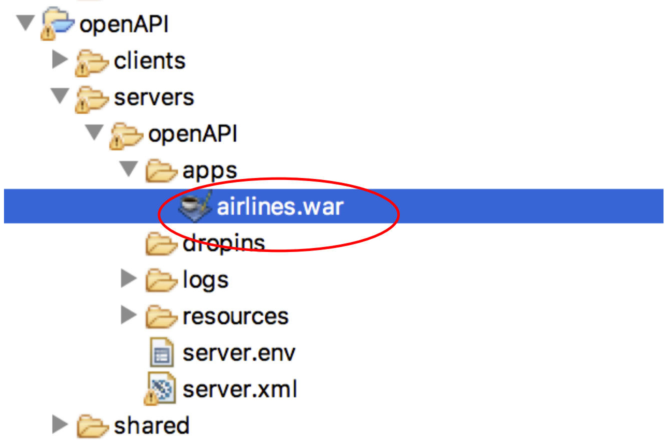
Lab 3: Using OpenAPI Programming Models

Introduction

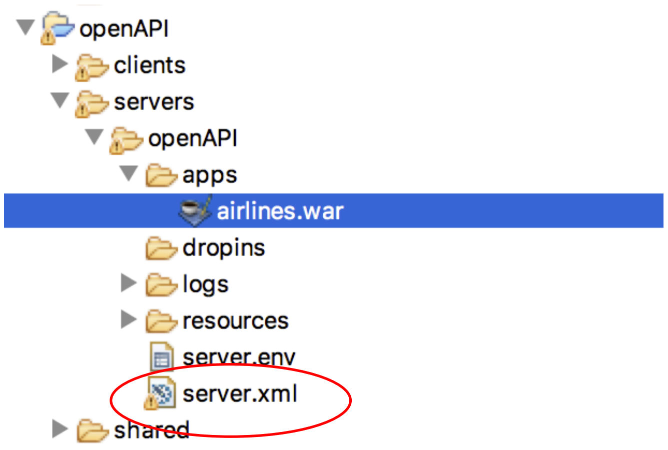
The purpose of this lab is to show you how to document REST APIs using the OpenAPI programming models. The application for which we will be creating REST APIs for is a simple Tasks app where a user can create and manage their everyday tasks. You will learn how to document your REST APIs using programming interfaces, and deploy it to a Liberty server.

3.1 Create a Liberty server to host the Tasks Application

1. Start Eclipse
    a. Start Eclipse
2. If you have not successfully completed Lab 1, skip ahead to step **d.**
    a. In the **Package Explorer** view, delete the **airlines.war** file from the apps folder in the **openAPI** server, by right-clicking the file and clicking **Delete**

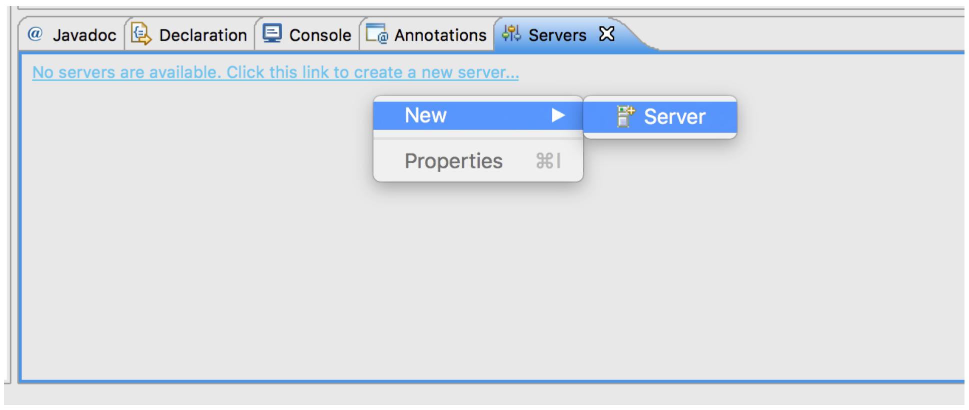


b. Also, delete the server.xml file from the openAPI server.



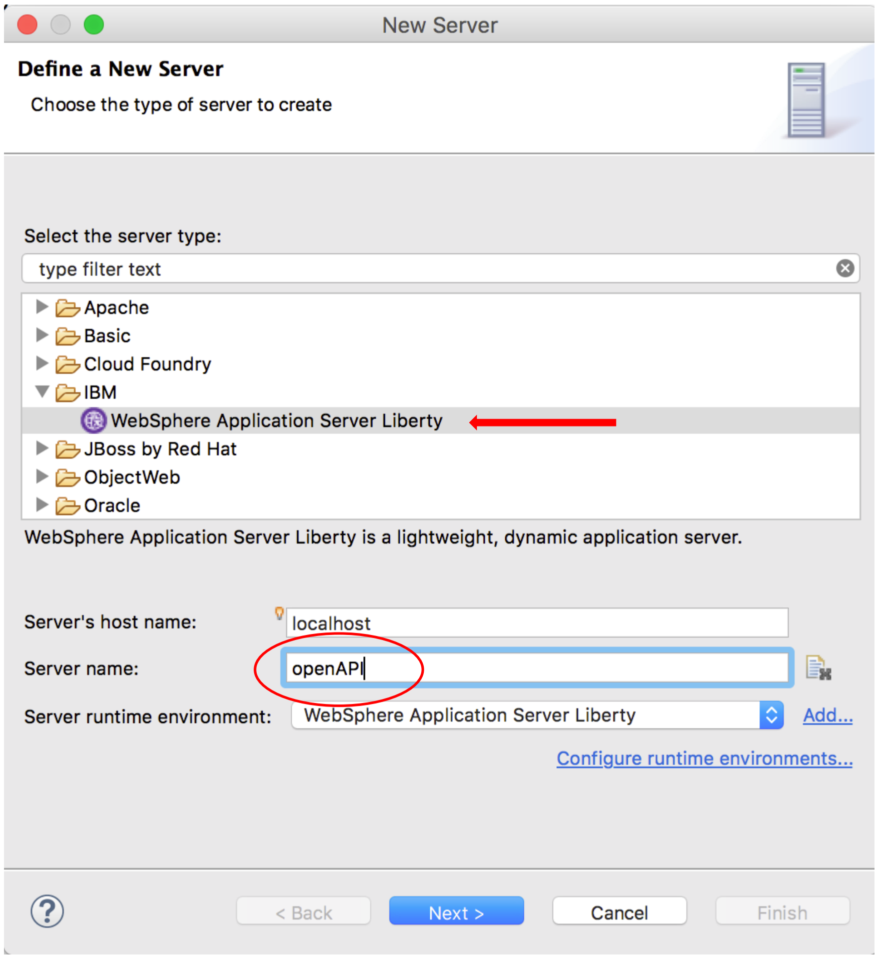
c. Skip ahead to step **3.**

d. Open the Servers view, and create a new server



No servers are available. Click this link to create a new server...

e. Create a new "**WebSphere Application Server Liberty**" server representation with the name `openAPI`. Hit **Next**.



