Creating the Backend, Step by Step  
https://scotch.io/tutorials/getting-started-with-node-express-and-postgres-using-sequelize

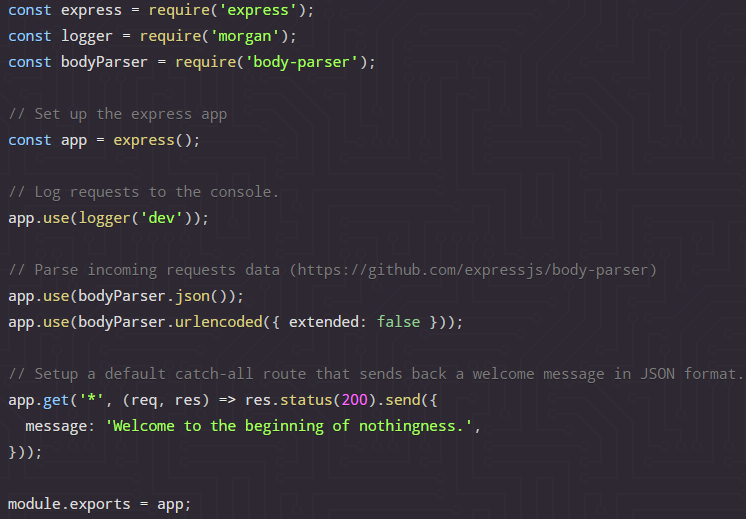
*Note: there’s a typo in the tutorial. He first creates server/controllers/todo.js, then references it in the code as server/controllers/todos.js. This checklist stays consistent with todo.js*

* npm init -y
* npm install --save express body-parser nodemon sequelize pg pg-hstore

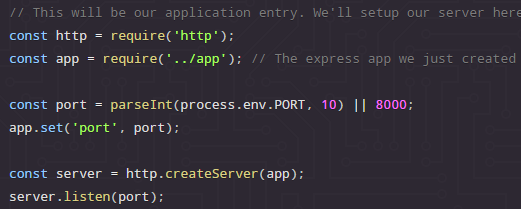
npm install --save sequelize-cli -g

*pg will be responsible for creating the databse, while pg-hstore is a module for serializing/ deserializing JSON data into the postgres hstore format*

* Createa file called app.js in the root:

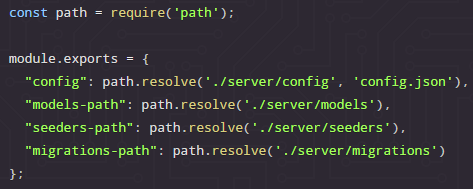


* Create a file called \_\_\_; this will be the application entry to set up the server:



* npm start
* Create a file called .sequelizerc in the root.

*Here we configure Sequelize by specifying the paths for the files that Sequelize will create upon running sequelize init. It looks like this:*



*config.json contains our application configuration settings, such as database authentication configuration. The migrations folder will hold our application's migrations, while the models folder will hold the application models. Seed data is initial data provided with a system for testing, training or templating purposes. The seeders folder typically holds seed data.*

* Run **sequelize init** to create the folders we just specified and generate the boilerplate code
* In **models/index.js**, change back slashes in config to forward slashes
* OPTIONAL: refactor the index.js boilerplate code from ES5 to ES6
* Update username, password, database and dialect in config.json



* "C:\Program Files\PostgreSQL\9.6\bin\createdb" -U postgres todos\_dev

*This creates the database from the command line; note that the database name must be the same as the one specified in our config.json. With our database and configuration in place, we’re now ready to generate models. In this example, We are going to have two models, Todo and TodoItem. The relationship between a Todo and its TodoItems is going to be one-to-many, such that a Todo can have many TodoItems while a TodoItem can only belong to one Todo.*

* sequelize model:create --name Todo --attributes title:string

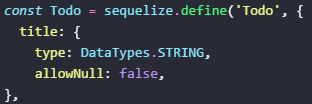
*This generates two files with boilerplate code:*

* *models/todo.js In this file, we are defining our Todo model.   
   It’s going to have a single attribute, title, which is a string.*
* *migrations/<date>-create-todo.js*
* sequelize model:create --name TodoItem --attributes content:string

*This generates two files with boilerplate code:*

* *models/todoitem.js In this file, we are defining our TodoItem model.   
   It’s going to have a single attribute, content, which is a string.*
* *migrations/<date>-create-todoitem.js*
* OPTIONAL: refactor the generated files from ES5 to ES6
* OPTIONAL: add a not-null constraint

