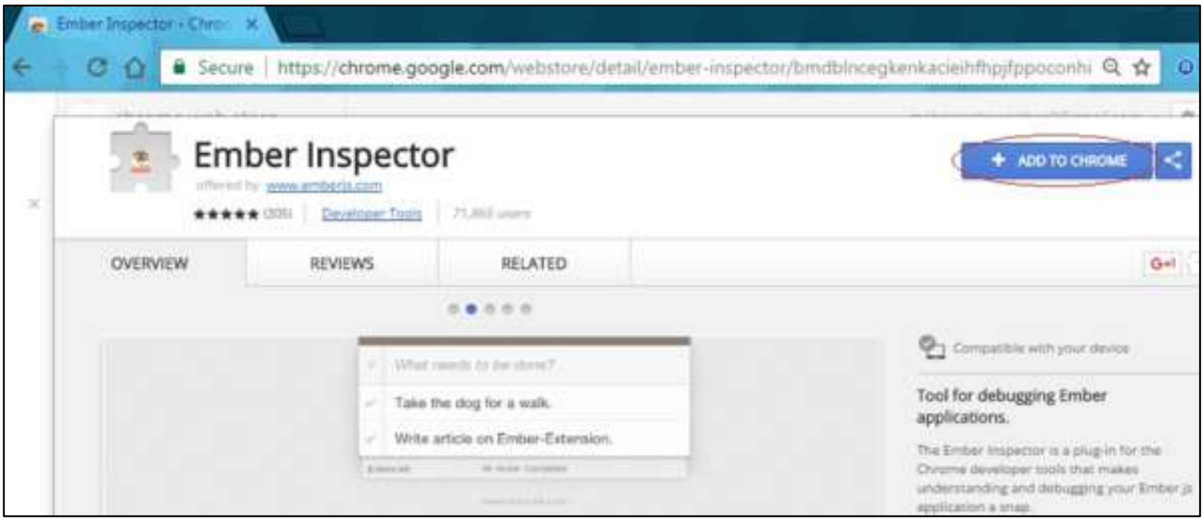
S.NO.	Ember inspector Ways & Description
1	**Installing the Inspector** You can install the Ember inspector to debug your application.
2	**Object Inspector** The Ember inspector allows interacting with the Ember objects.
3	**The View Tree** The view tree provides the current state of an application.
4	**Inspecting Routes, Data Tab and Library Info** You can see the list of application's routes defined by the inspector and the Data tab is used to display the list of model types.
5	**Debugging Promises** Ember inspector provides promises based on their states.
6	**Inspecting Objects and Rendering Performance** Use the Container for inspecting the object instances and compute the application's render time by using the Render Performance option.

EmberJS – Installing the Inspector

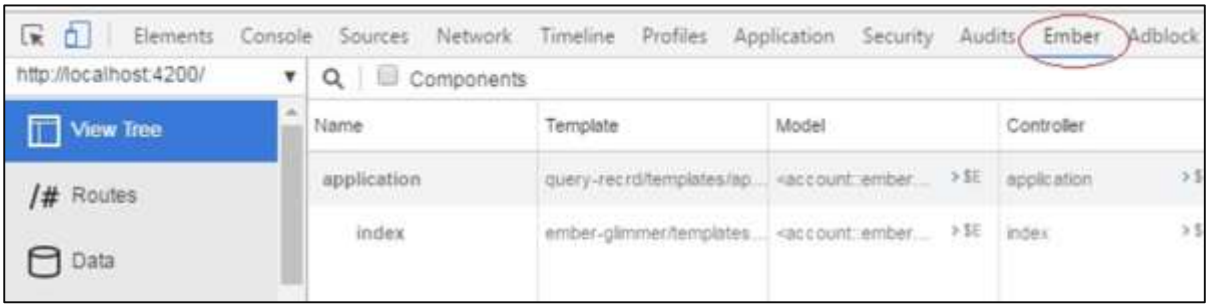
Ember inspector is a browser add-on, which is used to debug your Ember applications. Follow the steps given below to install it on Google Chrome, Firefox browsers:

Google Chrome

Install the Ember inspector on Google Chrome by visiting the Chrome Web Store page.

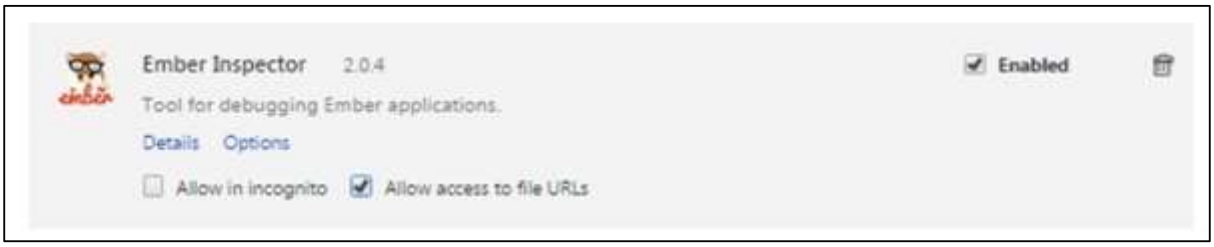

Ember Inspector - Chro
Secure
https://chrome.google.com/webstore/detail/ember-inspector/bmdblncegkenkacieihfhpjfppoconhi
Ember Inspector
offered by www.emberjs.com
★★★★★ (305)
Developer Tools
71,463 users
+ ADD TO CHROME
OVERVIEW
REVIEWS
RELATED
G+1
What needs to be done?
Take the dog for a walk.
Write article on Ember-Extension.
Compatible with your device
Tool for debugging Ember applications.
The Ember Inspector is a plug-in for the Chrome developer tools that makes understanding and debugging your Ember.js application a snap.

Click on the *Add To Chrome* option. After installing, open your Ember application, press *F12* key and click on the *Ember* tab at the right-hand side.