Define a New Server

Choose the type of server to create

Select the server type:

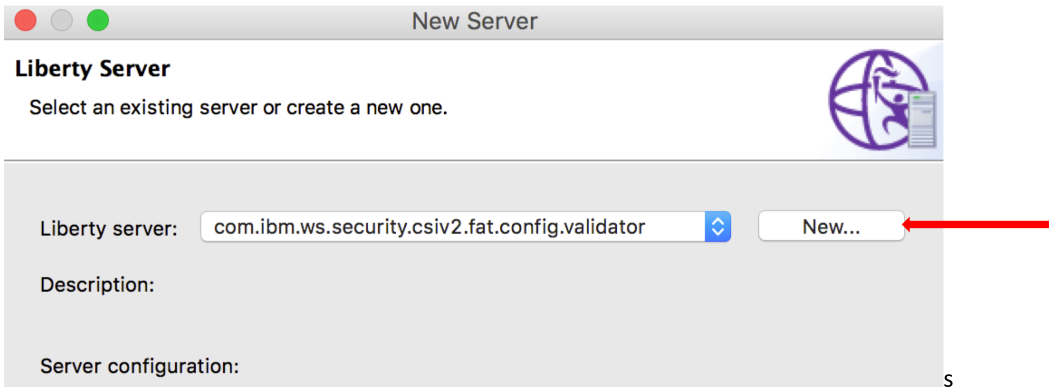
WebSphere Application Server Liberty is a lightweight, dynamic application server.

Server's host name: localhost

Server name: openAPI

Server runtime environment: WebSphere Application Server Liberty

f. Click **New…** to create a new Liberty server instance.



Liberty Server

Select an existing server or create a new one.

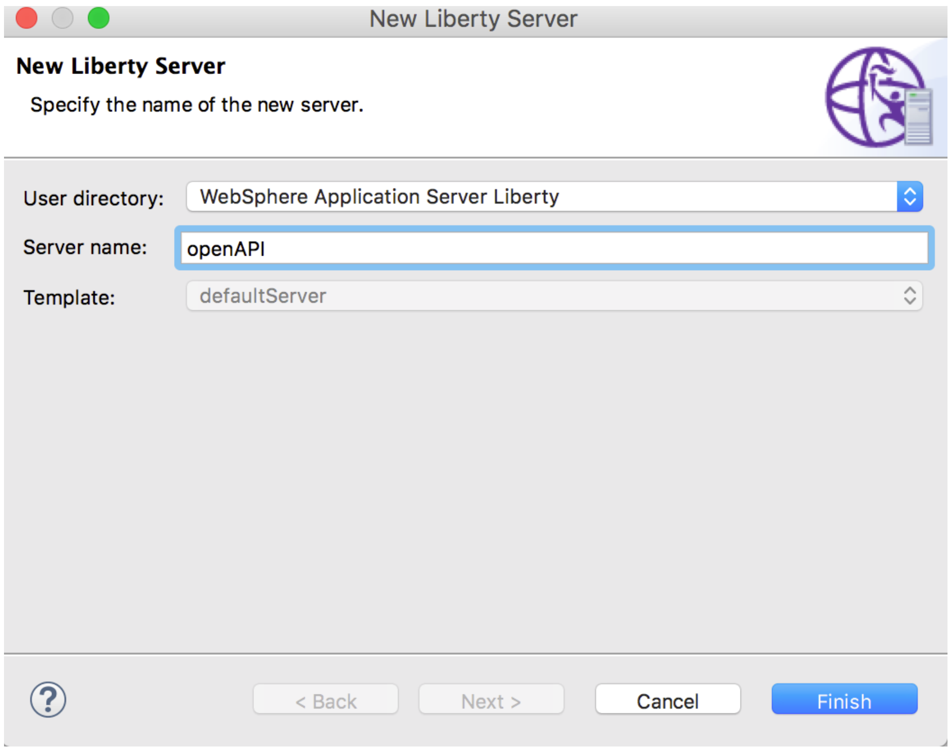
Liberty server: com.ibm.ws.security.csiv2.fat.config.validator

Description:

Server configuration:

s

g. Specify the server name `openAPI` and click **Finish**. And click **Finish** again.



New Liberty Server

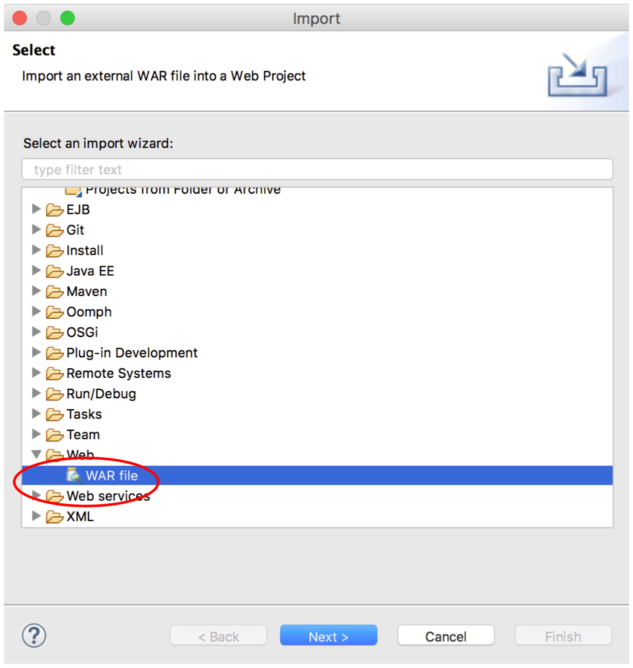
Specify the name of the new server.

