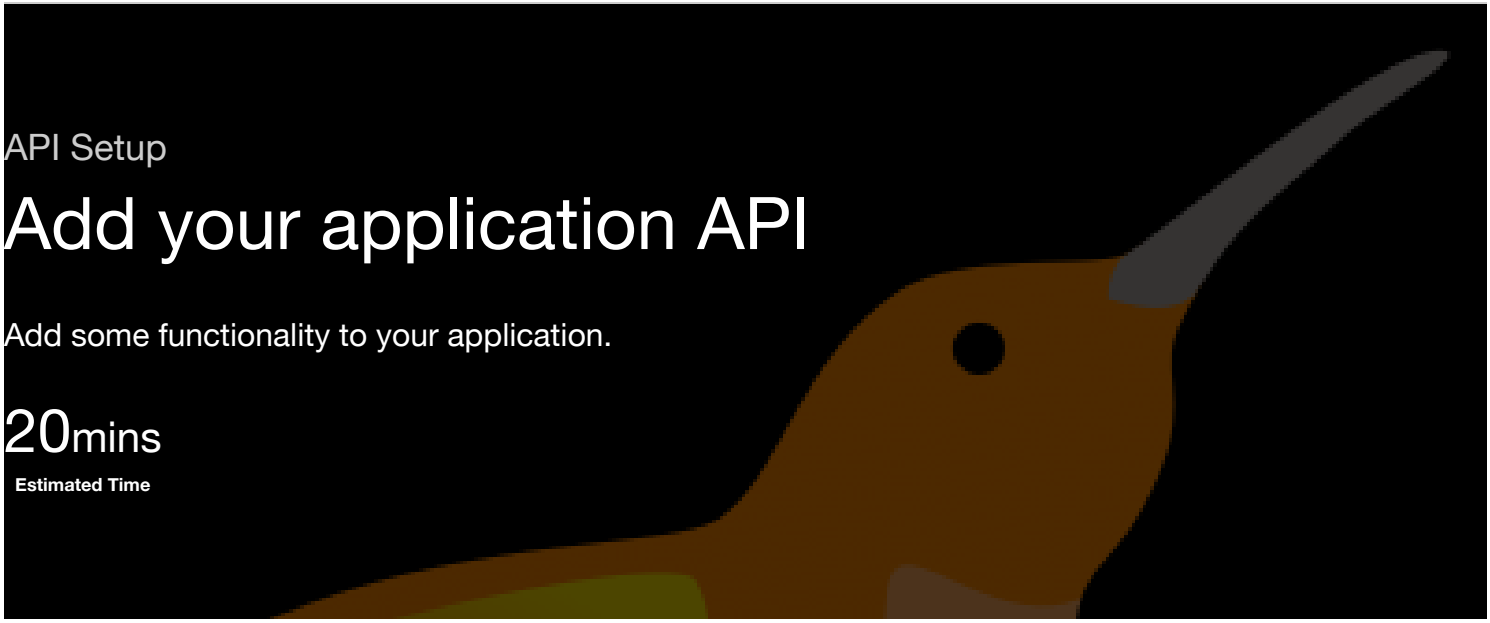

API Setup

Add your application API

Add some functionality to your application.

20mins

Estimated Time

Section 1

Todo Controller and Repository

Create a controller type that will hold the Todos API

Step 1

Add the new file `Sources/App/Controllers/TodoController.swift`.

Sources/App/Controllers/TodoController.swift

No Preview

```
1  import Hummingbird
2
3  struct TodoController {
4      // return todo endpoints
5      var endpoints: RouteCollection<AppRequestContext> {
6          return RouteCollection(context: AppRequestContext.self)
7      }
8  }
```

Step 2

Go back to `buildRouter()` in Application+build.swift. Routers ensure a Request is *routed* to the correct handler function.

Step 3

And add the TodoController endpoints to your router. The Todos API has a URI prefix of “todos”.

Step 4

We are going to use the repository design pattern to separate our storage concerns from our API. With this we should be able to create an API and test it without worrying about Database setup.

Step 5

Create a product `Todo` that includes everything to define a todo.

Step 6

We are going to use `Todo` as the return value for some of our routes, so it needs to conform to `ResponseEncodable`. This allows us to return it from our routes and have it automatically encoded as JSON. Later we will also be using it in tests so lets add `Decodable` and `Equatable` conformances, for reading JSON and comparing todos.