Name	Template	Model	Controller
application	query-recrd/templates/ap...	<account::ember...	application
index	ember-glimmer/templates...	<account::ember...	index

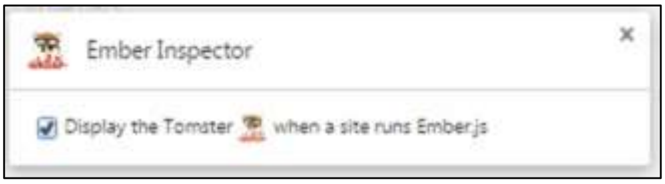
If you want to use the Ember inspector with the *file://* protocol, then go to *chrome://extensions* in Chrome and check the *Allow access to file URLs* option as shown below:



☑ Enabled

☐ Allow in incognito ☑ Allow access to file URLs

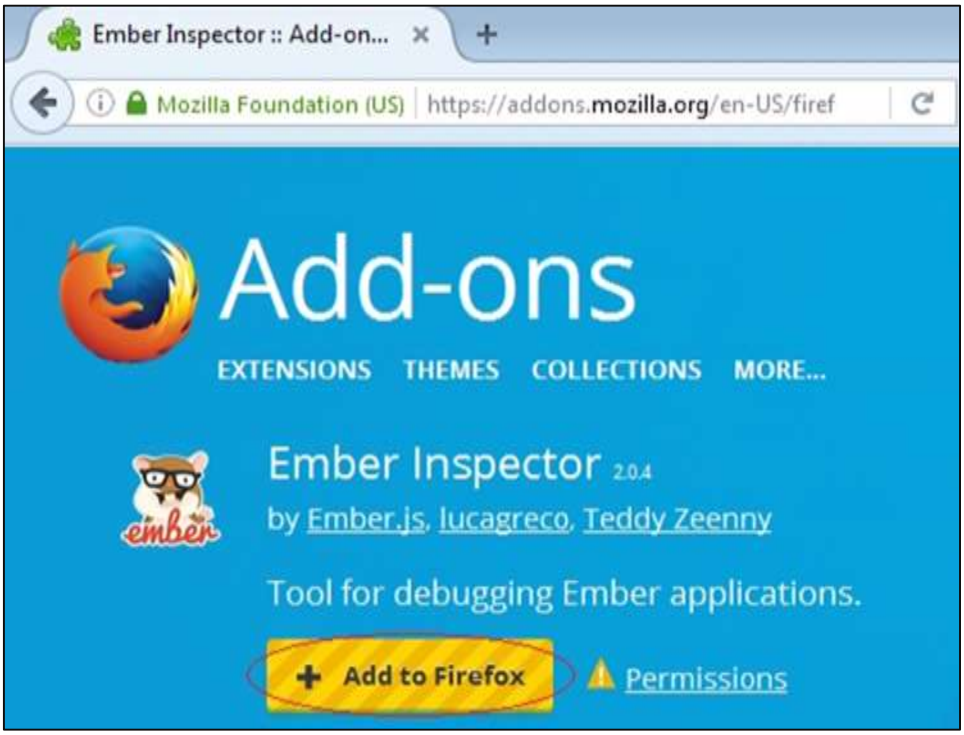
Next, click on the *options* and check the *Display the Tomster* option to show the Tomster icon in the URL bar.



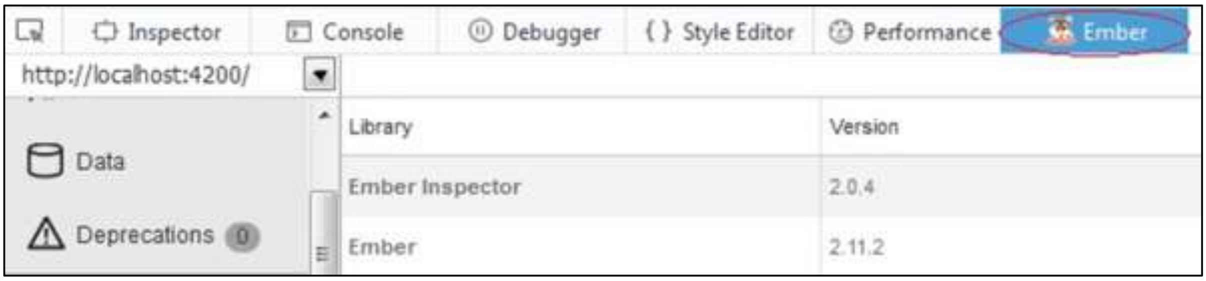
☑ Display the Tomster when a site runs Ember.js

Mozilla Firefox

To install the Ember inspector on Mozilla Firefox, go to the add-on page on the Mozilla Add-ons site.

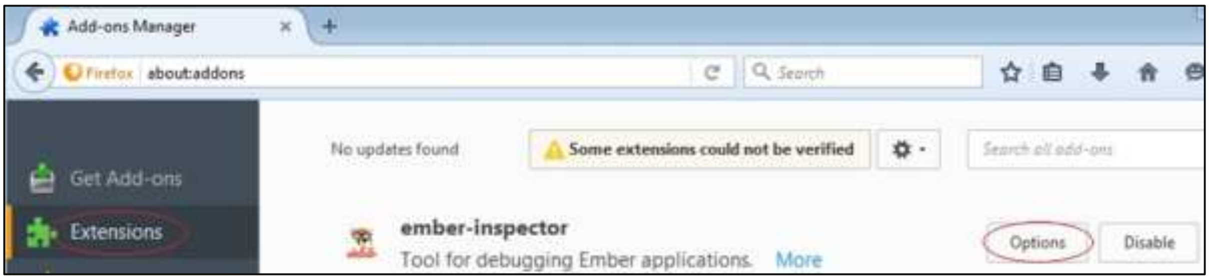

Ember Inspector :: Add-on...
Mozilla Foundation (US)
https://addons.mozilla.org/en-US/firef
Add-ons
EXTENSIONS
THEMES
COLLECTIONS
MORE...
Ember Inspector 2.0.4
by Ember.js, lucagreco, Teddy Zeenny
Tool for debugging Ember applications.
Add to Firefox
Permissions

Click on the *Add To Firefox* option. After installing, open your Ember application, press *F12* key and click on the *Ember* tab at the right-hand side.

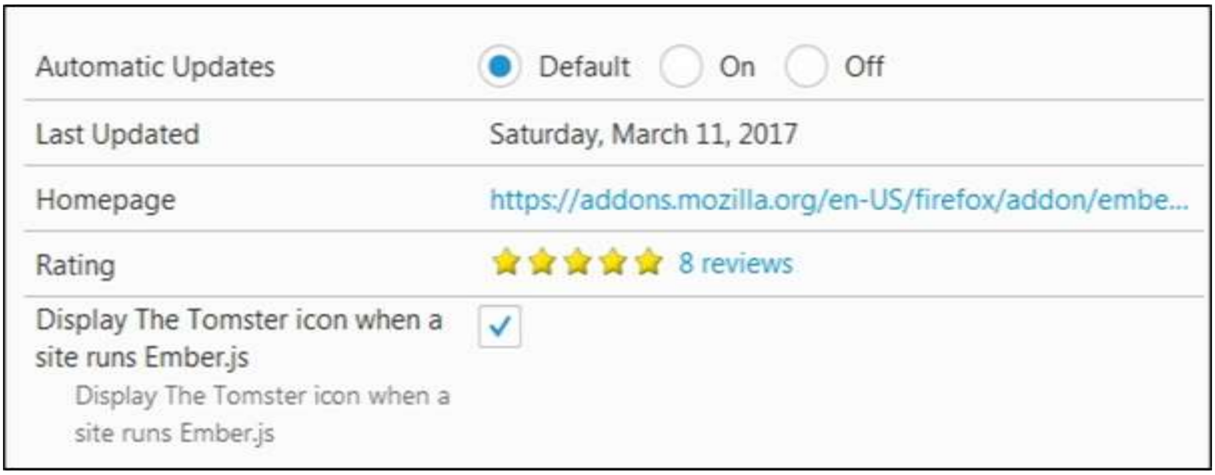


Library	Version
Ember Inspector	2.0.4
Ember	2.11.2

Next, you need to check the *Display the Tomster icon when a site runs Ember.js* option to show the Tomster icon in the URL bar. Type *about:addons* in the URL bar, go to the *Extensions* option and click on the *options* button:



