

How to create RESTful APIs in Kotlin with Ktor

Code example: [tutorial-server-restful-api](#)

Used plugins: [Routing](#), [Static Content](#), [Content Negotiation](#), [kotlinx.serialization](#)

In this tutorial, we'll explain how to build a backend service using Kotlin and Ktor, featuring an example of a RESTful API that generates JSON files.

In the [previous tutorial](#), we introduced you to the fundamentals of validation, error handling, and unit testing. This tutorial will expand on these topics by creating a RESTful service for managing tasks.

You will learn how to do the following:

- Create RESTful services that use JSON serialization.
- Understand the process of [Content Negotiation](#).
- Define the routes for a REST API within Ktor.

Prerequisites

You can do this tutorial independently, however, we strongly recommend that you complete the preceding tutorial to learn how to [handle requests and generate responses](#).

We recommend that you install [IntelliJ IDEA](#), but you could use another IDE of your choice.

Hello RESTful Task Manager

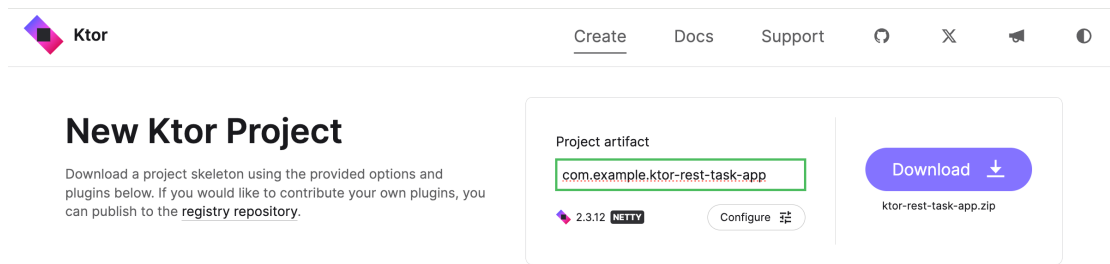
In this tutorial, you will be rewriting your existing Task Manager as a RESTful service. To do this you will use several Ktor [plugins](#).

While you could manually add it to your existing project, it's simpler to generate a new project and then incrementally add the code from the previous tutorial. You will reiterate all the code as you go, so you don't need to have the previous project to hand.

Create a new project with plugins

1. Navigate to the [Ktor Project Generator](#).

2. In the Project artifact field, enter `com.example.ktor-rest-task-app` as the name of your project artifact.




New Ktor Project

Download a project skeleton using the provided options and plugins below. If you would like to contribute your own plugins, you can publish to the [registry repository](#).

Project artifact: `com.example.ktor-rest-task-app`

2.3.12 **NETTY** [Configure](#)

[Download](#)  `ktor-rest-task-app.zip`

[Plugins](#) [Preview](#)

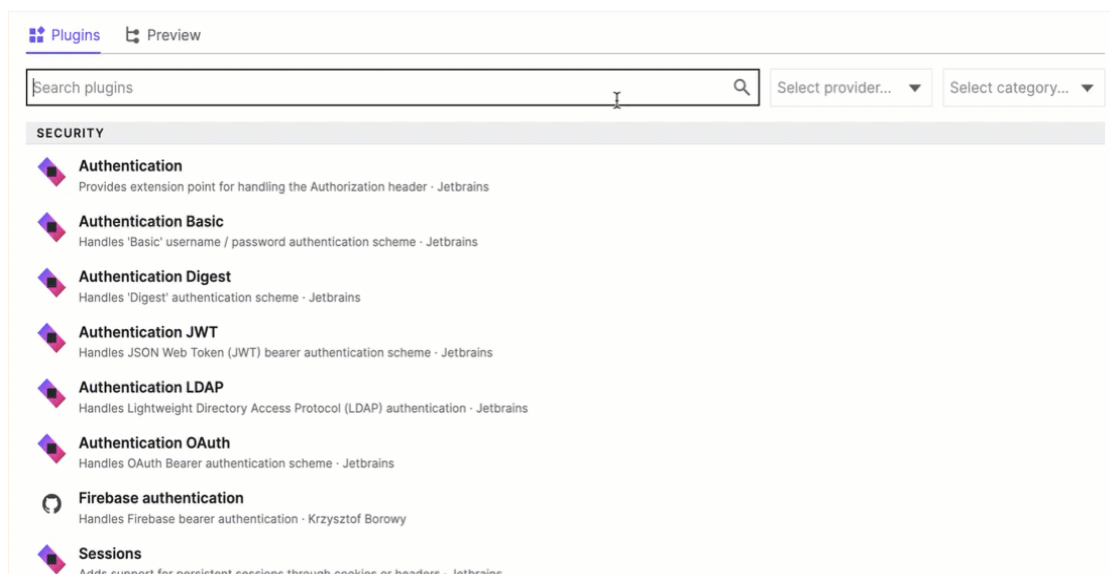
Search plugins  Select provider... Select category...