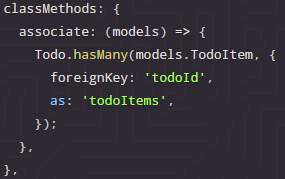
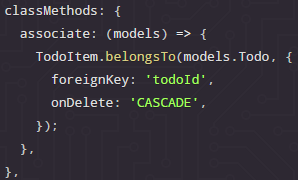
*This means that the database will not allow us to write to it if we don't provide a value for the title field.*



* Define the relationships between the models In the classMethods section of the generated model code

*The****as: 'todoItems'****means that every time we query for a todo and include its todo items, they'll be included under the key todoItems instead of TodoItems (Sequelize defaults to using the model name). This change is just optional*

*Notice that we've edited both the content and complete fields. We've added a not-null constraint in the content field and a default value for the complete field. In general, having a default value means that if we don't provide a value for that field when creating it, the database is going to use the provided default value for that field. In addition to that, we've also defined the relationship between the TodoItems and the Todo objects. The onDelete: CASCADE tells Postgres that if we delete a todo, its associated todo items should be deleted as well (cascade the delete action).*

* OPTIONAL: refactor migration files from ES5 to ES6

*Look in the migrations files. When we run these migrations, the up function will be executed. It will take care of creating the table and its associated columns for us. If, for whatever reason, we needed to rollback (undo) the migration, the down function would be executed and it would undo whatever the up function did, thus returning the our database to the same state it was in before we performed the migration.*

*These migrations are a representation of how we want our models to look like in the database. Notice we define the relationship between our models in the create-todo-item.js migration file as well. The todoId field was not automatically generated for the todo items and we've had to manually define it. Sequelize automatically generates the id, createdAt and updatedAt fields for you. In addition to that, any time a model is saved, the updatedAt field is automatically updated to reflect the new update time.*

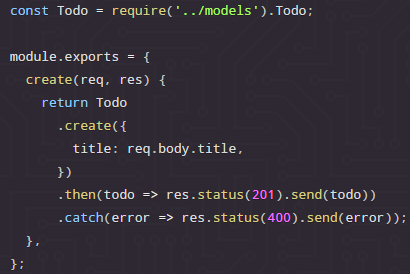
* sequelize db:migrate

*With the models and migrations in place, we're now ready to persist the models to the database by running the migrations.* *This command will discover the migrations in our migrations folder and execute them. If you try running the same command again, it would not execute any migrations since it's clever enough to know that all of the current migrations have been executed.*

* Create a new file: controllers/todo.js

*Now it’s time to make the controllers. We're going to have two controllers, todosController and todoItemsController. The todosController will be responsible for creating, listing, updating and deleting todos, while the todoItemsController will be responsible for creating, updating and deleting todo items.*

* ADD FUNCTIONALITY: create todos
  1. Open controllers/todo.js and add:



*This code creates a new todo and if successful, it returns it. If it encounters an error, it returns that error instead. (Granted, this isn't the best way to handle these errors, but we'll go with it for now, for the sake of simplicity.) This create function is designed to be a*[*route handler*](https://expressjs.com/en/guide/routing.html)*for whichever Express route we'll choose to attach it to. The req parameter is the incoming request from the client. The res parameter is the response we're preparing to eventually send back to the client in response to their request. All Express route handlers follow this method signature. We can have a third parameter, conventionally named next, which is a function that passes the request on to the next route handler (meaning that a route can be handled by multiple route handlers, in which case it's piped or passed along all of those route handlers). We are, however, not going to see a use case for that in this application*

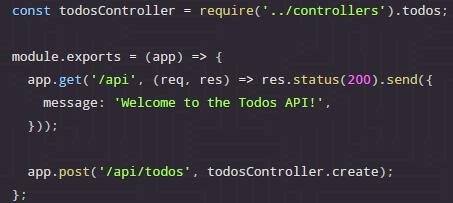
* Create a new file: controllers/index.js

*We’ll export our controllers from index.js. “I find this helpful since it helps me consolidate my imports (require statements) from one central place”. Next we need to add an API route that maps to this functionality.*

* Create a new folder and file: server/routes/index.js

*We are going to place all our routes here. However, in a real-world application, you might want to split up your routes and place them in different folders.*

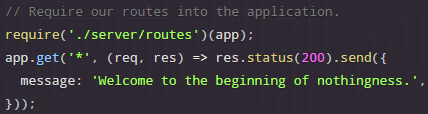
1. Create the api and the route to map to the create functionality