User directory: WebSphere Application Server Liberty

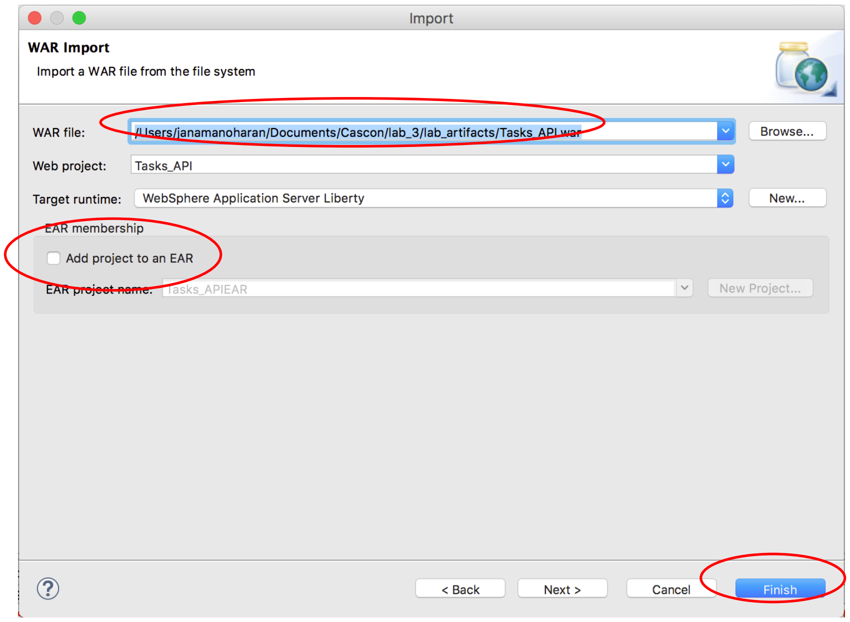
Server name: openAPI

Template: defaultServer

3. Import the Task project using the war file
   a. Right-click anywhere in the **Package Explorer** view, and click Import -> Web -> WAR File



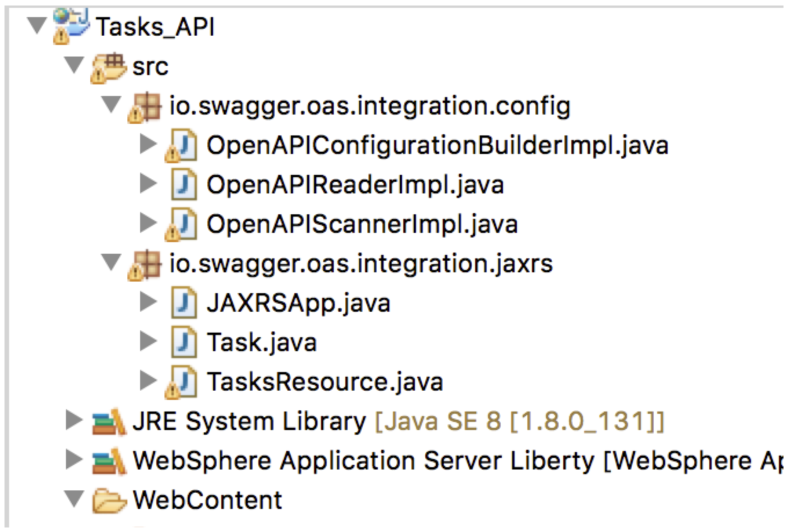
   a. Choose the **Task_API.war** file from the lab artifacts. Make sure the ""Add project to an EAR" (Enterprise Archive) is unchecked and hit **Finish**.



WAR file: /Users/janamanoharan/Documents/Cascon/lab_3/lab_artifacts/Tasks_API.war
Web project: Tasks_API
Target runtime: WebSphere Application Server Liberty
☐ Add project to an EAR
EAR project name: Tasks_APIEAR

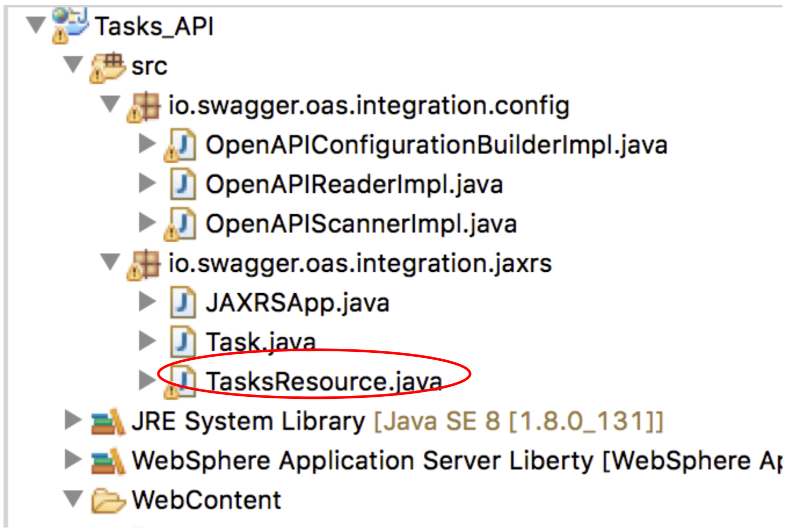
3.2 Explore the Tasks_API implementation and the REST APIs

1. Expand the packages in the **src** folder, you will see that the Tasks REST APIs is documented using both JAX-RS annotations and programming models. This application demonstrates how you can use the **OpenAPIConfigurationBuilder** interface to provide the OpenAPI model along with the JAX-RS annotations.



```
▼ Tasks_API
  ▼ src
    ▼ io.swagger.oas.integration.config
      ▶ OpenAPIConfigurationBuilderImpl.java
      ▶ OpenAPIReaderImpl.java
      ▶ OpenAPIScannerImpl.java
    ▼ io.swagger.oas.integration.jaxrs
      ▶ JAXRSApp.java
      ▶ Task.java
      ▶ TasksResource.java
  ▶ JRE System Library [Java SE 8 [1.8.0_131]]
  ▶ WebSphere Application Server Liberty [WebSphere Ap
  ▼ WebContent
```

2. Open the **TasksResources.java.**



Tasks_API
src
io.swagger.oas.integration.config
OpenAPIConfigurationBuilderImpl.java
OpenAPIReaderImpl.java
OpenAPIScannerImpl.java
io.swagger.oas.integration.jaxrs
JAXRSApp.java
Task.java
TasksResource.java
JRE System Library [Java SE 8 [1.8.0_131]]
WebSphere Application Server Liberty [WebSphere Ap
WebContent

a. In this class, a GET method, and a DELETE method are defined. The GET method will retrieve all the tasks and the DELETE method will delete a certain task given the id. OpenAPI annotations are used to document the description, the parameters, and the types of responses for each operation.