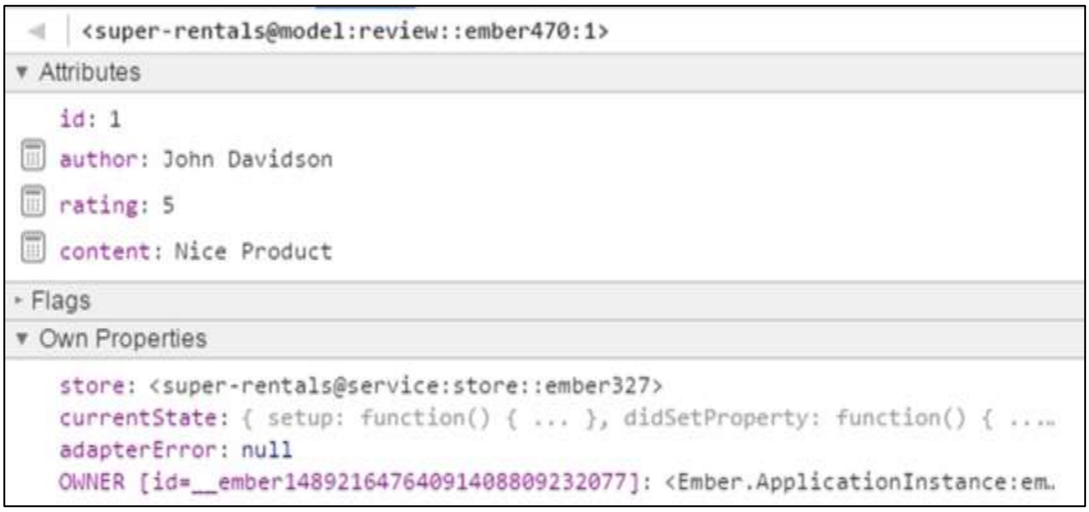
Scroll down and check the *Display the Tomster* option:



Automatic Updates	● Default ○ On ○ Off
Last Updated	Saturday, March 11, 2017
Homepage	https://addons.mozilla.org/en-US/firefox/addon/embe...
Rating	★★★★★ 8 reviews
Display The Tomster icon when a site runs Ember.js Display The Tomster icon when a site runs Ember.js	☑

EmberJS – Object Inspector

The Ember inspector allows interacting with the Ember objects. To view the object's properties, click on the Ember object:

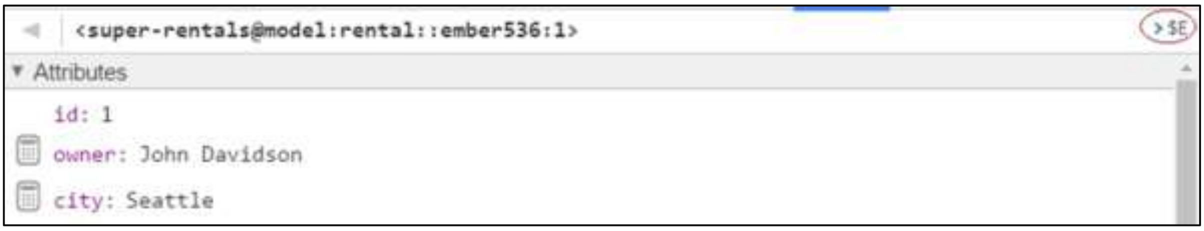


```
<super-rentals@model:review::ember470:1>
▾ Attributes
  id: 1
  author: John Davidson
  rating: 5
  content: Nice Product
▸ Flags
▾ Own Properties
  store: <super-rentals@service:store::ember327>
  currentState: { setup: function() { ... }, didSetProperty: function() { ....
  adapterError: null
  OWNER [id=__ember14892164764091408809232077]: <Ember.ApplicationInstance:em.
```

The objects can be viewed under parent objects along with the inherited properties. If there is a calculator icon along with the property name, then it is called the computed property.

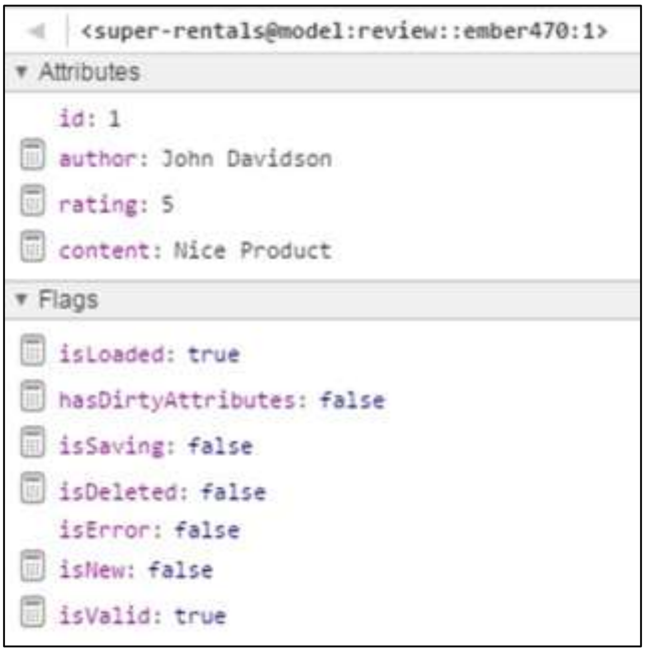
Send Objects to Console

The Ember objects can be sent to the console by clicking the *$E* button which specifies the global variable to the chosen object.



```
<super-rentals@model:rental::ember536:1>                    >$E
▾ Attributes
  id: 1
  owner: John Davidson
  city: Seattle
```