SECURITY


- Authentication**
Provides extension point for handling the Authorization header - JetBrains
- Authentication Basic**
Handles 'Basic' username / password authentication scheme - JetBrains
- Authentication Digest**
Handles 'Digest' authentication scheme - JetBrains
- Authentication JWT**
Handles JSON Web Token (JWT) bearer authentication scheme - JetBrains
- Authentication LDAP**
Handles Lightweight Directory Access Protocol (LDAP) authentication - JetBrains

3. In the plugins section search for and add the following plugins by clicking on the Add button:

- Routing
- Content Negotiation
- Kotlinx.serialization
- Static Content



[Plugins](#) [Preview](#)

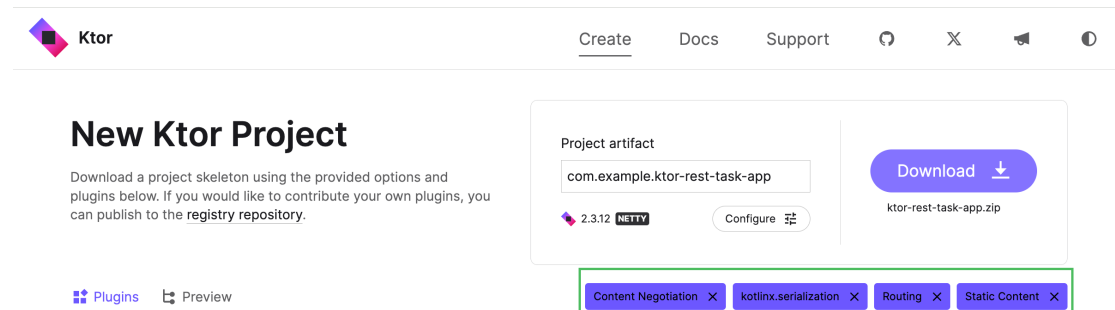
Search plugins  Select provider... Select category...

SECURITY

- Authentication**
Provides extension point for handling the Authorization header - JetBrains
- Authentication Basic**
Handles 'Basic' username / password authentication scheme - JetBrains
- Authentication Digest**
Handles 'Digest' authentication scheme - JetBrains
- Authentication JWT**
Handles JSON Web Token (JWT) bearer authentication scheme - JetBrains
- Authentication LDAP**
Handles Lightweight Directory Access Protocol (LDAP) authentication - JetBrains
- Authentication OAuth**
Handles OAuth Bearer authentication scheme - JetBrains
- Firebase authentication**
Handles Firebase bearer authentication - Krzysztof Borowy
- Sessions**
Adds support for persistent sessions through cookies or headers - JetBrains

Gif

Once you have added the plugins, you will see all four plugins listed below the project settings.



4. Click the Download button to generate and download your Ktor project.

Add starter code

1. Open your project in IntelliJ IDEA, as previously described in the [Create, open and run a Ktor project](#) tutorial.
2. Navigate to `src/main/kotlin/com/example` and create a subpackage called `model`.
3. Inside the `model` package, create a new `Task.kt` file.
4. Open the `Task.kt` file and add an enum to represent priorities and a class to represent tasks:
5. `package com.example.model`
- 6.
7. `import kotlinx.serialization.Serializable`
- 8.
9. `enum class Priority {`
10. `Low, Medium, High, Vital`
11. `}`
- 12.

```
13. @Serializable
14. data class Task(
15.     val name: String,
16.     val description: String,
17.     val priority: Priority
18. )
```

In the previous tutorial you used extension functions to convert a Task into HTML. In this case, the Task class is annotated with the [Serializable](#) type from the `kotlinx.serialization` library.

18. Open the `Routing.kt` file and replace the existing code with the implementation below:


```
19. package com.example
20.
21. import com.example.model.*
22. import io.ktor.server.application.*
23. import io.ktor.server.http.content.*
24. import io.ktor.server.response.*
25. import io.ktor.server.routing.*
26.
27. fun Application.configureRouting() {
28.     routing {
29.         staticResources("static", "static")
30.
31.         get("/tasks") {
32.             call.respond(
```

```

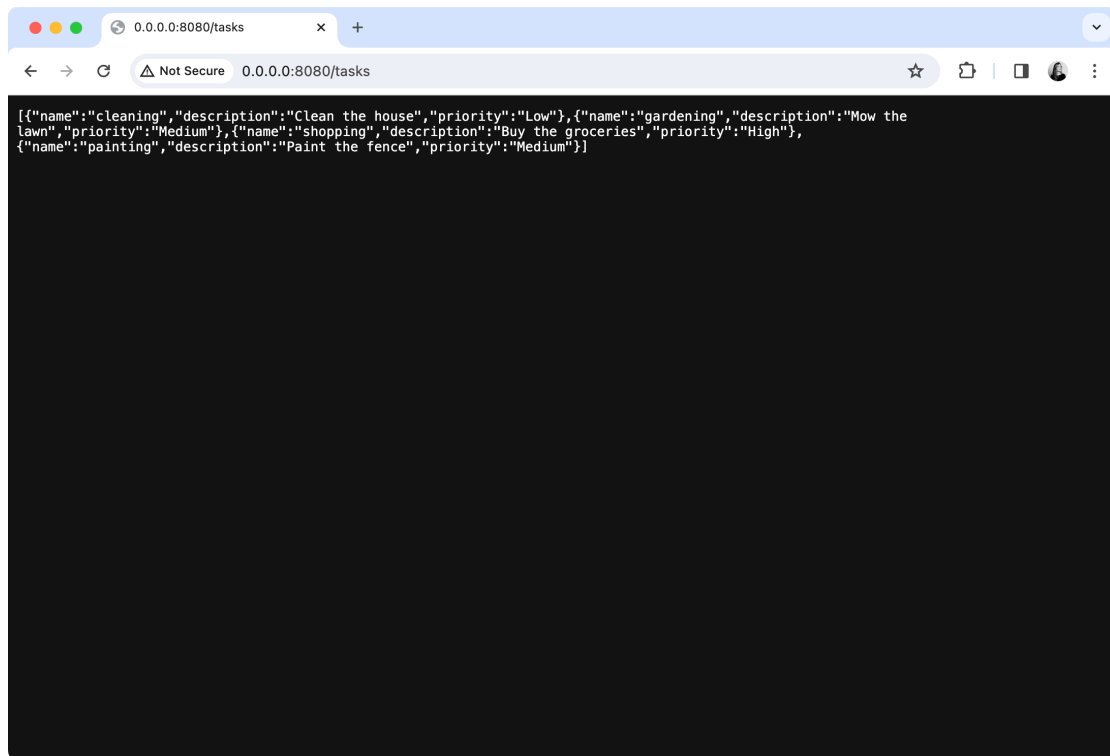
33.         listOf(
34.             Task("cleaning", "Clean the house", Priority.Low),
35.             Task("gardening", "Mow the lawn",
36.                 Priority.Medium),
37.             Task("shopping", "Buy the groceries", Priority.High),
38.             Task("painting", "Paint the fence", Priority.Medium)
39.         )
40.     }
41. }
}