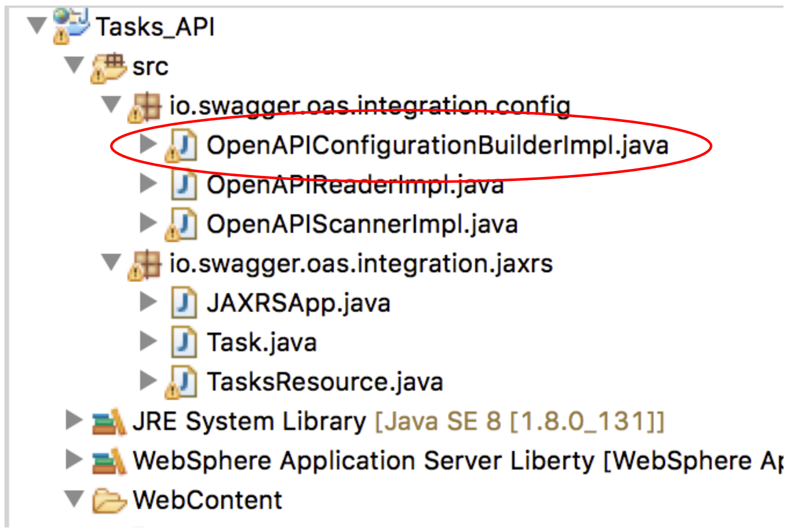
GET /tasks

```
@GET
@Operation(
        description = "get a list of all the tasks",
        responses = {
                @ApiResponse(
                        responseCode = "200",
                        content = @Content(
                                mediaType = "application/json",
                                schema = @Schema(
                                        implementation = Task.class
                                        )))
        }
        )
public Response getTasks() {
```

DELETE /tasks

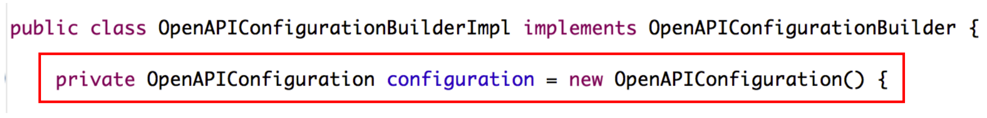
```
@DELETE
@Path("{id}")
@Operation(
        description = "delete a task from the list of tasks",
        parameters = {
            @Parameter(
                    required = true,
                    description = "the id of the task to delete",
                    in = "path",
                    schema = @Schema(
                            type = "integer",
                            format = "int64")
                    )
        },
        responses = {
                @ApiResponse(
                        responseCode = "200",
                        description = "successfully deleted the task",
                        content = @Content(
                                mediaType = "application/json",
                                schema = @Schema(
                                        description = "the deleted task",
                                        implementation = Task.class))),
                @ApiResponse(
                        responseCode = "404",
                        description = "task not found",
                        content = @Content(
                                schema = @Schema(
                                        example = "-1",
                                        type = "integer",
                                        format = "int64")))

        }
        )
public Response deleteTask(int id) {
```

3. Open the **OpenAPIConfigurationBuilderImpl.java** file



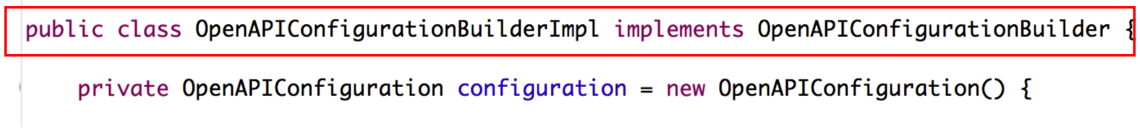
```
▼ Tasks_API
  ▼ src
    ▼ io.swagger.oas.integration.config
      ▶ OpenAPIConfigurationBuilderImpl.java
      ▶ OpenAPIReaderImpl.java
      ▶ OpenAPIScannerImpl.java
    ▼ io.swagger.oas.integration.jaxrs
      ▶ JAXRSApp.java
      ▶ Task.java
      ▶ TasksResource.java
  ▶ JRE System Library [Java SE 8 [1.8.0_131]]
  ▶ WebSphere Application Server Liberty [WebSphere Ap
  ▼ WebContent
```

a. The **OpenAPIConfiguration** is the container interface in which you can set all the OpenAPI processing parameters. This means that you can use this interface to provide an OpenAPI model, configure the annotation scanning, and set a custom reader and scanner.



```
public class OpenAPIConfigurationBuilderImpl implements OpenAPIConfigurationBuilder {
    private OpenAPIConfiguration configuration = new OpenAPIConfiguration() {
```

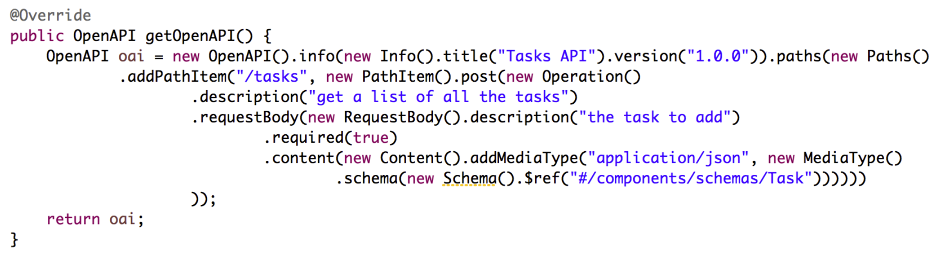
The **OpenAPIConfigurationBuilder** is the main service that enables customization of the OpenAPI V3 processing in Liberty. You can use the **OpenAPIConfigurationBuilder** to receive framework-dependent environment variables that it processes when it builds a new **OpenAPIConfiguration** object.



```
public class OpenAPIConfigurationBuilderImpl implements OpenAPIConfigurationBuilder {
    private OpenAPIConfiguration configuration = new OpenAPIConfiguration() {
```

b. In this app, the OpenAPIConfigurationBuilder is used to build an OpenAPI model. This model adds to what is already documented through the JAX-RS annotations, by including a POST method which will add a task to the list of tasks

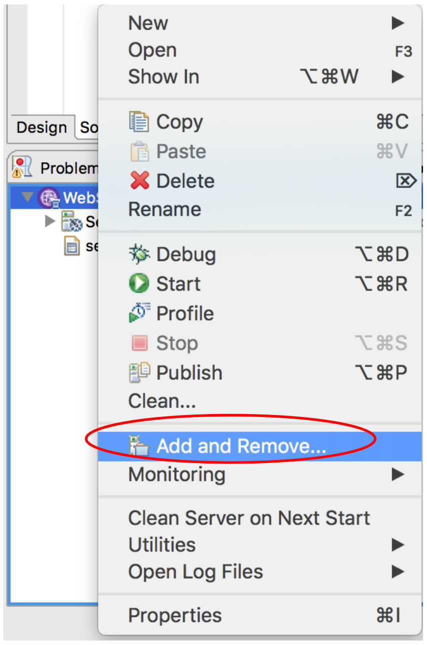
   i. This model has the documentation for the POST `/tasks` endpoint and it documents a description and how the request body for the POST request should be.