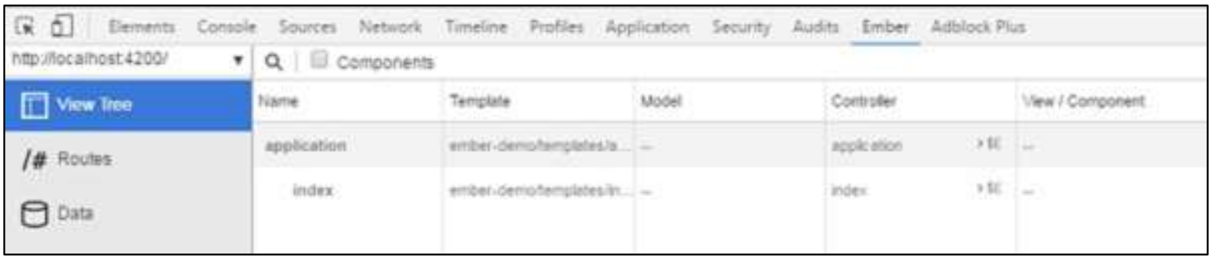
You can see some custom group properties such as *Attributes*, *Flags*, etc. by inspecting the Ember data model.



```
<super-rentals@model:review::ember470:1>
▾ Attributes
  id: 1
  author: John Davidson
  rating: 5
  content: Nice Product
▾ Flags
  isLoaded: true
  hasDirtyAttributes: false
  isSaving: false
  isDeleted: false
  isError: false
  isNew: false
  isValid: true
```

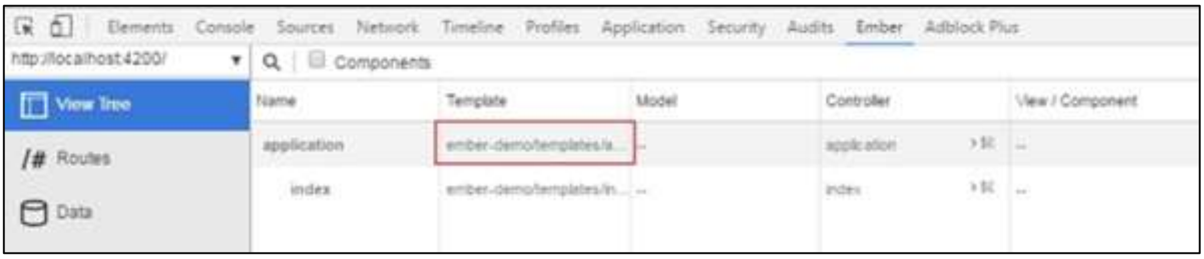
EmberJS – View Tree

The view tree provides current state of an application which includes currently rendered template, model, controller and components in a tree format.



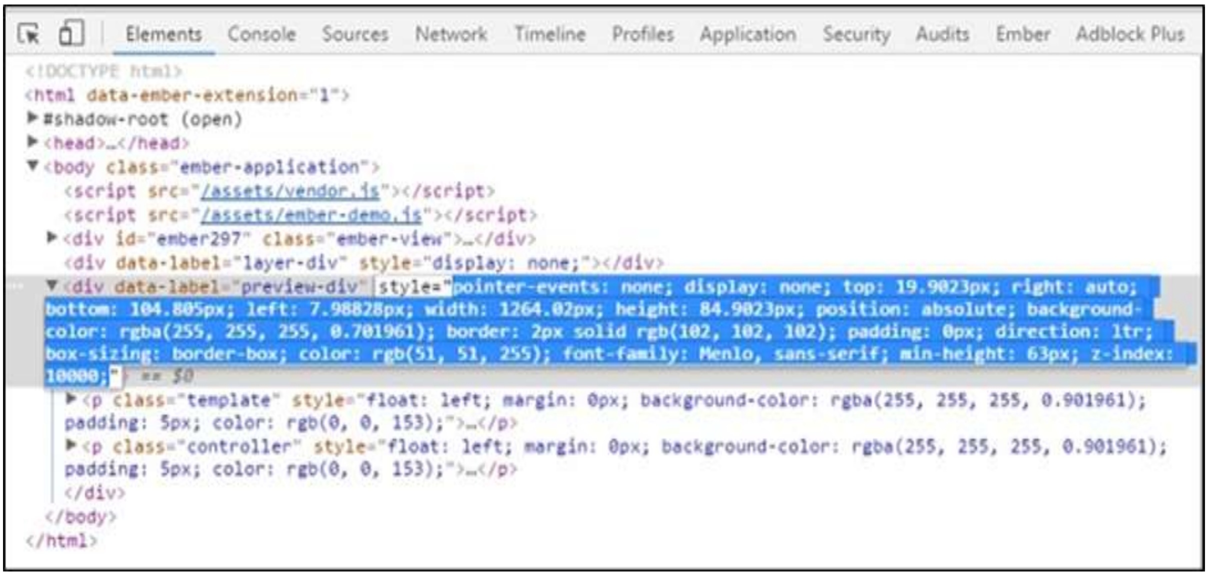
Name	Template	Model	Controller	View / Component
application	ember-demo/templates/a...	--	application	--
index	ember-demo/templates/in...	--	index	--

You can inspect the template to see how it is rendered by Ember application:



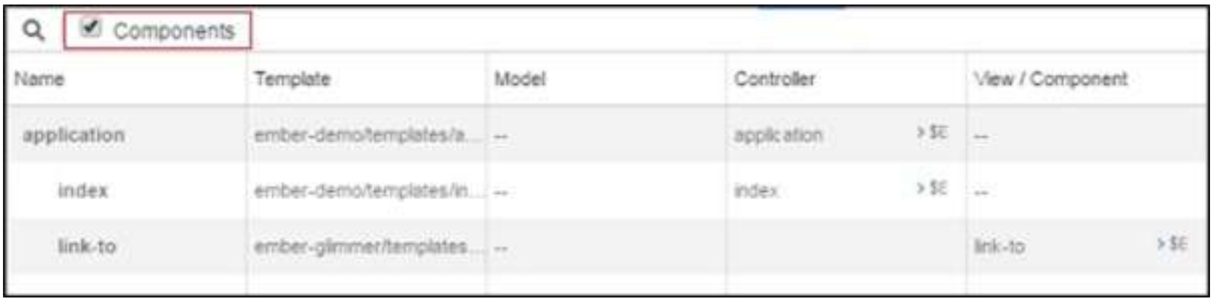
Name	Template	Model	Controller	View / Component
application	ember-demo/templates/a...	--	application	--
index	ember-demo/templates/in...	--	index	--

Click on the template name in the view tree and you will see the selected DOM elements under the *Elements* panel:



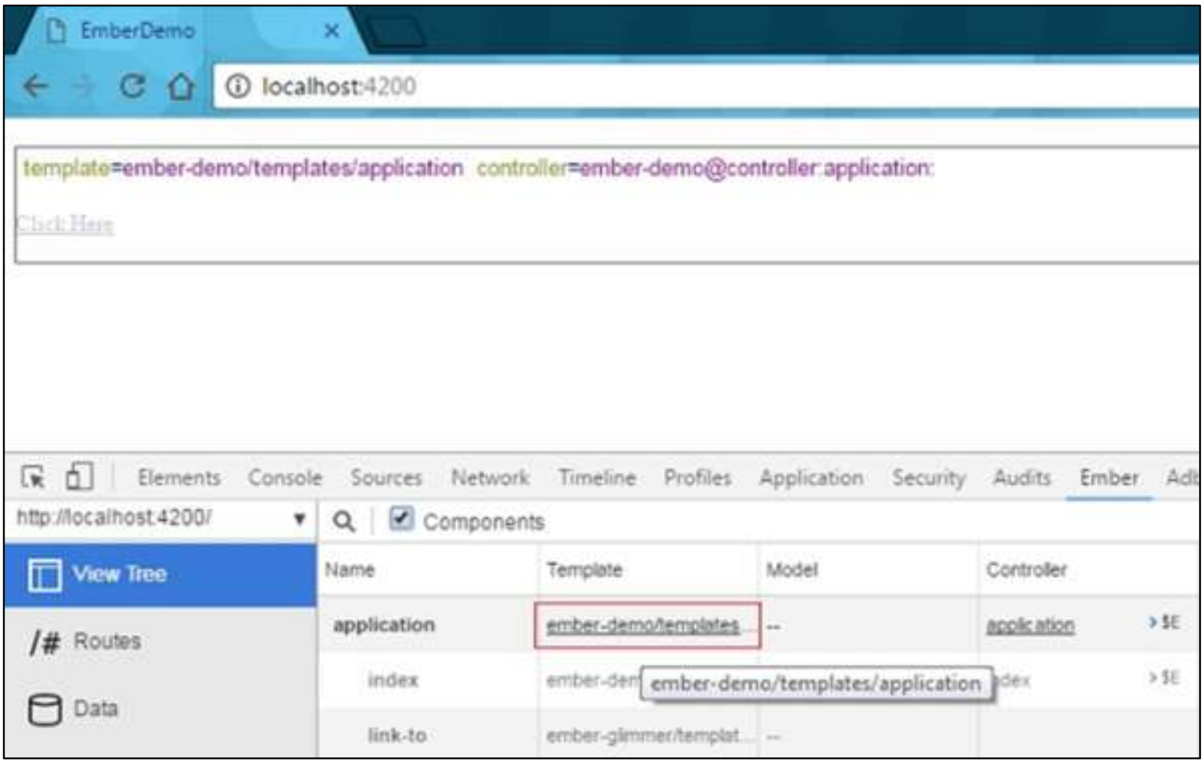
```
<!DOCTYPE html>
<html data-ember-extension="1">
▶#shadow-root (open)
▶<head>…</head>
▼<body class="ember-application">
    <script src="/assets/vendor.js"></script>
    <script src="/assets/ember-demo.js"></script>
  ▶<div id="ember297" class="ember-view">…</div>
    <div data-label="layer-div" style="display: none;"></div>
  ▼<div data-label="preview-div" style="pointer-events: none; display: none; top: 19.9023px; right: auto;
  bottom: 104.805px; left: 7.98828px; width: 1264.02px; height: 84.9023px; position: absolute; background-
  color: rgba(255, 255, 255, 0.701961); border: 2px solid rgb(102, 102, 102); padding: 0px; direction: ltr;
  box-sizing: border-box; color: rgb(51, 51, 255); font-family: Menlo, sans-serif; min-height: 63px; z-index:
  10000;"> == $0
    ▶<p class="template" style="float: left; margin: 0px; background-color: rgba(255, 255, 255, 0.901961);
    padding: 5px; color: rgb(0, 0, 153);">…</p>
    ▶<p class="controller" style="float: left; margin: 0px; background-color: rgba(255, 255, 255, 0.901961);
    padding: 5px; color: rgb(0, 0, 153);">…</p>
    </div>
  </body>
</html>
```

By default, the view tree disregards the components and can be loaded into the view tree by checking the *Components* checkbox.



Name	Template	Model	Controller	View / Component
application	ember-demo/templates/a...	--	application >$E	--
index	ember-demo/templates/in...	--	index >$E	--
link-to	ember-glimmer/templates...	--		link-to >$E

You can see the highlighted template name and its associated objects when you hover the items in the view tree.



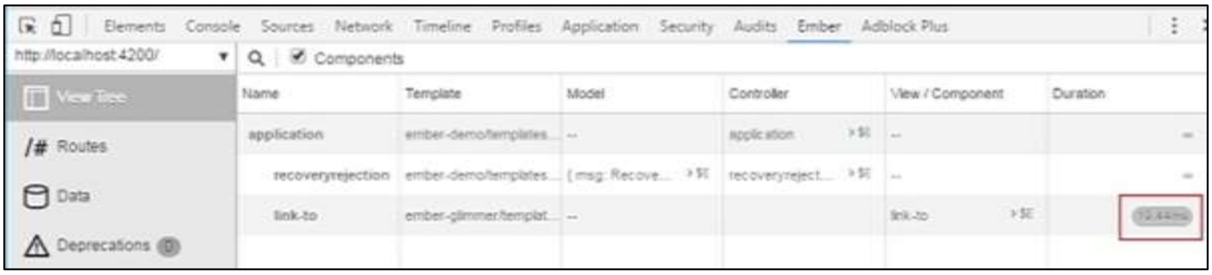
template=ember-demo/templates/application controller=ember-demo@controller:application:

Click Here

Name	Template	Model	Controller
application	ember-demo/templates...	--	application
index	ember-dem ember-demo/templates/application		index
link-to	ember-glimmer/templat...	--	

In the above image, we have hovered on an item which is rounded with red color and related template is highlighted with the other objects in your application.

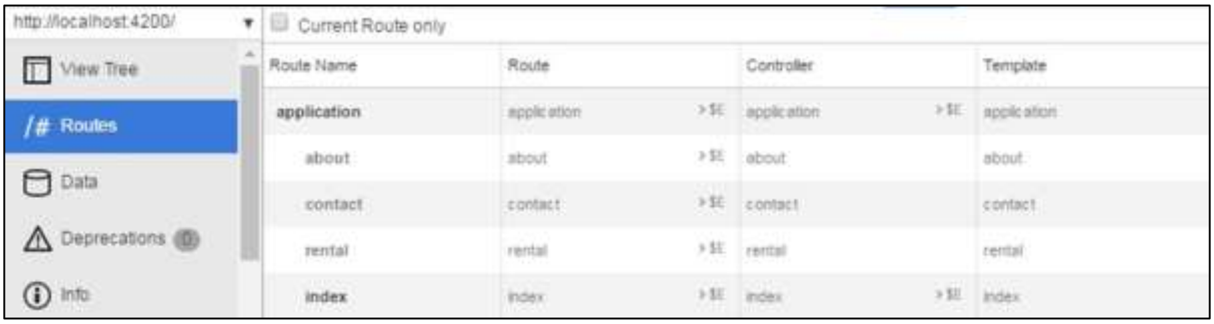
You can see the rendering duration for the given template under the *Duration* section in the view tree.