```

Similar to the previous tutorial, you've created a route for GET requests to the URL `/tasks`. This time, instead of manually converting the list of tasks, you are simply returning the list.

42. In IntelliJ IDEA, click on the run button () to start the application.

43. Navigate to <http://0.0.0.0:8080/tasks> in your browser. You should see a JSON version of the list of tasks, as shown below:



Clearly a lot of work is being performed on our behalf. What exactly is going on?

Understand Content Negotiation

Content Negotiation via the browser

When you created the project you included the [Content Negotiation](#) plugin. This plugin looks at the types of content that the client can render and matches these against the content types that the current service can provide. Hence, the term *Content Negotiation*.

In HTTP the client signals which content types it can render through the Accept header. The value of this header is one or more content types. In the case above you can examine the value of this header by using the development tools built into your browser.

Consider the following example:

```
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
```

Note the inclusion of `*/*`. This header signals that it accepts HTML, XML or Images - but it would also accept any other content type.

The Content Negotiation plugin needs to find a format to send data back to the browser. If you look inside the generated code in the project you will find a file called `Serialization.kt` inside `src/kotlin/main/com/example`, which includes the following:

```
install(ContentNegotiation) {  
  
    json()  
  
}
```

This code installs the `ContentNegotiation` plugin, and also configures the `kotlinx.serialization` plugin. With this, when clients send requests the server can send back objects serialized as JSON.

In the case of the request from the browser the `ContentNegotiation` plugin knows it can only return JSON, and the browser will try to display anything it is sent. So the request succeeds.

Content Negotiation via JavaScript

In a production environment, you wouldn't want to display JSON directly in the browser. Instead, there would be JavaScript code running within the browser, which would make the request and then display the data returned as part of a Single Page Application (SPA). Typically, this kind of application would be written using a framework like [React](#), [Angular](#), or [Vue.js](#).

1. To simulate this, open the `index.html` page inside `src/main/resources/static` and replace the default content with the following:
2. `<html>`
3. `<head>`
4. `<title>A Simple SPA For Tasks</title>`
5. `<script type="application/javascript">`
6. `function fetchAndDisplayTasks() {`
7. `fetchTasks()`

```
8.         .then(tasks => displayTasks(tasks))
9.     }
10.
11.     function fetchTasks() {
12.         return fetch(
13.             "/tasks",
14.             {
15.                 headers: { 'Accept': 'application/json' }
16.             }
17.         ).then(resp => resp.json());
18.     }
19.
20.     function displayTasks(tasks) {
21.         const tasksTableBody =
            document.getElementById("tasksTableBody")
22.         tasks.forEach(task => {
23.             const newRow = taskRow(task);
24.             tasksTableBody.appendChild(newRow);
25.         });
26.     }
27.
28.     function taskRow(task) {
29.         return tr([
30.             td(task.name),
```



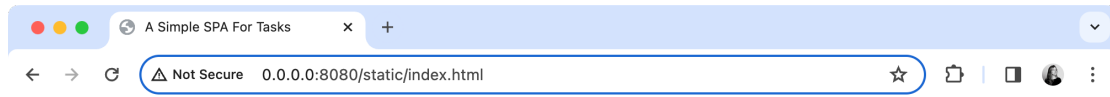
```
31.         td(task.description),
32.         td(task.priority)
33.     });
34. }
35.
36.     function tr(children) {
37.         const node = document.createElement("tr");
38.         children.forEach(child => node.appendChild(child));
39.         return node;
40.     }
41.
42.     function td(text) {
43.         const node = document.createElement("td");
44.         node.appendChild(document.createTextNode(text));
45.         return node;
46.     }
47.     </script>
48. </head>
49. <body>
50. <h1>Viewing Tasks Via JS</h1>
51. <form action="javascript:fetchAndDisplayTasks()">
52.     <input type="submit" value="View The Tasks">
53. </form>
54. <table>
```

```
55.     <thead>
56.     <tr><th>Name</th><th>Description</th><th>Priority</th></tr>
57.     </thead>
58.     <tbody id="tasksTableBody">
59.     </tbody>
60. </table>
61. </body>
</html>
```

This page contains an HTML form and an empty table. Upon submitting the form a JavaScript event handler sends a request to the /tasks endpoint, with the Accept header set to application/json. The data returned is then deserialized and added to an HTML table.

