Week 10

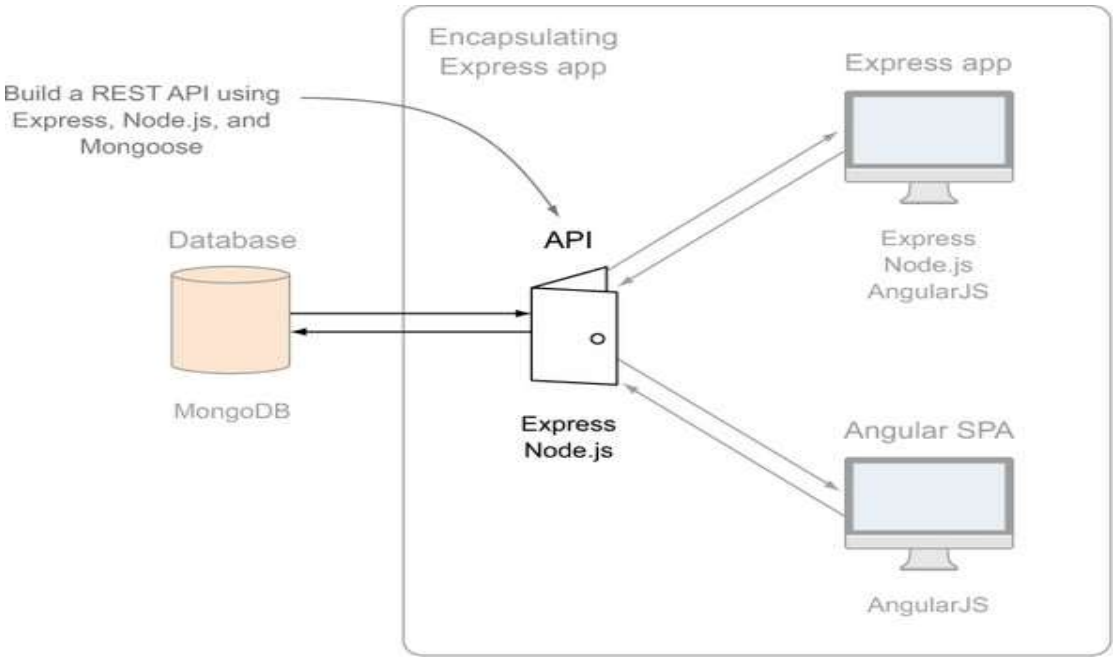
AWD

REST API: Exposing the MongoDB database to the application

- Rules of REST APIs
- API patterns
- Typical CRUD functions (create, read, update, delete)
- Using Express and Mongoose to interact with MongoDB
- Testing API endpoints

As we come in to this chapter we have a MongoDB database set up, but we can only interact with it through the MongoDB shell. During this chapter we'll build a REST API so that we can interact with our database through HTTP calls and perform the common CRUD functions: create, read, update, and delete.

We'll mainly be working with Node and Express, using Mongoose to help with the interactions. Figure 1 shows where this chapter fits into the overall architecture.



We'll start off by looking at the rules of a REST API. We'll discuss the importance of defining the URL structure properly, the different request methods (GET, POST, PUT, and DELETE) that should be used for different actions, and how an API should respond with data and an appropriate HTTP status code. Once we have that knowledge under our belts we'll move on to building our API for Loc8r, covering all of the typical CRUD operations. As we go, we'll discuss a lot about Mongoose, and get into some Node programming and more Express routing.

The rules of a REST API

Let's start with a recap of what a REST API is. From chapter 2 you may remember:

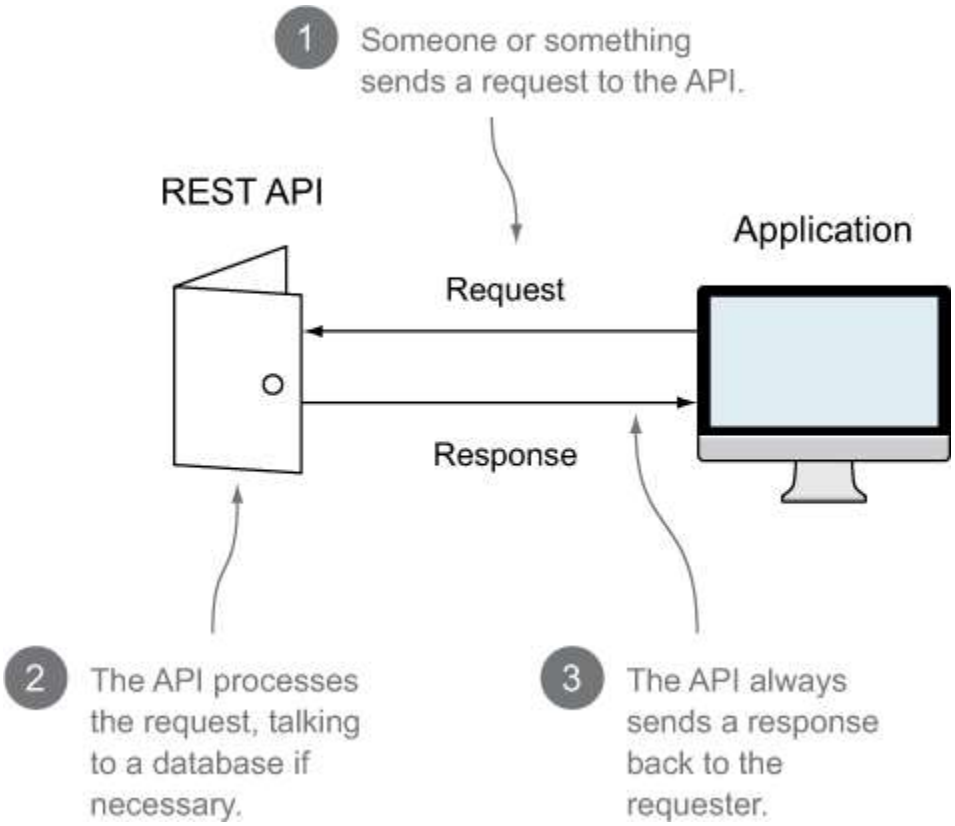
- REST stands for REpresentational State Transfer, which is an architectural style rather than a strict protocol. REST is stateless—it has no idea of any current user state or history.
- API is an abbreviation for application program interface, which enables applications to talk to each other.