Name	Template	Model	Controller	View / Component	Duration
application	ember-demo/templates...	--	application	--	--
recoveryrejection	ember-demo/templates...	{ msg: Recove...	recoveryreject...	--	--
link-to	ember-glimmer/templat...	--		link-to	[illegible]

EmberJS – Inspecting Routes, Data Tab and Library Info

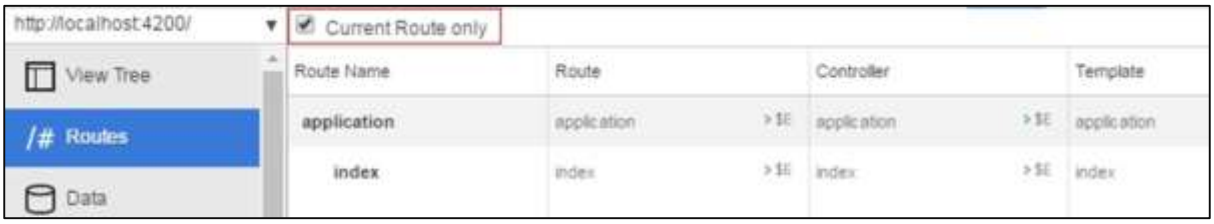
You can see the list of application's routes defined by the inspector.



Route Name	Route	Controller	Template
application	application	application	application
about	about	about	about
contact	contact	contact	contact
rental	rental	rental	rental
index	index	index	index

When you click on the */# Routes* tab, it will show the list of application's routes.

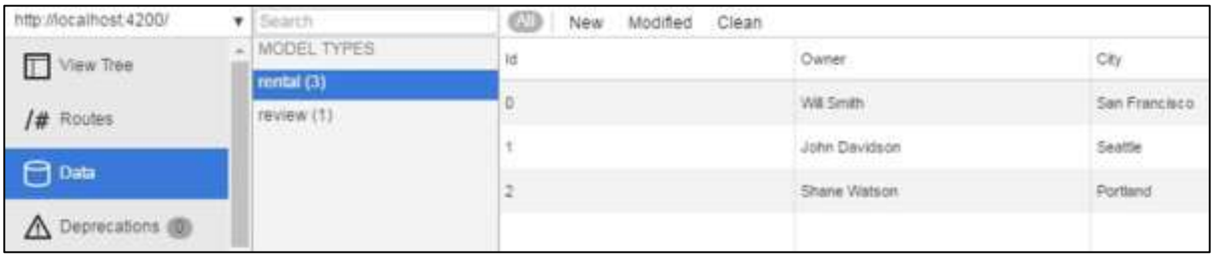
Ember inspector can display only the current active routes. This can be done by checking the *Current Route Only* checkbox to view the current routes.



Route Name	Route	Controller	Template
application	application	application	application
index	index	index	index

Data Tab

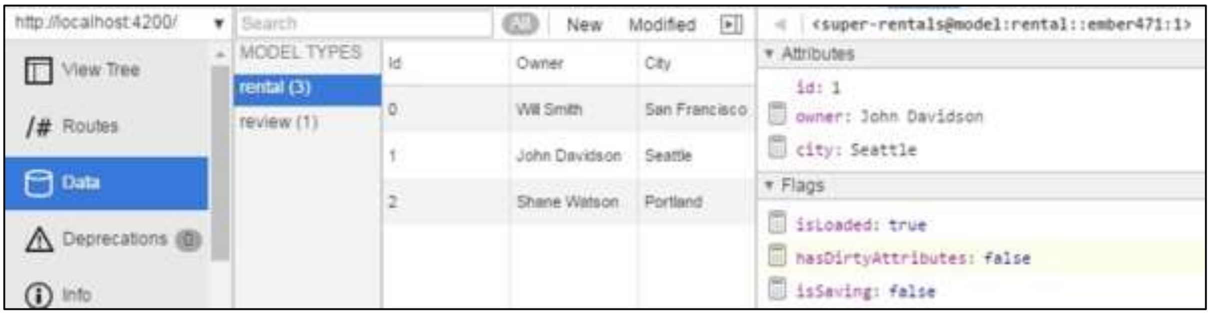
Ember inspector uses the Data tab to display the list of model types defined in your application. The Ember inspector will display the related records when you click on the model type.



Id	Owner	City
0	Will Smith	San Francisco
1	John Davidson	Seattle
2	Shane Watson	Portland

Inspecting Records

You can inspect the records of each model type by clicking on the record which will eventually display all attributes.

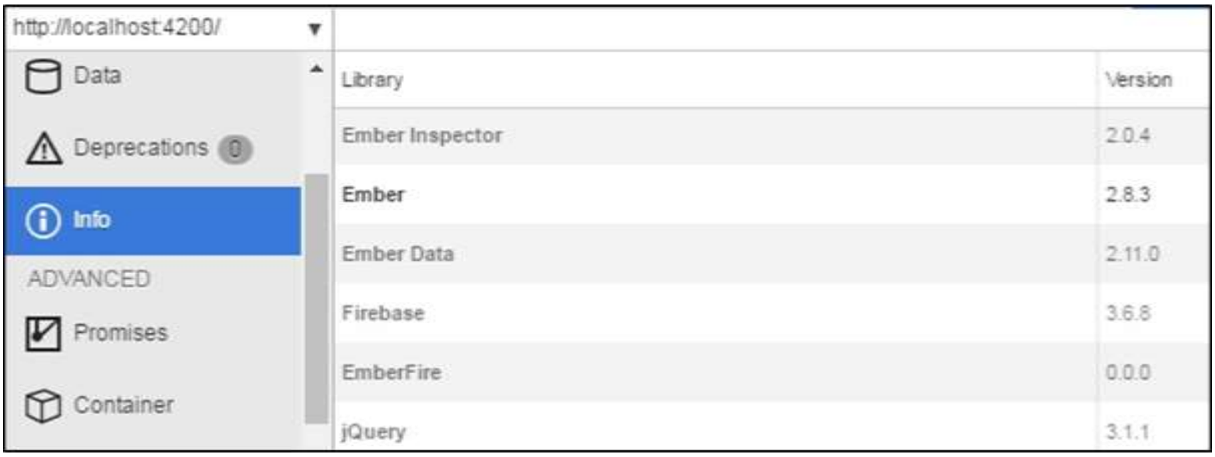


Id	Owner	City
0	Will Smith	San Francisco
1	John Davidson	Seattle
2	Shane Watson	Portland

```
<super-rentals@model:rental::ember471:1>
▾ Attributes
  id: 1
  owner: John Davidson
  city: Seattle
▾ Flags
  isLoaded: true
  hasDirtyAttributes: false
  isSaving: false
```

