API Folder Docs

**Freerouting.java:** The Freerouting.java file is a dummy class meant for logging purposes, and it does not contain any actual methods or functionality.

**Jakarta API used in the controller files:**

The import statements provided are part of the **Jakarta RESTful Web Services** (Jakarta REST) API, used to build RESTful web services in Jakarta EE applications. Here's a breakdown of each import statement and its significance:

**1. import jakarta.ws.rs.GET;**

* **Purpose**: This annotation indicates that a method in a RESTful resource class will respond to HTTP GET requests.
* **Usage**: It's placed above a method to define that it should be invoked when a GET request is made to a specific URL.

**2. import jakarta.ws.rs.Path;**

* **Purpose**: This annotation is used to define a URI path at the class or method level. It specifies the URL to which the resource or method responds.
* **Usage**: It is placed above a class or method to define the resource's access path.

**3. import jakarta.ws.rs.PathParam;**

* **Purpose**: This annotation allows you to extract values from the URI path. It is typically used to bind a parameter in the URL path to a method parameter.
* **Usage**: This is placed in method parameters to bind path variables to the method argument.

**4. import jakarta.ws.rs.Produces;**

* **Purpose**: This annotation defines the media type(s) that a method can produce as a response, such as JSON, XML, or plain text.
* **Usage**: It's placed above a method to specify the content type of the response.

**5. import jakarta.ws.rs.core.MediaType;**

* **Purpose**: This class provides constants for media types like application/json, text/plain, and application/xml. It is used with the @Produces and @Consumes annotations to specify the type of data a method can return or accept.

**6. import jakarta.ws.rs.core.Response;**

* **Purpose**: The Response class is used to build and return HTTP responses in a RESTful web service. It allows you to control the status code, headers, and body of the response.
* **Usage**: You typically build a Response object using its Response.ok(), Response.status(), or similar methods, followed by build().

# API Folder:

## V1 Folder:

* 1. **JobControllerV1**:

**Purpose**: This controller handles operations related to job management. It exposes endpoints that allow clients to interact with job-related data, specifically retrieving jobs for a session.

**Endpoints**:

* **Base Path**: /v1/jobs
* **Method: listJobs**
  + **HTTP Method**: GET
  + **Path**: /{sessionId}
  + **Parameters**:
    - @PathParam("sessionId"): A session identifier that is passed in the URL.
  + **Response**: The method is designed to return a JSON list of jobs associated with a specific session. The response is currently a placeholder that returns mock job data.

#### **SessionControllerV1.java**

#### **Purpose**: This controller handles session-related operations. It exposes an endpoint that lists all the active sessions.

#### **Endpoints**:

#### **Base Path**: /v1/sessions

#### **Method: listSessions**

#### **HTTP Method**: GET

#### **Path**: /

#### **Response**: This method returns a JSON list of active sessions. Like JobControllerV1, the response is currently hardcoded with placeholder session data.

#### **SystemControllerV1.java**

* **Purpose**: This controller handles system-level operations and configurations.
* **Endpoints**:
  + **Base Path**: /v1/system
  + **Method: getSystemStatus**
    - **HTTP Method**: GET
    - **Path**: /status
    - **Response**: This method is designed to return the system's current status, such as whether the system is operational or experiencing issues. Like the previous controllers, the response is a mock.

## ApiServlet.java:

The ApiServlet.java file you've shared is a Java servlet that handles HTTP GET requests. It extends the HttpServlet class from the **Jakarta Servlet API** and is part of the app.freerouting.api package.

**Imports:**

1. **import jakarta.servlet.http.HttpServlet;**
   * Imports the HttpServlet class, which provides methods to handle HTTP requests (GET, POST, PUT, DELETE, etc.).
2. **import jakarta.servlet.http.HttpServletRequest;**
   * Imports the HttpServletRequest class, representing the client's HTTP request. It provides methods to retrieve data from the request, such as parameters, headers, cookies, and more.
3. **import jakarta.servlet.http.HttpServletResponse;**
   * Imports the HttpServletResponse class, which represents the server's HTTP response. This class allows setting headers, content types, status codes, and sending data back to the client.
4. **import java.io.IOException;**
   * Handles input/output exceptions, especially when dealing with reading from or writing to streams in HTTP communication.

**doGet Method:**

**A screen shot of a computer program

Description automatically generated**

**Flow of the Code:**

1. **Client Sends Request:** A client sends an HTTP GET request to the ApiServlet.
2. **Servlet Handles Request**: The doGet method is triggered when the server receives the GET request.
3. **Response Setup:** The response content type is set to text/plain.
4. **Response Sent:** The text "Hello, World!" is sent back to the client as a plain text response.

## AppContextListener.java:

* The AppContextListener.java is a **context listener** that implements the **ServletContextListener** interface, which allows it to listen for lifecycle events of a **ServletContext**.
* These events occur when the web application is starting up or shutting down. The listener, in this case, is used to perform specific actions when the web server is initialized or destroyed.
* **@WebListener**:
  + The @WebListener annotation registers this class as a **servlet listener**. This tells the web container (such as Jetty) that this class should listen for the application's context lifecycle events (e.g., initialization and destruction).
* **contextInitialized(ServletContextEvent sce)**:
  + This method is triggered when the web application starts.
  + **Purpose**:
    - This method retrieves the running instance of the **Jetty server** from the ServletContext to log the web server's full URL (host and port).
    - It attempts to extract the host and port information of the server to construct the full URL where the API is hosted.

In **mainapplication.java**, the **AppContextListener** is explicitly added as an event listener for the application's context.