Step 7

Create a `TodoRepository` protocol that defines all the methods to manage todos: (get, list, create, update, delete and deleteAll). This allows us to use a different implementation of the repository for different storage methods.

Step 8

Create a concrete implementation of `TodoRepository` protocol that saves everything to memory. We use an actor because multiple tasks could be accessing the repository at the same time, and actors are thread safe.

Step 9

Return to TodoController.swift

Step 10

And add a generic repository member variable conforming to `TodoRepository` to be used by the `TodoController` routes. Generics allow us to use the same controller for different repository implementations.

Step 11

Go to `buildRouter()` in Application+build.swift

Step 12

And add the repository parameter to the TodoController initializer. We are using the memory implementation of the `TodoRepository` we have already implemented previously.

Step 13

Return to TodoController.swift. We can now start adding our endpoints. An endpoint (or Route) is a function that replies to a request if the path and method match.

Step 14

Our first endpoint is to return a Todo given an id in the URI. We extract the id from the URI, attempt to convert it to a UUID and then call the repository method `get` and return the result. The result is then converted to a response using the response encoder (JSONEncoder by default) attached to the context.

Step 15

This endpoint has a few other features. If it fails to convert the id to a UUID then it throws an `HTTPError`. This is an error that can be converted by the server to a valid HTTP response. If the server receives an error it cannot convert to an HTTP response it will return a 500 (Internal Server Error) HTTP error to the client.

If the endpoint returns nil because it could not find a todo this will automatically return a 204 (No Content) HTTP response to the client.

Step 16

Our second endpoint is to create a Todo. We have added a struct to decode from the request. In a similar way the get endpoint response uses JSONEncoder to generate its response, this uses the JSONDecoder attached to the context to read JSON from the request. We then call the repository `create` method and return the result.

Step 17

Returning an object and not a raw Response, in general sets the response status to 200 (OK). In this situation we want to return a 201 (Created) status. We can do this by returning an `EditedResponse` which can be used to edit the status code and headers of a generated response.

Step 18

We now have an API we can test. Lets use curl to create a Todo. If we include the command line parameter `-i` we get the full HTTP response and can see that the status code is 201 (Created).

Step 19

Then use curl to access the URL that was in the returned json from the previous curl call.

Cool it works!

Step 20

Continuing with adding our API endpoints

Step 21

This is the endpoint that lists all of the todos that have been created. Because `Todo` conforms to `ResponseEncodable`, `Array<Todo>` automatically conforms to `ResponseEncodable` as well.

Step 22

This is the endpoint that updates a todo. It extracts the todo id from the URI, decodes the UpdateRequest from the request and then calls the repository `update` function.

Step 23

This is the endpoint that deletes a todo. It extracts the todo id from the URI and calls the repository `delete` function.

Step 24

The final endpoint is the one that deletes all todos. It simply calls the repository `deleteAll` function, as it doesn't need to extract any information from the request.

Section 2

Testing your application with curl

Use curl to ensure your API is working as expected.

Test create and get

No Preview

```
> curl -i -X POST localhost:8080/todos -d'{"title": "Wash my hair"}'
HTTP/1.1 201 Created
```

```
Content-Type: application/json; charset=utf-8
Content-Length: 163
Date: Mon, 9 Sep 2024 11:48:06 GMT

{"completed":false,"title":"Wash my hair","id":"7BDECA4F-3A8A-49AC-A83C-

> curl -i -X POST localhost:8080/todos -d'{"title": "Brush my teeth"}'
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: 165
Date: Mon, 9 Sep 2024 11:48:11 GMT

{"id":"31B0FCCA-F084-4EB6-BCEF-002A00077549","title":"Brush my teeth","c
```

Step 1

Lets test the full API. First we are going to create two todos

Step 2

When we query the URL returned by the first curl request we should get the first Todo added.

Step 3

If we update the second Todo and set it to completed …

Step 4

when we list all the tests you will see it is now flagged as completed

Step 5

If we delete a Todo …

Step 6

when we try to get it again a 204 (No Content) response is returned, as it no longer exists.

Step 7

If we delete all of the Todos …

Step 8

when we list the Todos, the list is empty.

This is not exactly a thorough way to test your application. It is error prone and cumbersome. Move onto the next chapter to discover how you can make this process more streamlined.

Next

Testing your application

Test your application using the HummingbirdTesting framework

Get started