*This will add two new routes, a welcome route at /api and a route to create todos at /api/todos. When we hit /api, we are instructing our application to send back a JSON object welcoming the user to our life-changing Todos API. If we post some data to /api/todos, we are telling our application to run the todosController.create function, which will take the request object, extract the posted data and create a todo from it. In this case, we say that the todosController.create function is the POST route handler for the /api/todos endpoint.*

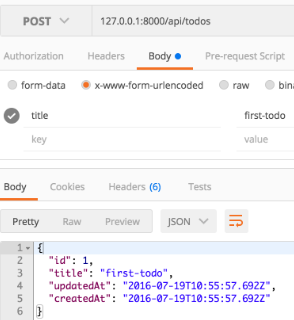
* Require the routes into the Express app (root: app.js)

*We need to make the application aware that we just added the routes. We do this by adding a require statement right before the route we'd earlier created, such that our app.js file now looks like:*



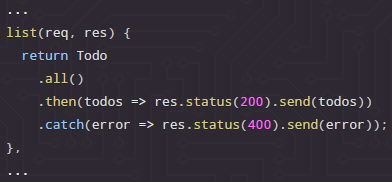
*Note that we have to require our routes before the app.get('\*', ...) catch-all route we'd added earlier. This is because the catch-all route will match any route and serve the welcome message, hence if we require our other routes after it, those other routes will never be hit.*

1. Open Postman and issue a POST request to create a new todo item:



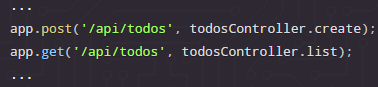
1. Make a GET request to /api using either Postman or your browser.

*You should see the welcome message we specified in our routes. If you visit any other route that we've not explicitly handled in our routes, you should still see the default "Welcome to the beginning of nothingness." message. Feel free to create a few more todos.*

* ADD FUNCTIONALITY: list all todos
  1. In server/controllers/todo.js, add the following code after the create method

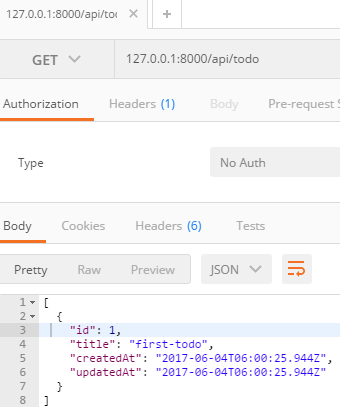
*In this code snippet, we're fetching all todos from our database and sending them back to the user as an array in the response. If we encounter an error while fetching the todos from the database, we send that error object instead.*

* 1. Create a new url that maps a todo GET request to the list method we just made



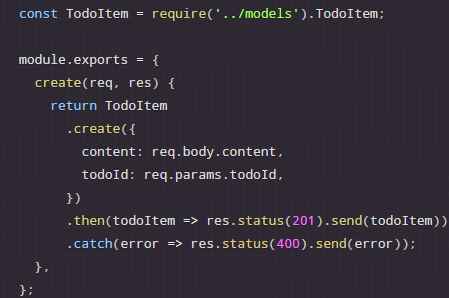
In server/routes/index.js, below the POST route we added earlier, enter:

* 1. Test the new GET route in Postman; you should see:



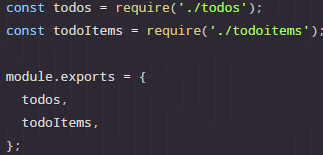
*If you inspect the output, you'll realize that our listed todos do not have any todoitems. Let's add functionality to create todoitems next, after which we'll modify our list method to return todos together with their todoitems.*

* ADD FUNCTIONALITY: create a todo item.
  1. Open controllers/todoitem.js and enter:



*In the above code snippet, we're creating a todoitem and associating it with a particular todo. We are grabbing the id of that particular todo from the request params.*

* In server/controllers/index.js, require todoItem and add it to the exports

**

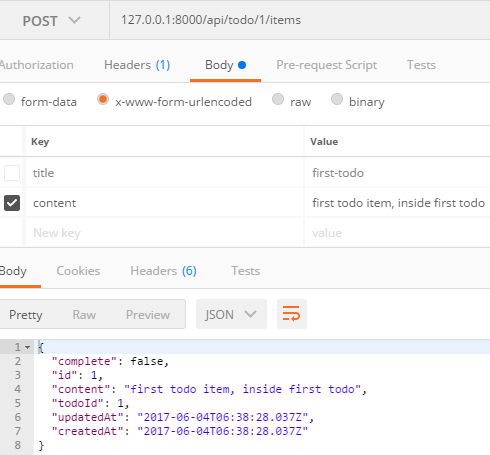
*Notice that we're using the*[*ES6 object shorthand notation*](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer)*to add the methods to module.exports. (We’re writing “todos” and “todoItems” instead of “todos: todos” and “todoItems: todoItems”)*