Library Info

You can view the list of libraries used in the application by clicking on the *Info* tab:

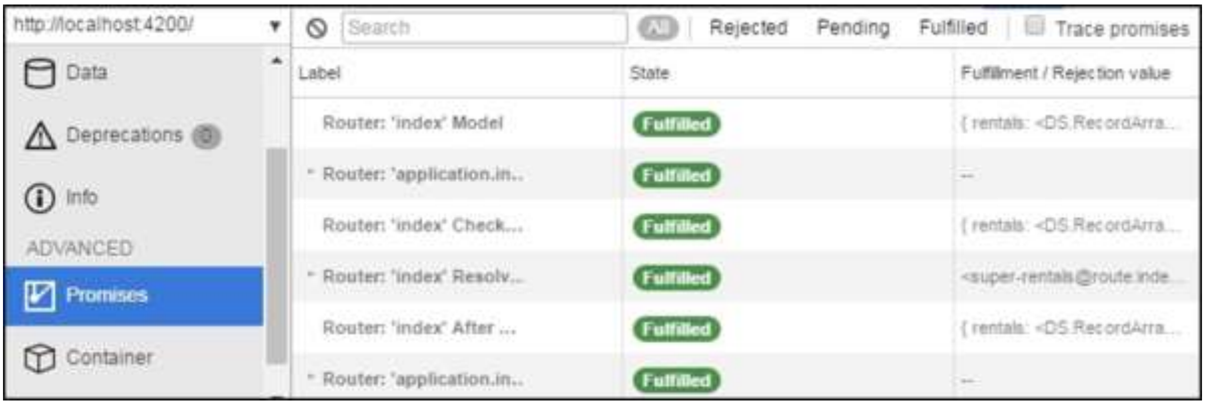


Library	Version
Ember Inspector	2.0.4
Ember	2.8.3
Ember Data	2.11.0
Firebase	3.6.8
EmberFire	0.0.0
jQuery	3.1.1

The *Info* tab displays the libraries along with the version as shown in the above image.

EmberJS – Debugging Promises

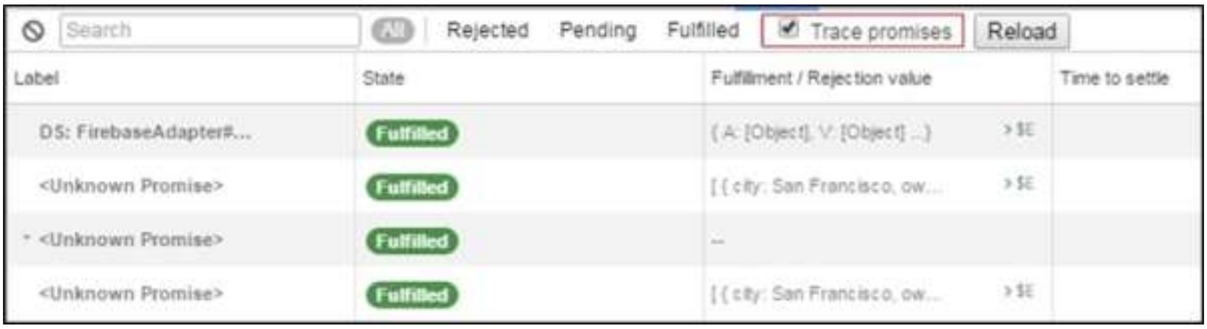
Ember inspector provides promises based on their states such as *Fulfilled*, *Pending* and *Rejected*. Click on the *Promises* tab and you will see the list of Promises with the specified state.



Label	State	Fulfillment / Rejection value
Router: 'index' Model	Fulfilled	{ rentals: <DS.RecordArra...
Router: 'application.in...	Fulfilled	--
Router: 'index' Check...	Fulfilled	{ rentals: <DS.RecordArra...
Router: 'index' Resolv...	Fulfilled	<super-rentals@route:inde...
Router: 'index' After ...	Fulfilled	{ rentals: <DS.RecordArra...
Router: 'application.in...	Fulfilled	--

As shown in the above screenshot, you can use *Rejected*, *Pending* and *Fulfilled* options to filter the promises. You can also use the search box for searching the promises.

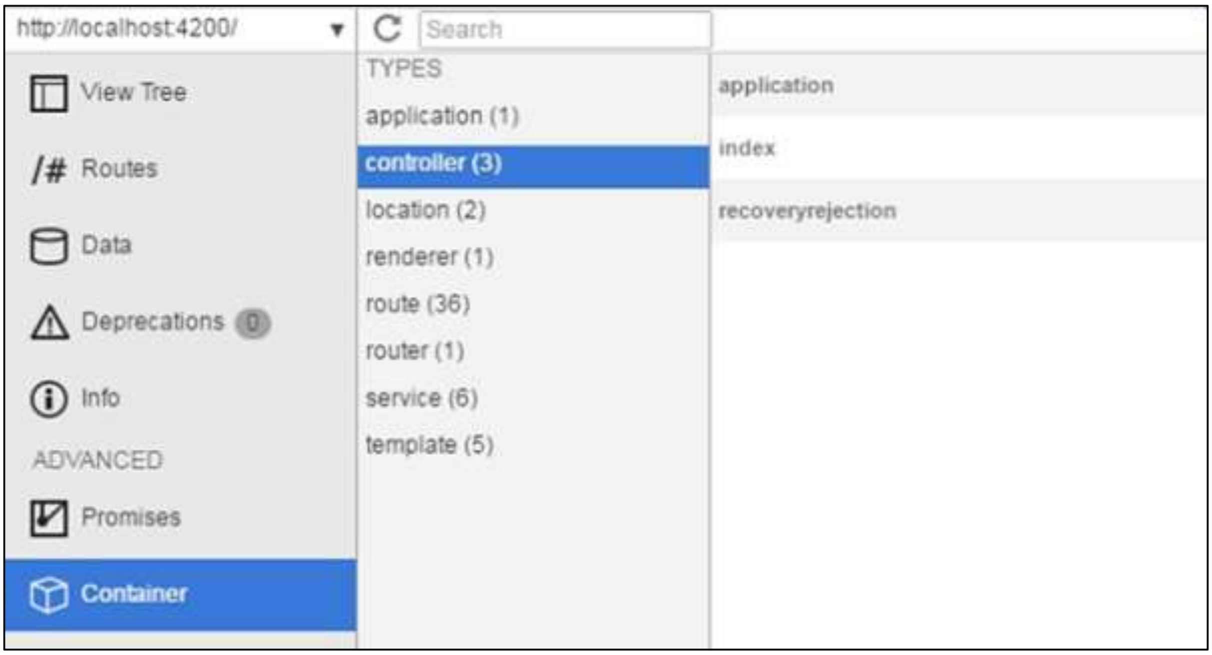
You can trace the promises by using the *Trace promises* option. By default, this option is disabled; you can enable it by checking the *Trace promises* checkbox as shown below:



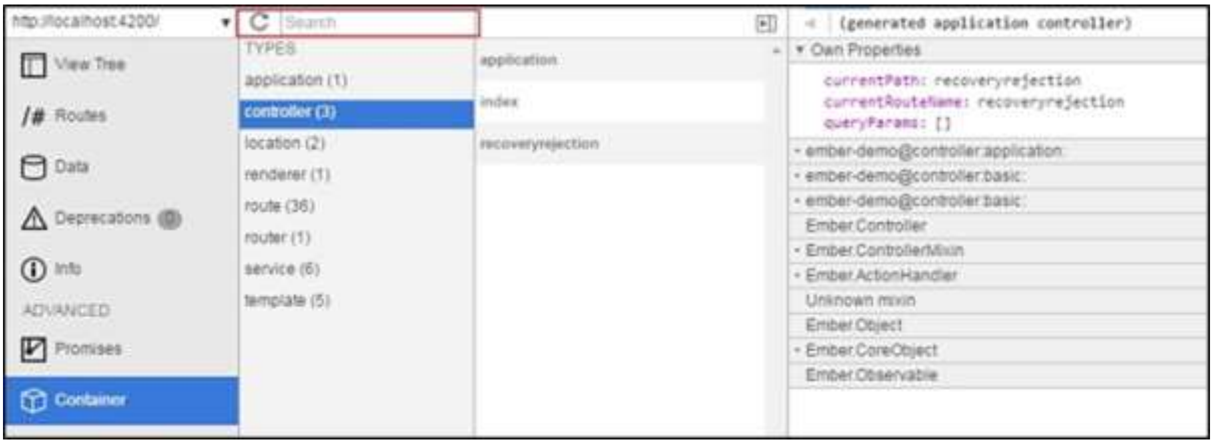
Label	State	Fulfillment / Rejection value	Time to settle
DS: FirebaseAdapter#...	Fulfilled	{ A: [Object], V: [Object] ...} › $E	
<Unknown Promise>	Fulfilled	[{ city: San Francisco, ow... › $E	
<Unknown Promise>	Fulfilled	--	
<Unknown Promise>	Fulfilled	[{ city: San Francisco, ow... › $E	

EmberJS – Inspecting Objects and Rendering Performance

In the Ember application, object instances are maintained by the container for inspecting the objects. You can inspect the objects by using the *Container* tab as shown below:



Next, click on the object name to inspect the instance by using an object inspector.

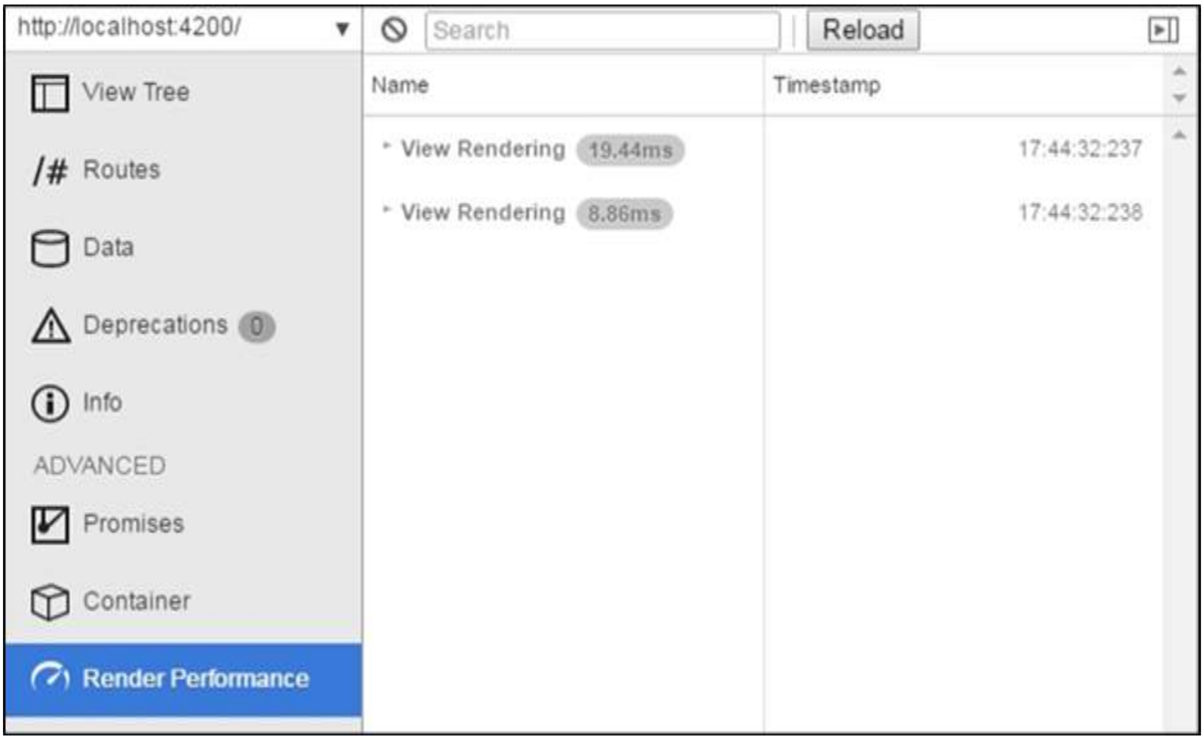


```
(generated application controller)
▾ Own Properties
    currentPath: recoveryrejection
    currentRouteName: recoveryrejection
    queryParams: []
▸ ember-demo@controller:application:
▸ ember-demo@controller:basic:
▸ ember-demo@controller:basic:
  Ember.Controller
▸ Ember.ControllerMixin
▸ Ember.ActionHandler
  Unknown mixin
  Ember.Object
▸ Ember.CoreObject
  Ember.Observable
```

In the above marked area, use the reload icon to reload the container and search box for searching the instances.

Rendering Performance

Ember inspector allows you to compute the application's render time by using the *Render Performance* tab.



Name	Timestamp
View Rendering 19.44ms	17:44:32:237
View Rendering 8.86ms	17:44:32:238

Use the *Reload* button to compute the render time of components and templates on the initial application boot and use the search box to filter the logs.