62. In IntelliJ IDEA, click the rerun button () to restart the application.

63. Navigate to the URL <http://0.0.0.0:8080/static/index.html>. You should be able to fetch the data by clicking on the View The Tasks button:



Viewing Tasks Via JS

View The Tasks

Name	Description	Priority
cleaning	Clean the house	Low
gardening	Mow the lawn	Medium
shopping	Buy the groceries	High
painting	Paint the fence	Medium

Add the GET routes

Now that you are familiar with the process of content negotiation, continue with transferring the functionality from the [previous tutorial](#) into this one.

Reuse the Task Repository

You can reuse the repository for Tasks without any modification, so let's do that first.

1. Inside the model package create a new TaskRepository.kt file.
2. Open TaskRepository.kt and add the code below:
3. `package com.example.model`
- 4.
5. `object TaskRepository {`
6. `private val tasks = mutableListOf(`
7. `Task("cleaning", "Clean the house", Priority.Low),`
8. `Task("gardening", "Mow the lawn", Priority.Medium),`

```

9.         Task("shopping", "Buy the groceries", Priority.High),
10.        Task("painting", "Paint the fence", Priority.Medium)
11.    )
12.
13.    fun allTasks(): List<Task> = tasks
14.
15.    fun tasksByPriority(priority: Priority) = tasks.filter {
16.        it.priority == priority
17.    }
18.
19.    fun taskByName(name: String) = tasks.find {
20.        it.name.equals(name, ignoreCase = true)
21.    }
22.
23.    fun addTask(task: Task) {
24.        if (taskByName(task.name) != null) {
25.            throw IllegalStateException("Cannot duplicate task
names!")
26.        }
27.        tasks.add(task)
28.    }
}

```

Reuse the routes for GET requests

Now that you've created the repository, you can implement the routes for GET requests. The previous code can be simplified because you no longer need to

worry about converting tasks to HTML:

1. Navigate to the Routing.kt file in src/main/kotlin/com/example.
2. Update the code for the /tasks route inside the Application.configureRouting() function with the following implementation:
3. package com.example
- 4.
5. import com.example.model.Priority
6. import com.example.model.TaskRepository
7. import io.ktor.http.*
8. import io.ktor.server.application.*
9. import io.ktor.server.http.content.*
10. import io.ktor.server.response.*
11. import io.ktor.server.routing.*
- 12.
13. fun Application.configureRouting() {
14. routing{
15. staticResources("static", "static")
- 16.
17. //updated implementation
18. route("/tasks") {
19. get {
20. val tasks = TaskRepository.allTasks()
21. call.respond(tasks)
22. }

```
23.
24.         get("/byName/{taskName}") {
25.             val name = call.parameters["taskName"]
26.             if (name == null) {
27.                 call.respond(HttpStatusCode.BadRequest)
28.                 return@get
29.             }
30.
31.             val task = TaskRepository.taskByName(name)
32.             if (task == null) {
33.                 call.respond(HttpStatusCode.NotFound)
34.                 return@get
35.             }
36.             call.respond(task)
37.         }
38.         get("/byPriority/{priority}") {
39.             val priorityAsText = call.parameters["priority"]
40.             if (priorityAsText == null) {
41.                 call.respond(HttpStatusCode.BadRequest)
42.                 return@get
43.             }
44.             try {
45.                 val priority = Priority.valueOf(priorityAsText)
46.                 val tasks = TaskRepository.tasksByPriority(priority)
```

```

47.
48.                if (tasks.isEmpty()) {
49.                    call.respond(HttpStatusCode.NotFound)
50.                    return@get
51.                }
52.                call.respond(tasks)
53.            } catch (ex: IllegalArgumentException) {
54.                call.respond(HttpStatusCode.BadRequest)
55.            }
56.        }
57.    }
58. }
}

```

With this, your server can respond to the following GET requests:

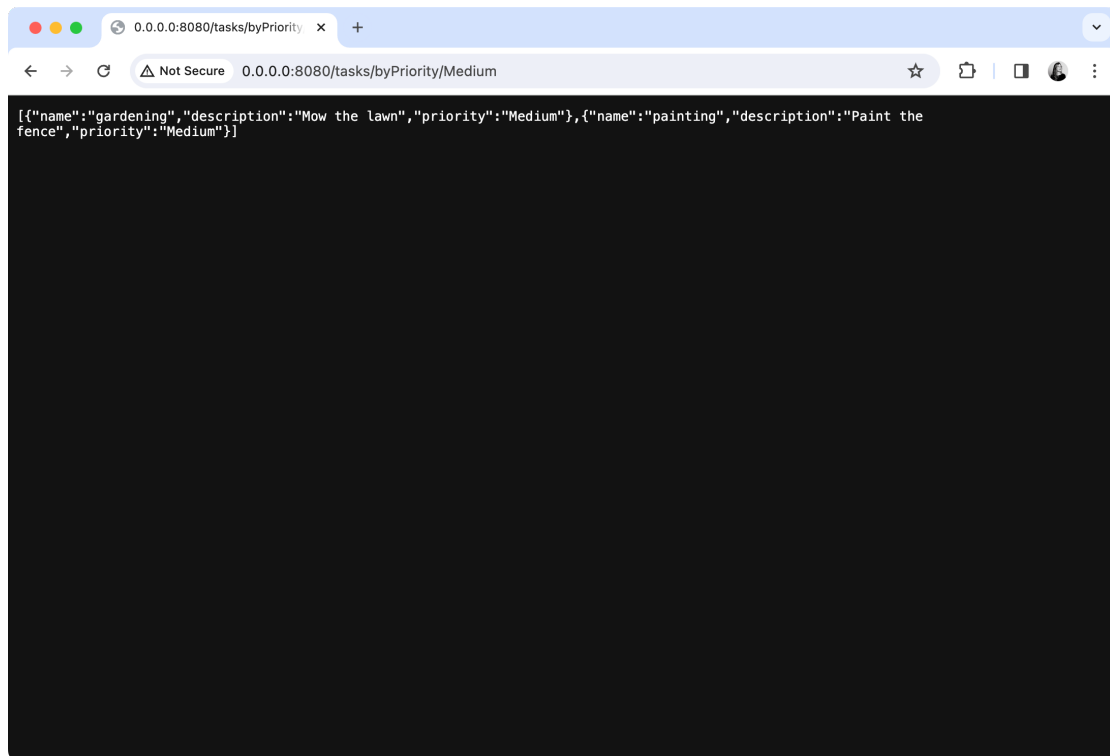
- /tasks returns all tasks in the repository.
- /tasks/byName/{taskName} returns tasks filtered by the specified taskName.
- /tasks/byPriority/{priority} returns tasks filtered by the specified priority.

59. In IntelliJ IDEA, click the rerun button () to restart the application.

Test the functionality

Use the browser

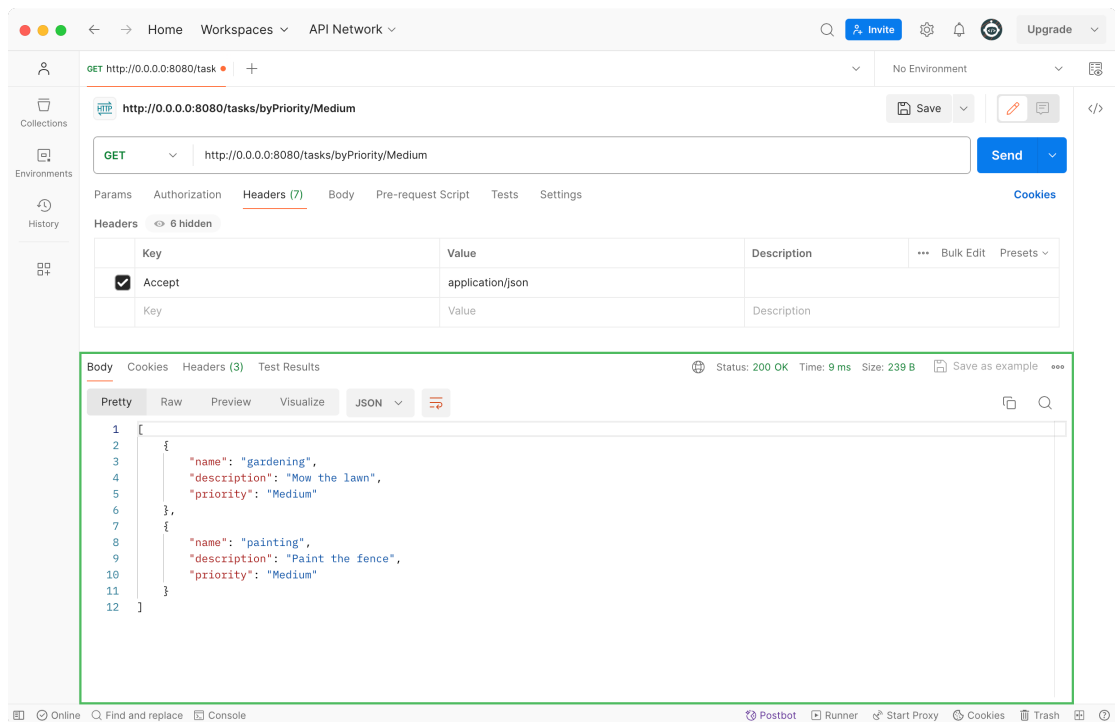
You could test these routes in the browser. For example, navigate to <http://0.0.0.0:8080/tasks/byPriority/Medium> to see all the tasks with a Medium priority displayed in JSON format:



Given that these kinds of requests will typically be coming from JavaScript, more fine-grained testing is preferable. For this, you can use a specialist tool such as [Postman](#).

Use Postman

1. In Postman, create a new GET request with the URL `http://0.0.0.0:8080/tasks/byPriority/Medium`.
2. In the Headers pane, set the value of the Accept header to `application/json`.
3. Click Send to send the request and see the response in the response viewer.