**Flow:**

1. **When the Application Starts:**
   * Jetty starts the server and initializes the ServletContext.
   * The AppContextListener listens for the contextInitialized event.
   * It retrieves the Jetty server, extracts the host and port, constructs the full URL, and logs it.
   * This helps in confirming that the API is live and accessible at a specific address**.**
2. **When the Application Stops:**
   * The contextDestroyed method is triggered.
   * It logs that the API web server has stopped, which can be useful for monitoring or during application shutdown.

**Summary of the AppContextListener Class:**

* **Purpose:** Listens to the servlet context lifecycle events and logs information when the web server starts and stops.
* **Initialization:** When the context is initialized, the listener extracts the server's host and port and logs the full API URL.
* **Destruction:** When the context is destroyed, it logs that the server has stopped.
* **Integration with Jetty:** The listener retrieves the Jetty server instance from the ServletContext and uses its connectors to determine the server's configuration (host and port).

**A screenshot of a computer screen

Description automatically generated**

**UML Sequence Diagram between the Web Application, Jetty Server, and Logger**

## OpenAPIConfig.java:

The **OpenAPIConfig.java** file you shared is responsible for setting up the OpenAPI and Swagger configuration for the API documentation in the project. This file is tied to the Jersey RESTful API framework and integrates with Swagger to auto-generate API documentation.

* + - * 1. **Annotations and Class Definition:**

**@ApplicationPath("/api")**:

This annotation defines the base URI path for all the REST endpoints in this application.

In this case, all API endpoints will be under /api. For example, an endpoint in v1 might be accessible at /api/v1.

**extends Application**:

The class extends the **Application**, which is a class provided by Jakarta REST API (JAX-RS). This allows us to define the configuration for the REST application.

* + - * 1. **getSingletons() Method:**

This method is overridden to provide the singletons (objects) that will be initialized and registered as part of the RESTful application.

In this case, it registers **Swagger's OpenApiResource** to generate OpenAPI documentation for the API.

##### Key Elements:

1. **OpenAPI Object**:
   * This object is the core model for Swagger/OpenAPI documentation.
   * It includes metadata such as the API title, description, and version
2. **SwaggerConfiguration**:
   * This class configures how **Swagger** and **OpenAPI** should behave.
   * The configuration object (oasConfig) specifies the OpenAPI instance and a set of resource packages (the API package paths) where the Swagger documentation will look for REST endpoints:
   * **prettyPrint(true)** enables pretty-printed JSON for easy reading in the generated documentation.
   * **resourcePackages** is used to specify which packages should be scanned for API documentation generation. Here, it scans the v1 API package (app.freerouting.api.v1).
3. **OpenApiResource**:
   * This is a **JAX-RS resource** provided by Swagger that exposes the OpenAPI documentation at runtime.
   * Swagger's OpenApiResource is used to serve the OpenAPI JSON file to clients (e.g., Swagger UI or any API documentation tool):

**Return the Singleton**:

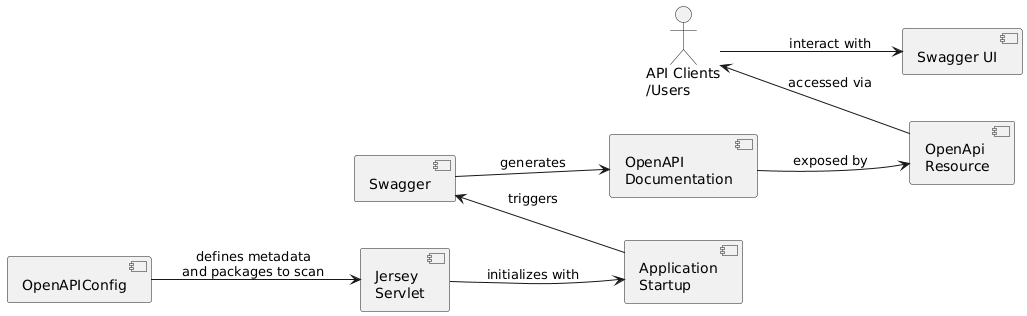
* The method returns a singleton set containing the openApiResource, which is responsible for serving the generated API documentation at runtime.
* In **mainapplication.java**, we are configuring the Jersey servlet to use the **OpenAPIConfig** class for the RESTful application.
* The key parameter being set is "javax.ws.rs.Application", which tells Jersey which class to use to configure the application. In this case, it’s pointing to **OpenAPIConfig**, which sets up the OpenAPI configuration.

**How it Works**:

* When the application starts, the Jersey servlet reads this configuration and initializes the REST application using the class **app.freerouting.api.OpenAPIConfig**.
* This links the **Swagger/OpenAPI** configuration to the running application, allowing the application to generate OpenAPI-compliant documentation for the APIs.

**Flow of the OpenAPI Configuration:**

1. **OpenAPIConfig Setup**:
   * The **OpenAPIConfig** class is responsible for setting up Swagger's OpenAPI integration. It defines the metadata (title, description, version) and specifies which packages to scan for REST endpoints.
2. **Jersey Servlet Initialization**:
   * In the mainapplication.java, the Jersey servlet is configured to use **OpenAPIConfig**. This configuration is done via the setInitParameter() method.
3. **OpenAPI Documentation**:
   * Once the application starts, **Swagger** generates the OpenAPI documentation based on the resources found in app.freerouting.api.v1.
   * **OpenApiResource** is responsible for exposing this documentation, which can be accessed via a defined URL, often /api-docs or /swagger-ui depending on the integration.
4. **Swagger UI/Client**:
   * API clients or users can access the Swagger documentation (typically via a Swagger UI page), view all the API endpoints, and even interact with the APIs (try out the endpoints) if Swagger UI is fully set up.



**Flow of the OpenAPI Configuration**