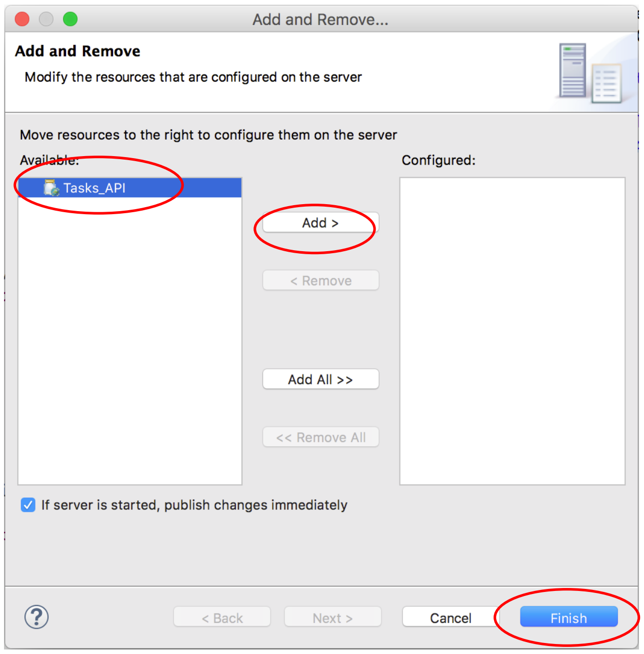
```
@Override
public OpenAPI getOpenAPI() {
    OpenAPI oai = new OpenAPI().info(new Info().title("Tasks API").version("1.0.0")).paths(new Paths()
            .addPathItem("/tasks", new PathItem().post(new Operation()
                    .description("get a list of all the tasks")
                    .requestBody(new RequestBody().description("the task to add")
                            .required(true)
                            .content(new Content().addMediaType("application/json", new MediaType()
                                    .schema(new Schema().$ref("#/components/schemas/Task"))))))
                    ));
    return oai;
}
```

4. Deploy the **Tasks_API** app on the Liberty Server and start the server

   a. On the **Servers** view, right-click the openAPI server and click **Add and Remove…**



b. Select the **Tasks_API** application from the Available section and click **Add** to deploy it into the server and hit **Finish**



Add and Remove

Modify the resources that are configured on the server

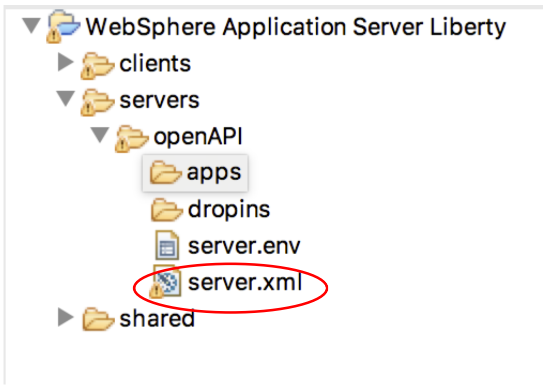
Move resources to the right to configure them on the server

Tasks_API

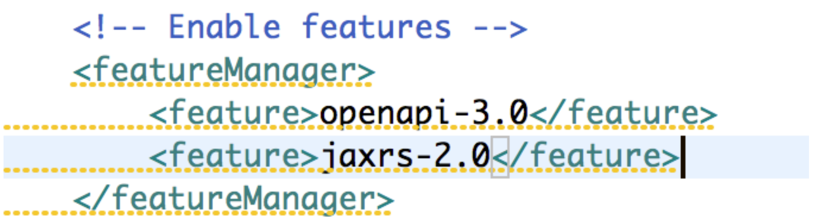
☑ If server is started, publish changes immediately

4. Copy lab artifacts to the `openAPI` server configuration. This lab relies on a pre-existing server configurations.

a. Copy the **server.xml** from location directory to **…/wlp/usr/servers/openAPI** directory and ***replace*** the existing **server.xml**

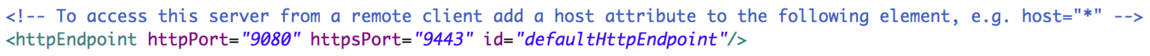


5. Understand the **server.xml** file
    a. The `jaxrs-2.0` feature allows us to support JAX-RS programming model and make http requests to our remote REST web services
    b. The `openapi-3.0` feature allows us to document our REST APIs for the airlines app in an organized manner



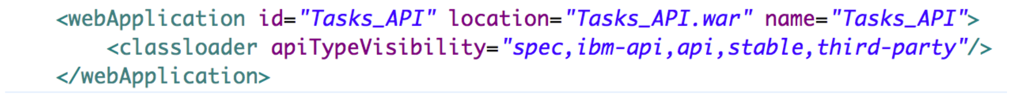
```
<!-- Enable features -->
<featureManager>
    <feature>openapi-3.0</feature>
    <feature>jaxrs-2.0</feature>
</featureManager>
```

    c. The port that we will be using will be 9080 and 9443



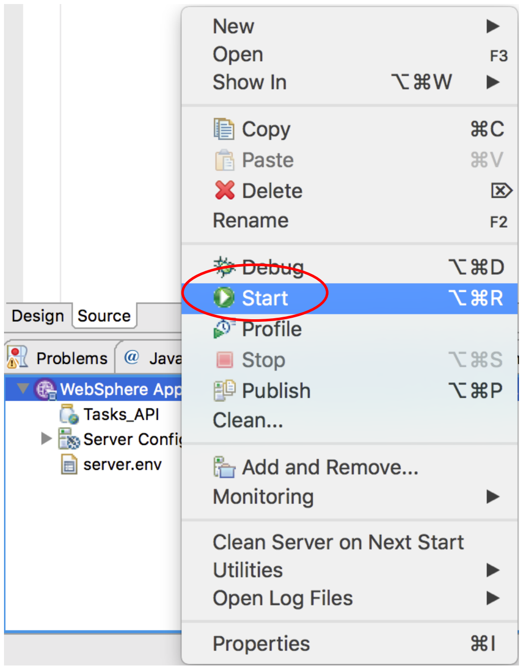
```
<!-- To access this server from a remote client add a host attribute to the following element, e.g. host="*" -->
<httpEndpoint httpPort="9080" httpsPort="9443" id="defaultHttpEndpoint"/>
```

    d. The `openAPI` server will be hosting the Tasks application by monitoring the **Tasks_API.war** file in the **apps** folder.
    e. By including the `third-party` value in `apiTypeVisibility`, we can now use third party annotations to document our REST APIs