1. In server/routes/index.js, require todoitem and set up the route for creating a new todoItem

 […]

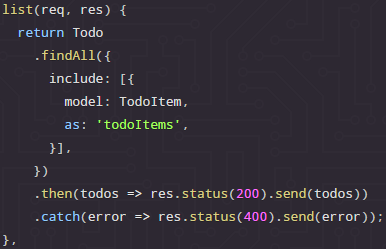
*The :todoId in the route is made available to us by Express in the request.params object as todoId and is the same one we're accessing in our controller.*

1. Test the new route in Postman



*Now that we can create todo items, let's modify our todoController.list code so that it returns a todo together with its associated items.*

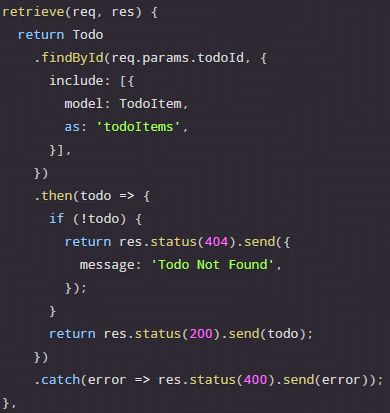
* In server/controller/todo.js, *change* it to:



*In the above code snippet, we find all todos and include all associated todoitems from the TodoItem model. We include them as todoItems, as we did when defining the relationship in the Todo model. Remember to require the TodoItem model at the top of your server/controllers/todos.js file.*

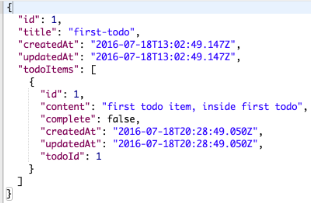
1. View the results by making a GET request to /api/todo in Postman



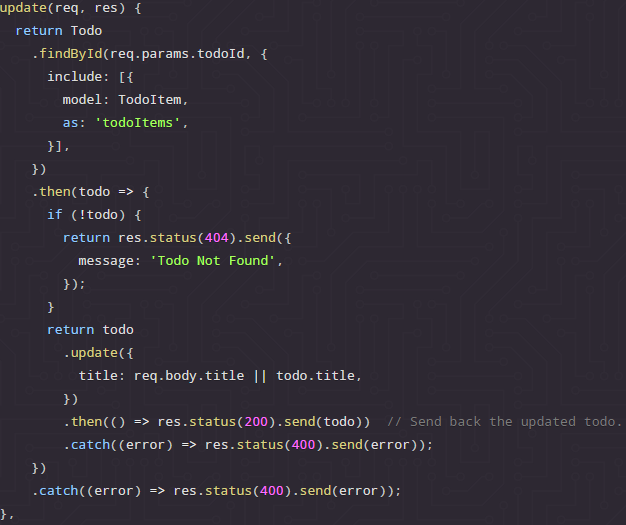
* ADD FUNCTIONALITY: retrieve a single todo based on its id
  1. Add retrieve method to server/controllers/todo.js:

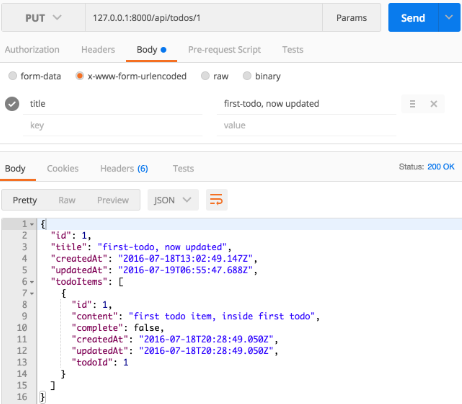
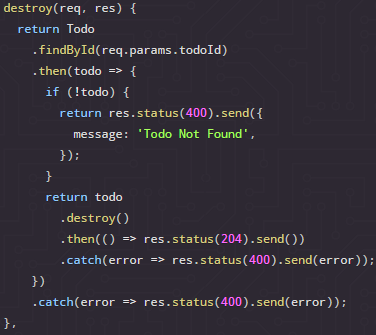
In this code snippet, we're finding the todo whose id matches the todoId we get from the request parameters and we're also including it's associated todo items. If such a todo exists, we're sending it back in the response. If it doesn't, we're sending back an error message letting the user know we didn't find the specified todo. If we encounter an error when processing this request, we're sending back the error object.