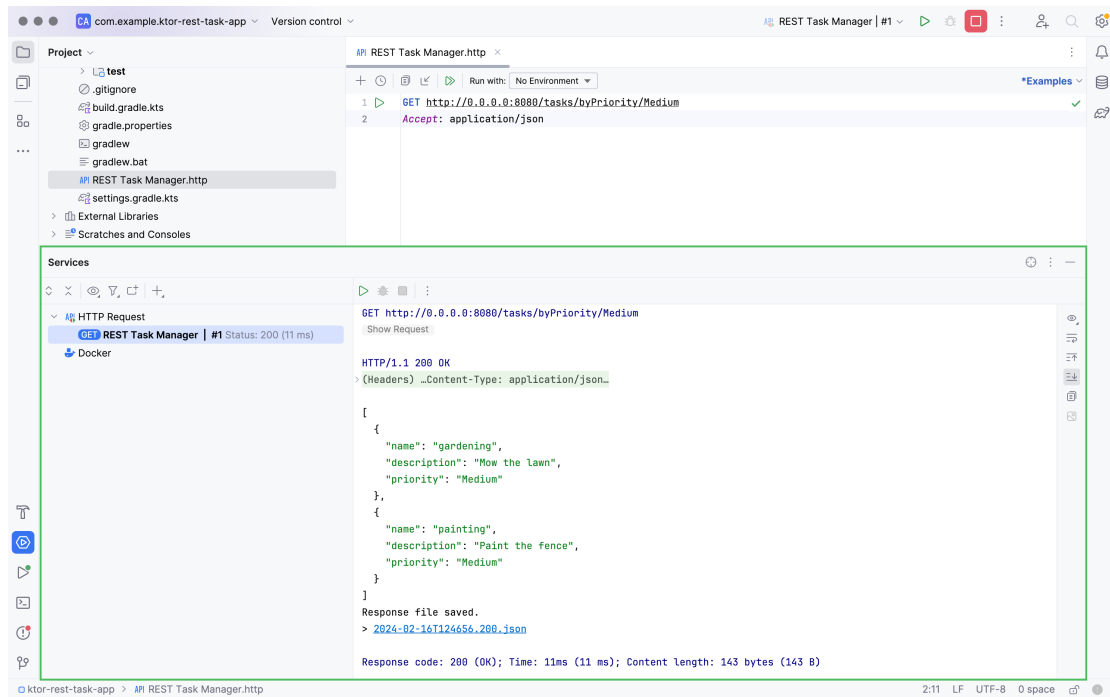
Use an HTTP Request File

Within IntelliJ IDEA Ultimate you could perform the same steps in an HTTP Request file.

1. In the project root directory, create a new REST Task Manager.http file.
2. Open the REST Task Manager.http file and add the following GET request:
3. GET http://0.0.0.0:8080/tasks/byPriority/Medium

Accept: application/json

4. To send the request within IntelliJ IDE, click on the gutter icon () next to it.
5. This will open and run in the Services tool window:



Another way to test the routes can be to use the [khttp](#) library from within a Kotlin Notebook.

Add a route for POST requests

In the previous tutorial, tasks were created through an HTML form. However, as you are now building a RESTful service, you no longer need to do that. Instead, you will make use of the `kotlinx.serialization` framework which will do the majority of the heavy lifting.

1. Open the `Routing.kt` file inside `src/main/kotlin/com/example`.
2. Add a new POST route to the `Application.configureRouting()` function as follows:
3. `//...`
- 4.
5. `fun Application.configureRouting() {`
6. `routing{`
7. `//...`
- 8.

```

9.         route("/tasks") {
10.             //...
11.
12.             //add the following new route
13.             post {
14.                 try {
15.                     val task = call.receive<Task>()
16.                     TaskRepository.addTask(task)
17.                     call.respond(HttpStatusCode.Created)
18.                 } catch (ex: IllegalStateException) {
19.                     call.respond(HttpStatusCode.BadRequest)
20.                 } catch (ex: JsonConvertException) {
21.                     call.respond(HttpStatusCode.BadRequest)
22.                 }
23.             }
24.         }
25.     }
}

```

Add the following new imports:

```
//...
```

```
import com.example.model.Task
```

```
import io.ktor.serialization.*
```

```
import io.ktor.server.request.*
```

When a POST request is sent to /tasks the kotlinx.serialization framework is used

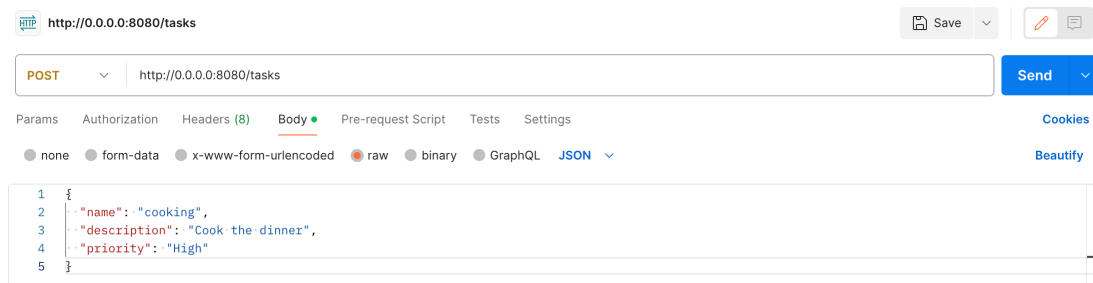
to convert the body of the request into a Task object. If this succeeds, the task will be added to the repository. If the deserialization process fails the server will need to handle a `JsonConvertException`, whereas if the task is a duplicate it will need to handle an `IllegalStateException`.