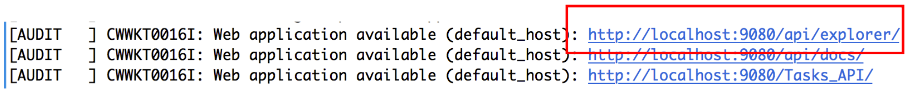
```
<webApplication id="Tasks_API" location="Tasks_API.war" name="Tasks_API">
    <classloader apiTypeVisibility="spec,ibm-api,api,stable,third-party"/>
</webApplication>
```

c. Right-click the server again and click **Start** to start the server



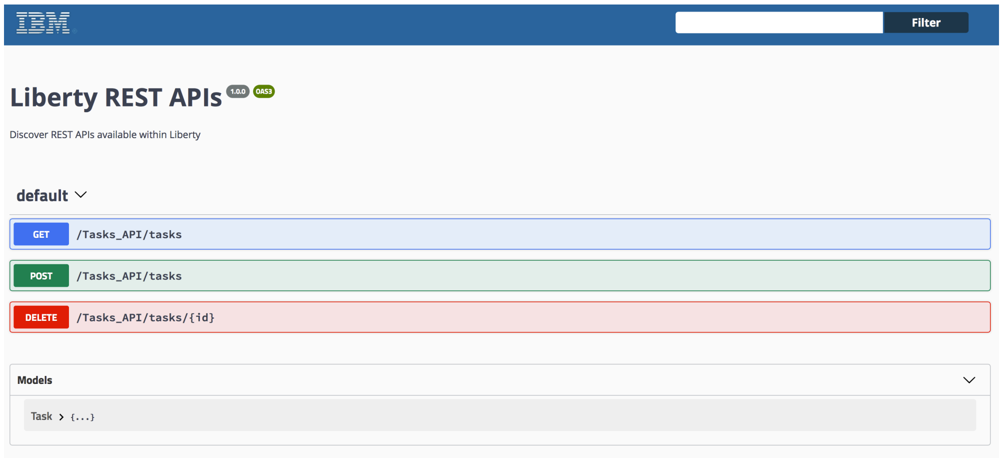
5. Explore the REST APIs

a. On the **Console** view, click the link to view the REST APIs in the API Explorer



```
[AUDIT   ] CWWKT0016I: Web application available (default_host): http://localhost:9080/api/explorer/
[AUDIT   ] CWWKT0016I: Web application available (default_host): http://localhost:9080/api/docs/
[AUDIT   ] CWWKT0016I: Web application available (default_host): http://localhost:9080/Tasks_API/
```

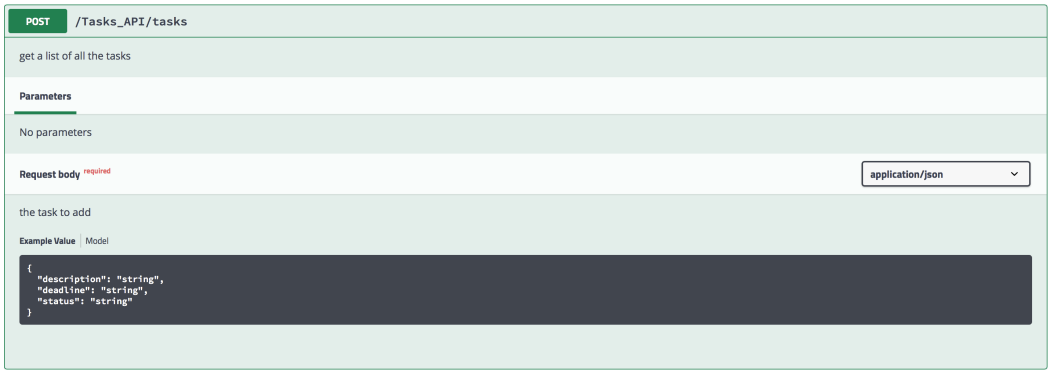
b. You will see that the GET and DELETE methods documented using JAX-RS annotations, and the POST method documented using the programming models are smoothly merged and rendered on the OpenAPI UI



c. Take some time to play around with the API Explorer and how the documentation in the app reflects on what you see in the UI

6. Manipulate the programming models to document a response for the POST method.

   a. You may have noticed that the API for the POST method doesn't specify the type of response it returns.



POST /Tasks_API/tasks

get a list of all the tasks

Parameters

No parameters

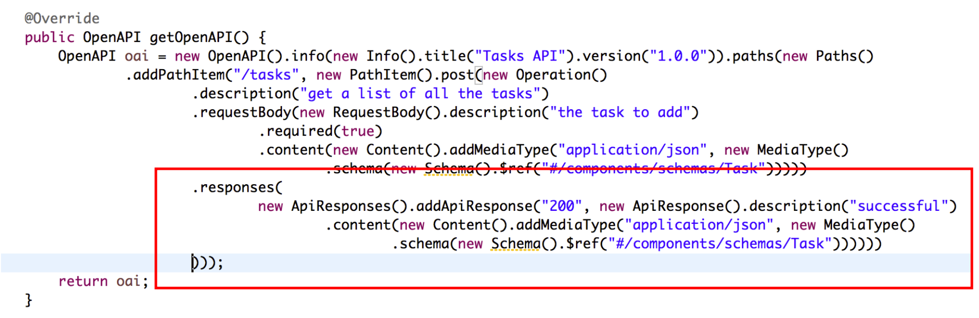
Request body required — application/json

