Connecting to a Server

While Backbone doesn't require a server, most projects will use one, first to serve the application to the user's browser, and then to synchronize model data changes. Backbone implements the client-side of a RESTful protocol for persisting changes to model data.

How Backbone Uses the Server

- Backbone uses RESTful web requests to synchronize data to and from a server. Model data is serialized as JSON for network transmission.
- Backbone's data server does not have to be the same server that served the application, but the same origin policy applies - the same origin policy prevents scripts from accessing resources belonging to another site. This means a Backbone application cannot synchronize models to a server other than the server that served the current page (unless CORS is being used)

Same origin policy: an origin is the application layer protocol + domain-name + port number

- http://localhost:3000
- http://localhost:3000/a is the same origin as http://localhost:300/b
- http://localhost:3001/a is **not** the same origin as http://localhost:3000/a
- http://localhost:3000/a is **not** the same origin as https://localhost:3000/a

Cross-origin Resource Sharing (CORS)

- technology that allows cross-origin requests. It is an explicit exception to the same origin policy.
- to implement CORS, a server uses special http headers to specify the set of valid origins
- it is an alternative to jsonp, which is supported in older browsers where CORS is not. CORS supports all http verbs while jsonp only supports GET.
- CORS is supported by most modern browsers

Backbone's persistence protocol is very similar to other RESTful systems, mapping HTTP verbs to data operations. Data fetching is accomplished using a GET verb. Data creation is done via POST, data updating by PUT, and deleting data uses DELETE. The only specialization is that Backbone expects the server to return an id property that uniquely identifies the resource on the server. The server can be any RESTful server technology: Ruby on Rails, .NET MVC, Express, Django.

Collection requests: create

```
var Books = Backbone.Collection.extend({
    url: '/books'
});
var books = new Books();

// create saves to the server and adds to the collection
books.create({ title: 'Wool', author: 'Hugh Howey', genre: 'Science Fiction' }
```

Collection requests: fetch

```
var Books = Backbone.Collection.extend({
    url: '/books'
});
var books = new Books();

// get the books collection from the server
books.fetch();
```

Model requests: fetch requires an id and retrieves a single model from the server

```
var Book = Backbone.Model.extend({
    url: '/books',
    id: 1
});

// calling fetch on a model gets the model's state from the server
model.fetch();
```

Model requests: save creates or updates depending on model.isNew() - if the model has never been saved before, Backbone will issue a POST to create a new model. If the model has been saved before, Backbone will issue a PUT request to update the model.

```
var Book = Backbone.Model.extend({
    urlRoot: '/books'
});
var book = new Book({ title: 'Wool', author: 'Hugh Howey' });

// calling save on a model without an id will do a POST
book.save();

var book = new Book({ title: 'Dust', author: 'Hugh Howey', id: 2 });

// calling save on a model with an id will do a PUT
book.save()
```

Saving a new model requires just making a new model and calling save() on it after adding it to the collection (unless you specify a url property on the model). Alternatively you can use collection.create to do both at once.

```
var Book = Backbone.Model.extend({
    url: '/books'
});
var Books = Backbone.Collection.extend({
    url: '/books',
    model: Book
});
var book = new Book({ title: 'Wool', author: 'Hugh Howey' });
var books = new Books();

// calling save on a model without an id will do a POST
book.save();
books.add(book)

// alternatively
books.create({ title: 'Wool', author: 'Hugh Howey' });
```

Model.destroy - deletes a single model on the server and removes it from its client-side collection

```
// deleting the last model in the collectin
var modelToDelete = collection.at(collection.length - 1);
modelToDelete.destroy();
```

Making Models flexible

- `parse()`
- `toJSON()`
- `sync()`

parse and toJSON: mirror images of each other

- **parse** represents how you affect the incoming serialization - how do you get from data you're putting into your model to what actually gets stored
- **toJSON** represents how you affect the representation of the data as it's pulled out of the model

example of parse

```
{
    book: {
        title: 'The Long Goodbye',
        author: 'Raymond Chandler',
        genre: 'Fiction'
    }
}

var BookModel = Backbone.Model.extend({
    parse: function (attrs) {
        return attrs.book;
    }
});

book.fetch();
book.get('author'); // Raymond Chandler
```

more involved example of parse

```
{
    content: 'you have no idea...',
    author: {
        email: 'jamdon@uw.edu',
        username: 'jamdon',
        avatar: 'someimage.jpg'
    }
}
```

```
var CommentModel = Backbone.Model.extend({
    initialize: function () {
        this.author = new UserModel()l
    },
    parse: function (attrs) {
        if (attrs.author) {
            this.author.set(attrs.author);
        }
        return _.omit(attrs, 'author');
    }
});
```

the inverse - modifying toJSON - the options passed to toJSON are the same options passed to sync()

```
var CommentModel = Backbone.Model.extend({
    ...
    toJSON: function (options) {
        var json = Backbone.Model.prototype.toJSON.call(this);

        if (options && options.author) {
            json.author = this.author.toJSON();
        }
        return json;
    }
});
```

```
comment.toJSON({ author: true });
```

sync() - Override this to get custom behavior when making ajax calls to a server endpoint. For example, if you're not talking to a RESTful endpoint, this still allows the underlying plumbing in Backbone to function seamlessly.