**REST APIs and models**

*REST* (**RE**presentational **S**tate **T**ransfer) is a style of software architecture, a set of principles for designing APIs.

When creating a REST API, our primary concern is communicating the state of *resources*. A resource could be something like a shopping list, a blog post, a set of blog posts, a representation of a user, etc.

There are four generic operations that most REST APIs support: *creating* new resources, *reading* or retrieving existing resources, *updating* resources, and *deleting* resources. This is where the acronym *CRUD* comes from (create, read, update, delete), and you'll frequently hear developers talk about basic CRUD operations, or CRUD applications.

A RESTful API exposes resources through URLs. For instance, you might have a single endpoint, /users, that is for exposing user resources. When a client makes a GET request to /users, the API might return a list of all users. When clients make a GET request to /users/my-unique-user-id, the API would return a single object representing the user with that ID. This same endpoint (/users) could be used for creating, updating, and deleting users as well.

Note what we do **not** do when creating a REST API. When creating an endpoint for creating a new user, we don't use the URL /users/add-new-user.

Instead, endpoints are *nouns* (that is, things), and we use [HTTP methods](http://www.restapitutorial.com/lessons/httpmethods.html) as *verbs* to indicate the appropriate actions. We use *GET* for reading or retrieving, *POST* for creating, *PUT* (and in some cases *PATCH*, but we'll use *PUT* in this course) for updating resources, and *DELETE* for deleting resources.

**Model layer**

When building REST APIs with Express and other modern server-side frameworks, we usually separate out our storage layer (that is, our database) from our API layer by using *models*. We won't cover databases until Unit 2 of this course, so as we build out the shopping list/recipes app, we'll be using volatile (in memory) storage; but nevertheless, the principle of decoupling API and storage layers applies here as well.

Here's an example of what that might look like:

*// server.js*

*//*

*// ... imports*

**const** {Users} = require('../models/users');

*// ... other stuff*

app.get('/users', (req, res) => {

res.json(Users.get()); *// returns a JSON array of user objects*

});

*// ... other stuff*

Here, we have a server that exposes an endpoint at /users. When a GET request is made to that endpoint, our server responds by calling the .get()method on a Users *model* that we've imported. Notice that the API layer doesn't have any conception of how the Users model goes about getting the data. The API layer just knows that the Users model has a get method that can be called to retrieve a list of stored users.

Best practice is to create "thin" APIs with a minimal amount of logic. Models are where the logic for sourcing and manipulating data should live. This will become more apparent in Unit 2 as we begin to work with MongoDB and Mongoose to model our data, but you'll get a taste of it in the remainder of this lesson.

**Our models for this lesson**

In the remainder of this lesson, we'll build endpoints for creating, retrieving, updating, and deleting shopping list items and recipes. To do this, we'll use pre-made ShoppingList and Recipes models.

Since we haven't covered databases yet, our models will store our data in memory, which means that when the server restarts, we'll lose our data.

Later, in Unit 2, when we start working with Mongo and Mongoose, the database-backed models we create will have a similar feel to the ShoppingList and Recipes model.

Let's have a brief look at the ShoppingList model.

**const** ShoppingList = {

create: **function**(name, checked) {

console.log('Creating new shopping list item');

**const** item = {

name: name,

id: uuid.v4(),

checked: checked

};

**this**.items[item.id] = item;

**return** item;

},

get: **function**() {

console.log('Retrieving shopping list items');

**return** Object.keys(**this**.items).map(key => **this**.items[key]);

},

**delete**: **function**(id) {

console.log(`Deleting shopping list item \`${id}\``);

**delete** **this**.items[id];

},

update: **function**(updatedItem) {

console.log(`Deleting shopping list item \`${updatedItem.id}\``);

**const** {id} = updatedItem;

**if** (!(id **in** **this**.items)) {

**throw** StorageException(

`Can't update item \`${id}\` because doesn't exist.`)

}

**this**.items[updatedItem.id] = updatedItem;

**return** updatedItem;

}

};

The first thing to say about this model is that *the implementation details are not important* to understand at this point. What is important to know is that the model has four methods: create, get, delete, and update. This model is for managing items in a single shopping list — we are not providing a way to manage multiple shopping lists.

The create() method takes two parameters: name which is the name of the shopping list item, and checked which is a boolean value indicating if the item is checked off or not. To add a new item to the ShoppingList model, just call create with the appropriate arguments. Note that this method generates a random id for the new item, and returns an object representing the item.

The get() method is used to retrieve all items that are currently stored in the ShoppingList. At this point, we're not supporting retrieval of a specific shopping list item.

The delete() method takes a single argument: the id of the shopping list item to be deleted. Note that if you pass this method an id for an item that does not exist, no error will be thrown.

Finally, the update() method takes a single parameter: updatedItem which is an object representing the updated item. Note that this method *will* throw an error if it tries to update a non-existent item.

The Recipes model looks much the same. Here it is:

**const** Recipes = {

create: **function**(name, ingredients) {

console.log('Creating a new recipe');

**const** item = {

name: name,

id: uuid.v4(),

ingredients: ingredients

};

**this**.items[item.id] = item;

**return** item;

},

get: **function**() {

console.log('Retreiving recipes');

**return** Object.keys(**this**.items).map(key => **this**.items[key]);

},

**delete**: **function**(itemId) {

console.log(`Deleting recipe with id \`${itemId}\``);

**delete** **this**.items[itemId];

},

update: **function**(updatedItem) {

console.log(`Updating recipe with id \`${updatedItem.id}\``);

**const** {id} = updatedItem;

**if** (!(id **in** **this**.items)) {

**throw** StorageException(

`Can't update item \`${id}\` because doesn't exist.`)

}

**this**.items[updatedItem.id] = updatedItem;

**return** updatedItem;

}

};

Notice that both models have the same methods: create, get, delete, and update. For get, delete, and update, the call signature (that is, the arguments you need to supply to the method) is identical. The createmethod for Recipes differs in that you must supply a name for the recipe and an array of ingredients.