the task to add

Example Value | Model

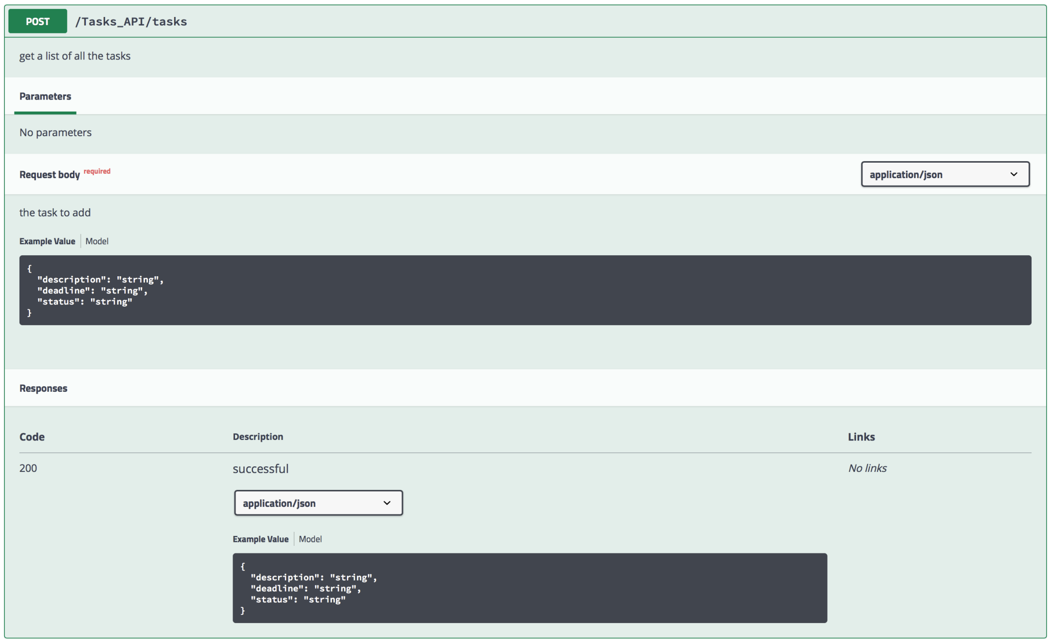
```
{
  "description": "string",
  "deadline": "string",
  "status": "string"
}
```

   b. Insert a ApiResponse model to the OpenAPI model to document what will be returned from the POST request.
   Add the responses as shown below



```
@Override
public OpenAPI getOpenAPI() {
    OpenAPI oai = new OpenAPI().info(new Info().title("Tasks API").version("1.0.0")).paths(new Paths()
            .addPathItem("/tasks", new PathItem().post(new Operation()
                    .description("get a list of all the tasks")
                    .requestBody(new RequestBody().description("the task to add")
                            .required(true)
                            .content(new Content().addMediaType("application/json", new MediaType()
                                    .schema(new Schema().$ref("#/components/schemas/Task")))))
                    .responses(
                            new ApiResponses().addApiResponse("200", new ApiResponse().description("successful")
                                    .content(new Content().addMediaType("application/json", new MediaType()
                                            .schema(new Schema().$ref("#/components/schemas/Task"))))))
                    )));
    return oai;
}
```

c. Save the file and refresh the link for the API Explorer and you will see that the responses are now in the API Explorer.



POST /Tasks_API/tasks

get a list of all the tasks

Parameters

No parameters

Request body required — application/json

the task to add

Example Value | Model

```
{
  "description": "string",
  "deadline": "string",
  "status": "string"
}
```

Responses

Code	Description	Links
200	successful	*No links*

application/json

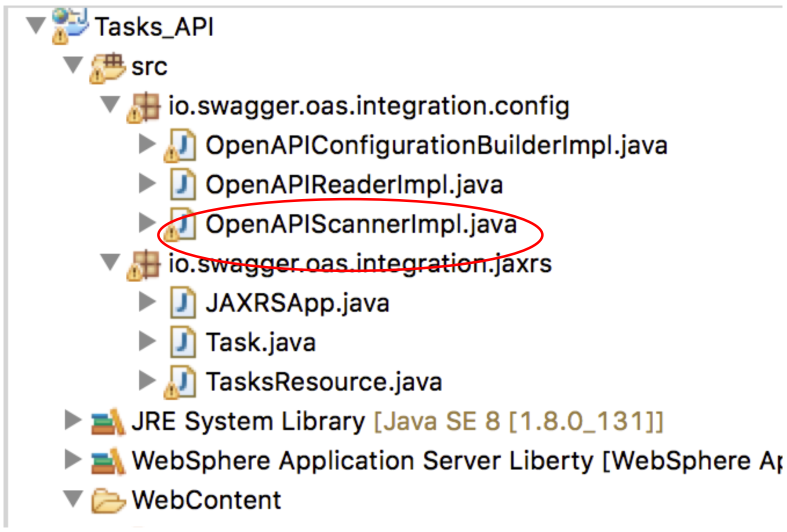
Example Value | Model

```
{
  "description": "string",
  "deadline": "string",
  "status": "string"
}
```

You have just learned how to use the OpenAPI programming models to document your REST APIs for your application and saw how smoothly they merge with other methods of documentations such as using JAX-RS annotations. In the next section, you will see how we can specify your own custom scanner and readers for the REST APIs.

3.3 Add your own custom OpenAPI Scanner and Reader

1. The purpose of adding a custom scanner is to configure which classes and resources are to be processed by the JAX-RS reader. This will override the default scanner which scans for all the classes.
    a. Create a **OpenAPIScanner** that will not scan the **TaskResources** class. Open the **OpenAPIScannerImpl.java** file.



```
Tasks_API
  src
    io.swagger.oas.integration.config
      OpenAPIConfigurationBuilderImpl.java
      OpenAPIReaderImpl.java
      OpenAPIScannerImpl.java
    io.swagger.oas.integration.jaxrs
      JAXRSApp.java
      Task.java
      TasksResource.java
  JRE System Library [Java SE 8 [1.8.0_131]]
  WebSphere Application Server Liberty [WebSphere Ap
  WebContent
```

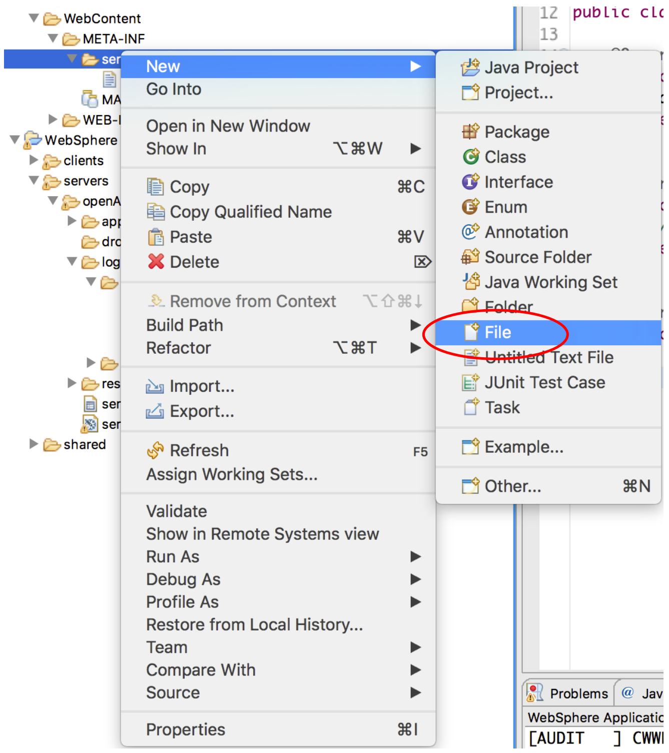
    b. Modify the **getClasses** method so that it returns an empty set of classes. (the default scanner would get all the classes). By doing this, you, effectively, ignore the **TaskResources class**.