26. Restart the application.

27. To test this functionality in Postman, create a new POST request to the URL <http://0.0.0.0:8080/tasks>.

28. In the Body pane add the following JSON document to represent a new task:

```
29. {  
30.     "name": "cooking",  
31.     "description": "Cook the dinner",  
32.     "priority": "High"  
}
```



33. Click Send to send the request.

34. You can verify the task has been added by sending a GET request to <http://0.0.0.0:8080/tasks>.

35. Within IntelliJ IDEA Ultimate you could perform the same steps by adding the following to your HTTP Request file:

36. ###

37.

38. POST http://0.0.0.0:8080/tasks

39. Content-Type: application/json

40.

41. {

42. "name": "cooking",

43. "description": "Cook the dinner",

44. "priority": "High"

}

Add support for removals

You have almost finished adding the basic operations to your service. These are often summarized as the CRUD operations - short for Create, Read, Update, and Delete. The only operation you are missing is the Delete.

1. In the TaskRepository.kt file add the following method within the TaskRepository object to remove tasks based on their name:

2. fun removeTask(name: String): Boolean {

3. return tasks.removeIf { it.name == name }

}

4. Open the Routing.kt file and add an endpoint into the routing() function to handle DELETE requests:

5. fun Application.configureRouting() {

6. //...

7.

8. routing {

9. route("/tasks") {

10. //...

```

11.          //add the following function
12.          delete("/{taskName}") {
13.              val name = call.parameters["taskName"]
14.              if (name == null) {
15.                  call.respond(HttpStatusCode.BadRequest)
16.                  return@delete
17.              }
18.
19.              if (TaskRepository.removeTask(name)) {
20.                  call.respond(HttpStatusCode.NoContent)
21.              } else {
22.                  call.respond(HttpStatusCode.NotFound)
23.              }
24.          }
25.      }
26.  }
}

```

27. Restart the application.

28. Add the following DELETE request to your HTTP Request File:

29. ###

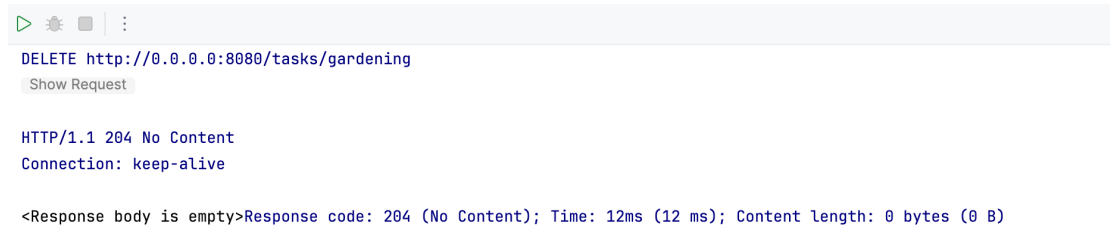
30.

DELETE http://0.0.0.0:8080/tasks/gardening

31. To send the DELETE request within IntelliJ IDE, click on the gutter icon

() next to it.

32. You will see the response in the Services tool window:



```
DELETE http://0.0.0.0:8080/tasks/gardening
Show Request

HTTP/1.1 204 No Content
Connection: keep-alive

<Response body is empty>Response code: 204 (No Content); Time: 12ms (12 ms); Content length: 0 bytes (0 B)
```

Create unit tests with Ktor Client

So far you have tested your application manually, but, as you've already noticed, this approach is time-consuming and will not scale. Instead, you can implement [JUnit tests](#), using the built-in client object to fetch and deserialize JSON.

