Backbone.js Cheat Sheet

A concise reference for Backbone.js, covering models, views, collections, routers, and events, along with best practices for building structured JavaScript applications.

Backbone.js Fundamentals

Core Concepts

Models: Represent data and business logic.
Views: Handle the user interface and presentation.
Collections: Ordered sets of models.
Routers: Manage application state and navigation.
Events: Enable communication between components.

Backbone.js is a lightweight framework that provides structure to JavaScript applications by introducing models with key-value binding and custom events, collections with a rich API of enumerated functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

Setting up Backbone

Include Backbone.js library	`<script src="underscore.js"></script>` `<script src="jquery.js"></script>` `<script src="backbone.js"></script>`
Dependencies	Backbone.js depends on Underscore.js and jQuery (or Zepto.js).

Backbone Object

The `Backbone` object is the entry point to the library and contains all the core functionalities.

It provides methods for creating models, views, collections, and routers.

Models & Collections

Model Definition

```
var Book = Backbone.Model.extend({
  defaults: {
    title: 'Default Title',
    author: 'Unknown',
    year: 2023
  },
  initialize: function() {
    console.log('A new book has been created.');
  }
});
```

Define a Model by extending `Backbone.Model`.

`defaults`: Specify default attribute values.

`initialize`: Constructor logic for the model.

Model Attributes

Get Attribute	`book.get('title'); // Returns the title`
Set Attribute	`book.set({ title: 'New Title' });`
Check if Attribute Exists	`book.has('title'); // Returns true/false`

Collection Definition

```
var Library = Backbone.Collection.extend({
  model: Book
});
```

Define a Collection by extending `Backbone.Collection`.

`model`: Specify the type of model the collection contains.

Collection Operations

Add Model	`library.add(book);`
Remove Model	`library.remove(book);`
Fetch Models from Server	`library.fetch();`
Filter Models	`library.where({ year: 2023 });`

Views & Events

View Definition

```
var BookView = Backbone.View.extend({
  el: '#book-container',
  initialize: function() {
    this.render();
  },
  render: function() {
    this.$el.html('Book Title: ' + this.model.get('title'));
    return this;
  }
});
```

Define a View by extending `Backbone.View`.

`el`: Specify the DOM element the view is associated with.

`initialize`: Constructor logic for the view.

`render`: Method to render the view's content.

Event Handling

View Events	`events: { 'click .button': 'handleClick' }, handleClick: function() { console.log('Button clicked!'); }`
Model Events	`this.listenTo(this.model, 'change', this.render);`
Collection Events	`this.listenTo(this.collection, 'add', this.render);`

Rendering Views

Views are rendered by populating the DOM with data from the model.

Use templates (e.g., Underscore templates, Handlebars) to generate HTML.

```
render: function() {
  var template = _.template($('#book-template').html());

  this.$el.html(template(this.model.toJSON()));
  return this;
}
```

Routers & Best Practices

Router Definition

```
var AppRouter = Backbone.Router.extend({
  routes: {
    '': 'home',
    'books/:id': 'bookDetails'
  },
  home: function() {
    console.log('Home route');
  },
  bookDetails: function(id) {
    console.log('Book details for ID: ' + id);
  }
});
```

Define a Router by extending `Backbone.Router`.

`routes`: Map URL routes to handler functions.

Navigation

Navigate to Route	`router.navigate('books/1', { trigger: true });`
Start History	`Backbone.history.start();`

Best Practices

- **Use a build tool:** Webpack, Parcel, or Browserify to manage dependencies and bundle your application.
- **Keep views small and focused:** Each view should be responsible for a small part of the UI.
- **Use events for communication:** Models, views, and collections can communicate through events.
- **Follow a consistent coding style:** Use a linter to enforce a consistent coding style.

- **Use a modular architecture:** Break your application into smaller, reusable modules.
- **Test your code:** Write unit tests and integration tests to ensure your code is working correctly.
- **Use a RESTful API:** Design your API to follow RESTful principles.