So a REST API is a stateless interface to your application. In the case of the MEAN stack the REST API is used to create a stateless interface to your database, enabling a way for other applications to work with the data.

REST APIs have an associated set of standards. While you don't have to stick to these for your own API it's generally best to, as it means that any API you create will follow the same approach. It also means you're used to doing things in the "right" way if you decide you're going to make your API public.

In basic terms a REST API takes an incoming HTTP request, does some processing, and always sends back an HTTP response, as shown in figure .2.

Figure 2. A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.



The standards that we're going to follow for Loc8r revolve around the requests and the responses.

1. Request URLs

Request URLs for a REST API have a simple standard. Following this standard will make your API easy to pick up, use, and maintain.

The way to approach this is to start thinking about the collections in your database, as you'll typically have a set of API URLs for each collection. You may also have a set of URLs for each set of subdocuments. Each URL in a set will have the same basic path, and some may have additional parameters.

Within a set of URLs you need to cover a number of actions, generally based around the standard CRUD operations. The common actions you'll likely want are

- Create a new item
- Read a list of several items
- Read a specific item
- Update a specific item
- Delete a specific item

Using Loc8r as an example, the database has a Locations collection that we want to interact with. Table 6.1 shows how the URLs and parameters might look for this collection.

Table . URL paths and parameters for an API to the Locations collection; all have the same base path, and several have the same location ID parameter (view table figure)

Action	URL path	Parameters	Example
Create new location	/locations		http://loc8r.com/api/locations
Read list of locations	/locations		http://loc8r.com/api/locations
Read a specific location	/locations	locationid	http://loc8r.com/api/locations/123
Update a specific location	/locations	locationid	http://loc8r.com/api/locations/123
Delete a specific location	/locations	locationid	http://loc8r.com/api/locations/123

As you can see from table 6.1, each action has the same URL path, and three of them expect the same parameter to specify a location. This poses a very obvious question: How do you use the same URL to initiate different actions? The answer lies in request methods.

1.2. Request methods

HTTP requests can have different methods that essentially tell the server what type of action to take. The most common type of request is a GET request—this is the method used when you enter a URL into the address bar of your browser. Another common method is POST, often used when submitting form data.

Table 6.2 shows the methods we'll be using in our API, their typical use cases, and what you'd expect returned.

Table 2. Four request methods used in a REST API (view table figure)

Request method	Use	Response
POST	Create new data in the database	New data object as seen in the database
GET	Read data from the database	Data object answering the request
PUT	Update a document in the database	Updated data object as seen in the database
DELETE	Delete an object from the database	Null

The four HTTP methods that we'll be using are POST, GET, PUT, and DELETE. If you look at the first word in the "Use" column you'll notice that there's a different method for each of the four CRUD operations.

Each of the four CRUD operations uses a different request method.

The method is important, because a well-designed REST API will often have the same URL for different actions. In these cases it's the method that tells the server which type of operation to perform. We'll discuss how to build and organize the routes for this in Express later in this chapter.

So if we take the paths and parameters and map across the appropriate request method we can put together a plan for our API, as shown in table 6.3.

Table 3. Request method is used to link the URL to the desired action, enabling the API to use the same URL for different actions (view table figure)

Action	Method	URL path	Parameters	Example
Create new location	POST	/locations		http://loc8r.com/api/locations
Read list of locations	GET	/locations		http://loc8r.com/api/locations
Read a specific location	GET	/locations	locationid	http://loc8r.com/api/locations/123
Update a specific location	PUT	/locations	locationid	http://loc8r.com/api/locations/123
Delete a specific location	DELETE	/locations	locationid	http://loc8r.com/api/locations/123

Table 6.3 shows the paths and methods we'll use for the requests to interact with the location data. As there are five actions but only two different URL patterns, we can use the request methods to get the desired results.