1. Open the ApplicationTest.kt file within src/test/kotlin/com/example.
2. Replace the contents of the ApplicationTest.kt file with the following:
3. package com.example
- 4.
5. import com.example.model.Priority
6. import com.example.model.Task
7. import io.ktor.client.call.*
8. import io.ktor.client.plugins.contentnegotiation.*
9. import io.ktor.client.request.*
10. import io.ktor.http.*
11. import io.ktor.serialization.kotlinx.json.*
12. import io.ktor.server.testing.*

```
13. import kotlin.test.*

14.

15. class ApplicationTest {

16.     @Test

17.     fun tasksCanBeFoundByPriority() = testApplication {

18.         application {

19.             module()

20.         }

21.         val client = createClient {

22.             install(ContentNegotiation) {

23.                 json()

24.             }

25.         }

26.

27.         val response = client.get("/tasks/byPriority/Medium")

28.         val results = response.body<List<Task>>()

29.

30.         assertEquals(HttpStatusCode.OK, response.status)

31.

32.         val expectedTaskNames = listOf("gardening", "painting")

33.         val actualTaskNames = results.map(Task::name)

34.         assertEquals(expectedTaskNames, actualTaskNames)

35.     }

36.
```



```
37.     @Test
38.     fun invalidPriorityProduces400() = testApplication {
39.         application {
40.             module()
41.         }
42.         val response = client.get("/tasks/byPriority/Invalid")
43.         assertEquals(HttpStatusCode.BadRequest, response.status)
44.     }
45.
46.
47.     @Test
48.     fun unusedPriorityProduces404() = testApplication {
49.         application {
50.             module()
51.         }
52.         val response = client.get("/tasks/byPriority/Vital")
53.         assertEquals(HttpStatusCode.NotFound, response.status)
54.     }
55.
56.     @Test
57.     fun newTasksCanBeAdded() = testApplication {
58.         application {
59.             module()
60.         }
```

```
61.         val client = createClient {
62.             install(ContentNegotiation) {
63.                 json()
64.             }
65.         }
66.
67.         val task = Task("swimming", "Go to the beach", Priority.Low)
68.         val response1 = client.post("/tasks") {
69.             header(
70.                 HttpHeaders.ContentType,
71.                 ContentType.Application.Json
72.             )
73.
74.             setBody(task)
75.         }
76.         assertEquals(HttpStatusCode.Created, response1.status)
77.
78.         val response2 = client.get("/tasks")
79.         assertEquals(HttpStatusCode.OK, response2.status)
80.
81.         val taskNames = response2
82.             .body<List<Task>>()
83.             .map { it.name }
84.
```

```
85.         assertContains(taskNames, "swimming")
86.     }
}
```

Note that you need to install the ContentNegotiation and kotlinx.serialization plugins into the [Ktor client](#), in the same way as you did on the server.

87. Add the following dependency into your version catalog located in gradle/libs.versions.toml:

```
88. [libraries]
89. # ...
```

```
ktor-client-content-negotiation = { module = "io.ktor:ktor-client-content-
negotiation", version.ref = "ktor-version" }
```

90. Add the new dependency into your build.gradle.kts file:

```
testImplementation(libs.ktor.client.content.negotiation)
```

Create unit tests with JsonPath

Testing your service with the Ktor client, or a similar library, is convenient, but it has a drawback from a Quality Assurance (QA) perspective. The server, not directly handling JSON, can't be certain about its assumptions regarding the JSON structure.

For example, assumptions like:

- Values are being stored in an array when in reality an object is used.
- Properties are being stored as numbers, when they are in fact strings.
- Members are being serialized in the order of declaration when they are not.

If your service is intended for use by multiple clients, it's crucial to have confidence in the JSON structure. To achieve this, use the Ktor Client to retrieve text from the server and then analyze this content using the [JSONPath](#) library.

1. In your build.gradle.kts file, add the JSONPath library to the dependencies block:

```
testImplementation("com.jayway.jsonpath:json-path:2.9.0")
```

2. Navigate to the src/test/kotlin/com/example folder and create a new ApplicationJsonPathTest.kt file.
3. Open the ApplicationJsonPathTest.kt file and add the following content to it:

4. package com.example

- 5.

6. import com.jayway.jsonpath.DocumentContext

7. import com.jayway.jsonpath.JsonPath

8. import io.ktor.client.*

9. import com.example.model.Priority

10. import io.ktor.client.request.*

11. import io.ktor.client.statement.*

12. import io.ktor.http.*

13. import io.ktor.server.testing.*

14. import kotlin.test.*

- 15.

- 16.

17. class ApplicationJsonPathTest {

18. @Test

19. fun tasksCanBeFound() = testApplication {

20. application {

21. module()

```
22.         }
23.         val jsonDoc = client.getAsJsonPath("/tasks")
24.
25.         val result: List<String> = jsonDoc.read("[$*].name")
26.         assertEquals("cleaning", result[0])
27.         assertEquals("gardening", result[1])
28.         assertEquals("shopping", result[2])
29.     }
30.
31.     @Test
32.     fun tasksCanBeFoundByPriority() = testApplication {
33.         application {
34.             module()
35.         }
36.         val priority = Priority.Medium
37.         val jsonDoc = client.getAsJsonPath("/tasks/byPriority/$priority")
38.
39.         val result: List<String> =
40.             jsonDoc.read("[$?(@.priority == '$priority')].name")
41.         assertEquals(2, result.size)
42.
43.         assertEquals("gardening", result[0])
44.         assertEquals("painting", result[1])
45.     }
```

```

46.
47.     suspend fun HttpClient.getAsJsonPath(url: String):
        DocumentContext {
48.         val response = this.get(url) {
49.             accept(ContentType.Application.Json)
50.         }
51.         return JsonPath.parse(response.bodyAsText())
52.     }
}

```

The JsonPath queries work as follows:

- `[$*].name` means “treat the document as an array and return the value of the name property of each entry”.
- `[$[?(@.priority == '$priority')].name` means “return the value of the name property of every entry in the array with a priority equal to the supplied value”.

You can use queries like these to confirm your understanding of the returned JSON. When you do code refactoring and service redeployment, any modifications in serialization will be identified, even if they don't disrupt deserialization with the current framework. This allows you to republish publicly available APIs with confidence.

Next steps

Congratulations! You have now completed creating a RESTful API service for your Task Manager application and learned the nits and grits of unit testing with the Ktor Client and JsonPath.