**900102-000-00-KM-05, REST API and Modularization, NQF Level 4, Credits 2**

**Learner Guide**

**Module Five (5`)**

|  |  |
| --- | --- |
| **Module Code** | 900102-000-00-KM-05 |
| **NQF Level** | 4 |
| **Credits** | 2 |
| **Skills Programme ID Number** | SP- 220329 |
| **Curriculum Title** | Java Programmer |
| **Curriculum Code** | 900102-000-00-00 |

|  |  |
| --- | --- |
| **Name** |  |
| **Contact Address** |  |
| **Telephone (H)** |  |
| **Telephone (W)** |  |
| **Facsimile** |  |
| **Cellular** |  |
| **E-mail** |  |

**Note to the learner**

This Learner Guide provides a comprehensive overview of the module. It is designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks.

**Purpose of the Module**

The main focus of the learning in this knowledge module is to build an understanding of the functionalities of REST API and modularization and when to use them.

The learning will enable learners to demonstrate an understanding of:

* KM-05-KT01: REST API (Application Programming Interface) for Java 50%
* KM-05-KT02: Java Modularization 50%

**Provider Accreditation Requirements for the Knowledge Module**

**Physical Requirements:**

* The provider must have lesson plans and structured learning material or provide learners with access to structured learning material that addresses all the topics in all the knowledge modules as well as the applied knowledge in the application.
* QCTO/ MICT SETA requirements

**Human Resource Requirements:**

* Qualification of lecturer (SME):
* NQF 5 qualified in industry recognised qualifications with 1 year experience in the IT industry o Cybersecurity vendor certification
* Assessors and moderators: accredited by the MICT SETA

**Legal Requirements:**

* Legal (product) licences to use the software for learning and training
* OHS compliance certificate

**Exemptions**

* RPL based

**Venue, Date and Time:**

Consult your facilitator should there be any changes to the venue, date and/or time.Refer to your timetable.

**Assessments**

**Integrated Formative Assessment:** The skills development provider will use the curriculum to guide them on the stipulated internal assessment criteria and weighting. They will also apply the scope of practical skills and applied knowledge as stipulated by the internal assessment criteria. This formative assessment leads to entrance into the integrated external summative assessment.

**Integrated Summative Assessment**: An external integrated summative assessment conducted through the relevant QCTO Assessment Quality Partner is required to issue this qualification. The external integrated summative assessment will focus on the exit level outcomes and associated assessment criteria.

**Skills Programme Purpose**

A Java Programmer will be able to implement solutions to solve real-life problems in an efficient manner, applying a knowledge and understanding of the principles of programming with Java and applicable tools. Tasks that the learner will be able to know, do and understand after achievement of the skills programme include:

* Create well-written and readable Java programs, using a disciplined coding style, including documentation and indentation standards.
* Use Git functionalities for working collaboratively in a team and execute version control.

**Skills Programme Rationale**

Realising the importance and future impact of the Fourth Industrial Revolution (4IR) on the economy of South Africa and its competitiveness, the Minister of Communications gazetted the Presidential Commission on the Fourth Industrial Revolution (PC4IR) on 9 April 2019. By March 2020 this Commission delivered a report with wide ranging recommendations for Human Capital Development that will drive the 4IR forward. It clearly indicated the speed at which companies will have to invest in big data analysis, web-enabled market investment and the use of cloud computing and machine learning.

Software development is central to these initiatives. Software developers are the creative minds behind computer programs. Some develop the applications that allow people to do specific tasks on a computer or another device. Others develop the underlying systems that run the devices or that control networks. The software developer is the important cog in designing advanced computerised technologies. South Africa has a scarcity of software developers and there is a clear need for a qualification focusing specifically on the training and education of software developers.

**Entry Requirements**

Grade 11 with Maths Lit and English.

Access to equipment, internet connectivity and how to work remotely

**EXIT LEVEL OUTCOMES**

**Exit Level Outcomes (ELO) 1**

Describe the basics of Java Programming

Associated Assessment Criteria (AACs)

* The fundamentals of the Java programming language are explained.
* The basic concepts and methods of object-oriented programming and object-oriented design are described.
* The development life-cycle as a means of creating applications is described.

**Exit Level Outcomes (ELO) 2**

Programme effectively using Java frameworks and functionalities

Associated Assessment Criteria (AACs)

* Java syntax is demonstrated, using the Java API.
* Well-written and readable Java programs are created, using a disciplined coding style, including documentation and indentation standards.
* Problems with application development are addressed by troubleshooting.