Loc8r only has one collection right now, so this is our starting point. But the documents in the Locations collection do have reviews as subdocuments, so let's quickly map those out too.

API URLs for subdocuments

Subdocuments are treated in a similar way, but require an additional parameter. Each request will need to specify the ID of the location, and some will also need to specify the ID of a review. Table 6.4 shows the list of actions and their associated methods, URL paths, and parameters.

Table 4. API URL specifications for interacting with subdocuments; each base URL path must contain the ID of the parent document (view table figure)

Action	Method	URL path	Parameters	Example
Create new review	POST	/locations/locationid/reviews	locationid	http://loc8r.com/api/locations/123/reviews
Read a specific	GET	/locations/locationid/reviews	locationid reviewid	http://loc8r.com/api/locations/123/reviews/abc

Action	Method	URL path	Parameters	Example
review				
Update a specific review	PUT	/locations/locationid/reviews	locationid reviewid	http://loc8r.com/api/locations/123/reviews/abc
Delete a specific review	DELETE	/locations/locationid/reviews	locationid reviewid	http://loc8r.com/api/locations/123/reviews/abc

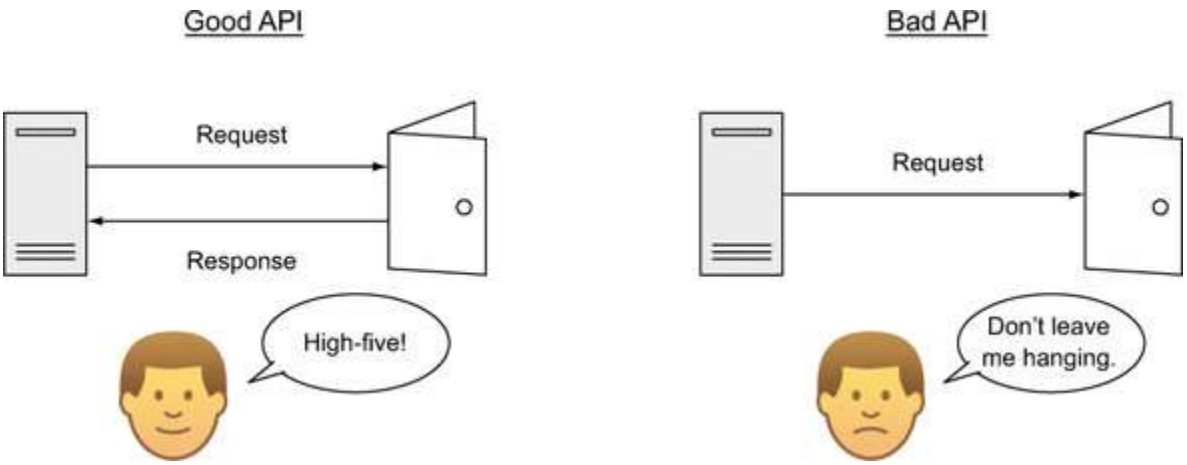
You may have noticed that for the subdocuments we don't have a "read a list of reviews" action. This is because we'll be retrieving the list of reviews as part of the main document. The preceding tables should give you an idea of how to create basic API request specifications. The URLs, parameters, and actions will be different from one application to the next, but the approach should remain consistent.

That's requests covered. The other half of the flow, before we get stuck in some code, is responses.

6.1.3. Responses and status codes

A good API is like a good friend. If you go for a high-five a good friend will not leave you hanging. The same goes for a good API. If you make a request, a good API will always respond and not leave you hanging. Every single API request should return a response. This contrast is shown in figure 6.3.

Figure 6.3. A good API always returns a response and shouldn't leave you hanging.



For a successful REST API, standardizing the responses is just as important as standardizing the request format. There are two key components to a response:

- The returned data
- The HTTP status code

Combining the returned data with the appropriate status code correctly should give the requester all of the information required to continue.

Returning data from an API

Your API should return a consistent data format. Typical formats for a REST API are XML and JSON. We'll be using JSON for our API because it's the natural fit for the MEAN stack, and it's more compact than XML, so it can help speed up the response times of an API.

Our API will return one of three things for each request:

- A JSON object containing data answering the request query
- A JSON object containing error data
- A null response

During this chapter we'll discuss how to do all of these things as we build the Loc8r API. As well as responding with data, any REST API should return the correct HTTP status code.

Using HTTP status codes

A good REST API should return the correct HTTP status code. The status code most people are familiar with is Table 6.5 shows the 10 most popular HTTP status codes and where they might be useful when building an API.

Table 5. Most popular HTTP status codes and how they might be used when sending responses to an API request (view table figure)

Status code	Name	Use case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	An unsuccessful GET, POST, or PUT request, due to invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials

Status code	Name	Use case
403	Forbidden	Making a request that isn't allowed
404	Not found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request method not allowed for the given URL
409	Conflict	Unsuccessful POST request when another object already exists with the same data
500	Internal server error	Problem with your server or the database server

As we go through this chapter and build the Loc8r API we'll make use of several of these status codes, while also returning the appropriate data.