    Add to the **getClasses** method as shown below.

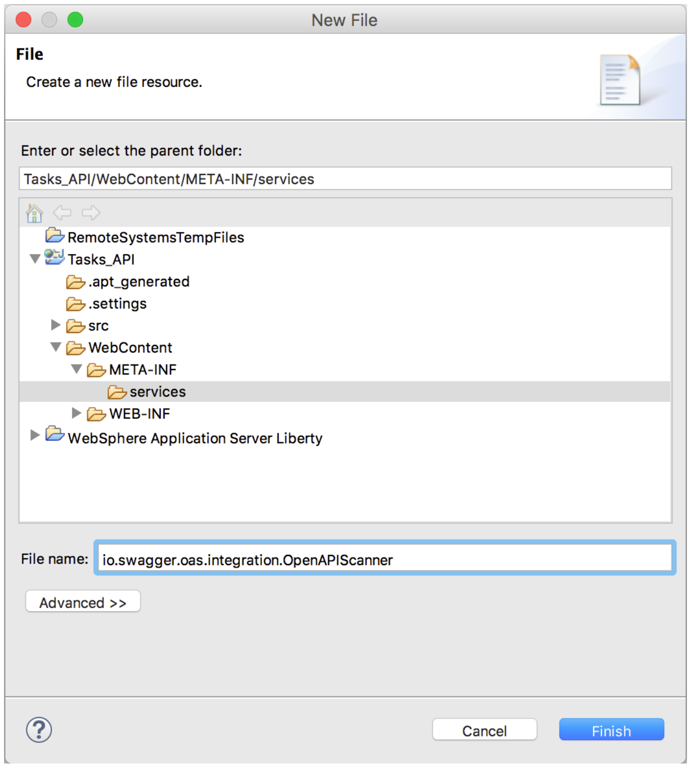
```
@Override
public Set<Class<?>> getClasses() {
    HashSet<Class<?>> set = new HashSet<>();
    return set;
}
```

c. The scanner is now complete. But for the openapi-3.0 feature to load this **OpenAPIScanner**, the application must have a file called **io.swagger.oas.integration.OpenAPIScanner** in the **META-INF/services** folder.

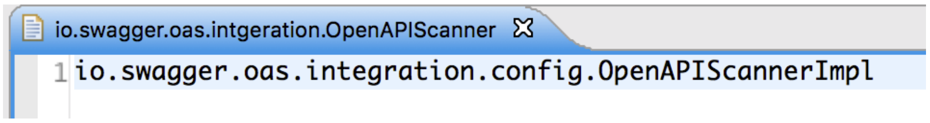
   i. Expand the **Web Content/META-INF** folder, right-click services and click New -> File.



ii. Give the file the name **io.swagger.oas.integration.OpenAPIScanner**.


New File
File
Create a new file resource.
Enter or select the parent folder:
Tasks_API/WebContent/META-INF/services
RemoteSystemsTempFiles
Tasks_API
.apt_generated
.settings
src
WebContent
META-INF
services
WEB-INF
WebSphere Application Server Liberty
File name: io.swagger.oas.integration.OpenAPIScanner
Advanced >>
Cancel
Finish

iii. In that file, specify the fully qualified name of the **OpenAPIScanner** implementation. In this case, the name is **io.swagger.oas.integration.config.OpenAPIScannerImpl**.



```
io.swagger.oas.integration.config.OpenAPIScannerImpl
```

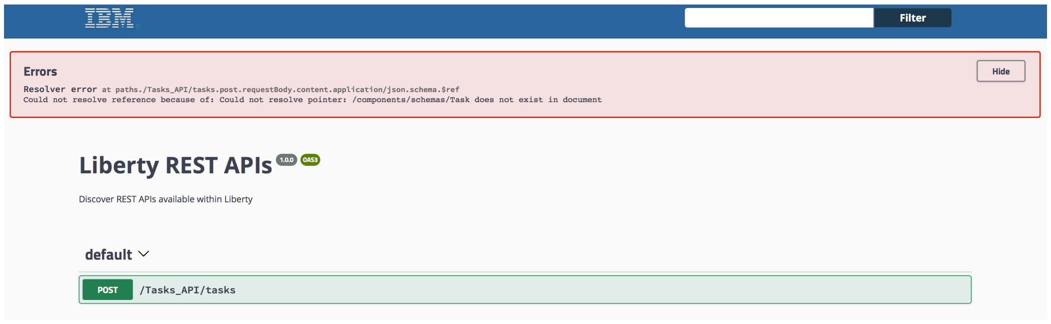
iv. Alternatively, you can also specify the custom **OpenAPIScanner** using the **getScannerClass** in the **OpenAPIConfiguration**. One of these methods of specifying the custom class will suffice.

```
@Override
public String getScannerClass() {
    // TODO Auto-generated method stub
    return "io.swagger.oas.integration.config.OpenAPIScannerImpl";
}
```

v. After saving all the files, the Liberty server will log the fact that you have specified a custom scanner in the **Console** view

```
CWWKO1612I: The server found an implementation of the OpenAPIScanner programming model interface for the OpenAPIWebProvider={contextRoot=/Tasks_API}.
```

d. In the **Console** view, click the link for the API Explorer, you will see that the methods documented in the **TasksResource** class using the annotations are no longer rendered. Moreover, an error is shown in the UI since the **Task** object is not defined. This is because they were documented using the JAX-RS annotations.



Errors

```
Resolver error at paths./Tasks_API/tasks.post.requestBody.content.application/json.schema.$ref
Could not resolve reference because of: Could not resolve pointer: /components/schemas/Task does not exist in document
```

Liberty REST APIs 1.0.0 OAS3

Discover REST APIs available within Liberty

default

POST /Tasks_API/tasks

e. As an exercise, you can fix this by documenting the Task object using the programming models. As seen before, objects are referenced from the components defined. So use the OpenAPI model's components property to define the Task object.

```
   .components(new Components().addSchemas("Task", new Schema()
           .addProperties("description", new Schema().type("string"))
           .addProperties("deadline", new Schema().type("string"))
           .addProperties("status", new Schema().type("string"))));
```

Congratulations! You have successfully completed this lab!