* 1. Open up routes/index.js and add a new route that maps to the retrieve view 
  2. Make a GET request to Postman at /api/todo/1  
      You’ll get todo 1 with its todo items in an array



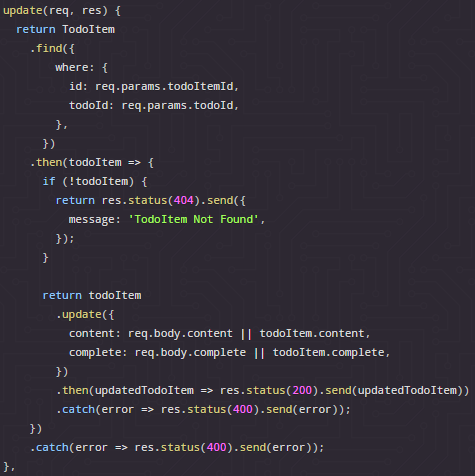
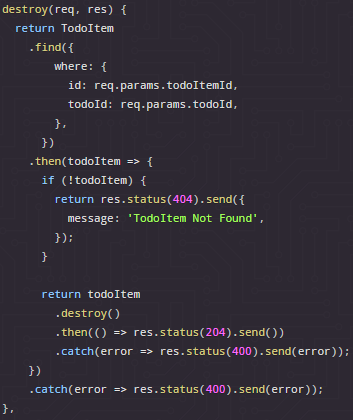
* Add Functionality: update a specific todo
  1. Open controllers/todo.js and add an update method with findById:

In this code snippet, we're finding the todo whose id matches the todoId supplied in the request params. We are then updating it's title. If no title was provided, we're defaulting to the title the todo already had.

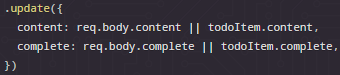
* 1. Open routes/index.js and add a new route that maps to the update method
  2. Test the PUT request in Postman
* Add functionality: deleting todos
  1. Open controllers/todo.js and add a destroy method with findById:

This code is almost the same as the one we had for updating a todo, except we're not including the todo items. Remember that when you delete a todo, its corresponding todo items are deleted as well. This is because we specified the onDelete action as CASCADE when we were setting up our models. If you try this out in Postman, you might be surprised that you don't get any data back. You can modify the delete code to return a 200 status code and a delete successful message

* 1. Open routes/index.js and add the corresponding route:
  2. Test in Postman
* Add Functionality: updating and deleting todo items
  1. Open controllers/todoitem.js and add two methods: update and destroy

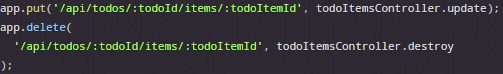
 *you'll notice that we're finding the todo item to either update or delete by two criteria: its own id which we're grabbing from the params as todoItemId and the id of it's parent todo, which we're obtaining from the params object as todoId.*

*Let us, for a moment, focus on the update method. In the update method, we are grabbing the provided todoItemId from the request. We are then finding the todo item with that id and in readiness to update it. If we don't find it, we return early and send and error message to the user.*

*Earlier on, when we were updating the todo title, we had this statement: To recap, we said that we either use the new title or default to the old one if a title was not provided. You will notice the same pattern when we're updating the todo item in this statement: While this approach works for our application, since we have a small number of fields, it wouldn't scale very well if you had to update a model with many fields. As such, you might want to use another approach where you give the Sequelize model update function the data and then specify the fields it should update. Using this approach, we change change our .update statement to:*

*To recap, using this approach, we pass the whole update object we get from the request (req.body) to the update function. Using ES6's Object.keys function, we extract the keys from the update object and tell the TodoItem Sequelize model to only update the fields that are present in the update data object. If we have a field in our model that's missing from the update object, the update operation will leave that field untouched. This saves us the trouble of having to define defaults using the || operator.*

* 1. Add two new routes in routes/index.js



* 1. Test in Postman

Author’s Final Thoughts:

There are some improvements you could decide to make, for example:

* Better error handling. Currently, we're assuming that all errors are due to the data the user has provided. We're also sending back the whole error object. That could be a security issue since you might leak information about your architecture to the end user.
* Form fields validation. We currently have no front-facing input fields validation. Whenever you're building a web application, it's imperative that you validate user input before it hits the database. Our current validation (not null constraint) occurs at the database level. One way of performing this validation would be by intercepting the request in a middleware and validating that it contains the required fields.