**Exit Level Outcomes (ELO) 3**

Work collaboratively in a team using GitHub platform

Associated Assessment Criteria (AACs)

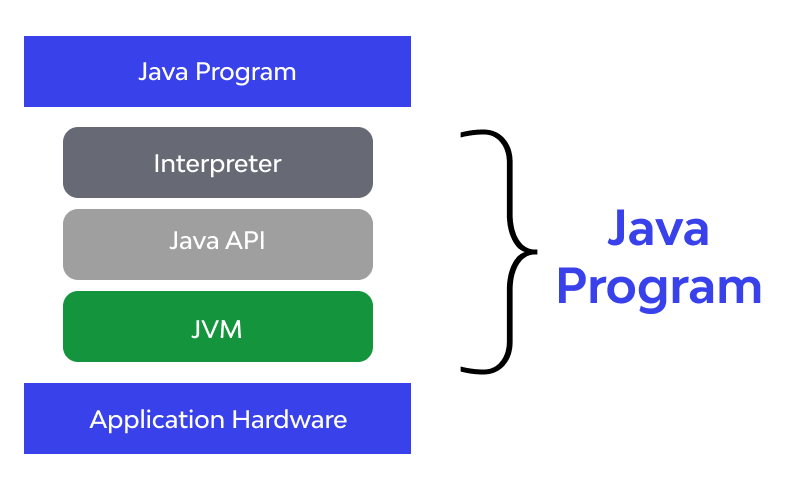
* An ability to work with GitHub is demonstrated.
* Working in a team collaboratively is achieved by using GitHub.
* Version control is exercised using GitHub. functionalities such as repositories, branches, commits and pull requests

**Session 1:** **KM-05-KT01: REST API (Application Programming Interface) for Java**

Topic elements to be covered include:

* KT0101 Definition
* KT0102 Functions
* KT0103 Features
* KT0104 Understanding REST API - 10000 feet overview
* KT0105 Using Docker to Launch MySQL - An Overview

**KT0101 Definition**

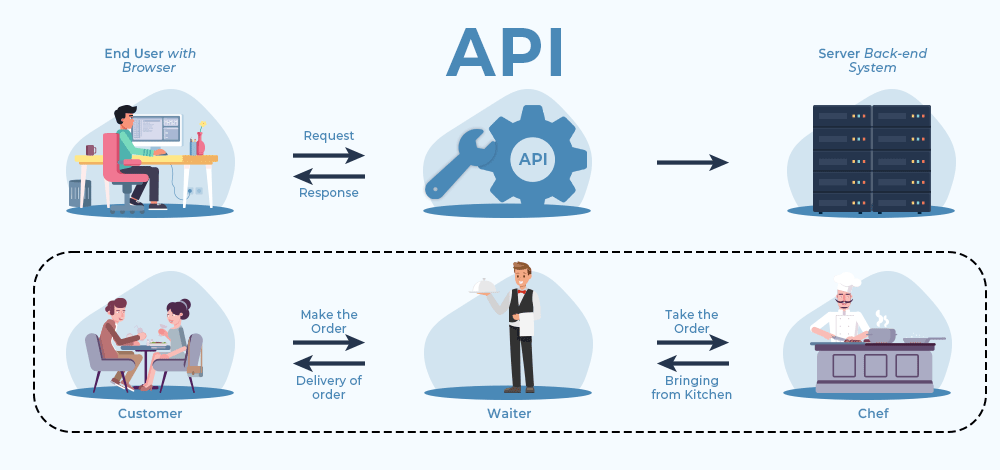


A REST API (Representational State Transfer Application Programming Interface) for Java is a set of rules and conventions for building and interacting with web services using the principles of REST. REST is an architectural style that emphasizes the use of standard HTTP methods and resources to facilitate communication between clients (such as web browsers or mobile apps) and servers. Here's a breakdown of the key components and concepts related to a REST API in the context of Java:

1. **Resources:**
   * Resources are the fundamental entities that a REST API deals with. In the context of Java, resources often represent objects or data models.
   * Each resource is identified by a unique URI (Uniform Resource Identifier) or URL (Uniform Resource Locator), which clients use to access or manipulate that resource.
2. **HTTP Methods (Verbs):**
   * REST APIs use standard HTTP methods (also known as HTTP verbs) to perform actions on resources. The primary HTTP methods used in RESTful APIs are:
     + **GET**: Used to retrieve resource representations.
     + **POST**: Used to create new resources.
     + **PUT**: Used to update existing resources.
     + **DELETE**: Used to remove resources.
   * Java libraries, such as the **javax.ws.rs** package, can be used to map Java methods to specific HTTP methods.
3. **Representation:**
   * Resources can have different representations, such as JSON, XML, HTML, or plain text. These representations determine how data is structured when it's exchanged between the client and server.
   * Java applications often use libraries like Jackson (for JSON) or JAXB (for XML) to serialize and deserialize representations.
4. **Stateless:**
   * RESTful APIs are designed to be stateless, meaning each request from a client to the server must contain all the information needed to understand and process the request. The server does not store client state between requests.
   * Java web frameworks, like Spring Boot or JAX-RS, can help build stateless RESTful services.
5. **Uniform Interface:**
   * REST APIs provide a uniform and consistent interface for interacting with resources. This simplifies client interactions and allows for easy discovery of resources and their capabilities.
   * In Java, annotations like **@GET**, **@POST**, **@PUT**, and **@DELETE** can be used to define resource operations.
6. **Status Codes:**
   * HTTP status codes are used to indicate the outcome of a client's request. Common status codes include 200 (OK), 201 (Created), 204 (No Content), 400 (Bad Request), and 404 (Not Found).
   * Java frameworks often handle status codes automatically based on the response from the server.
7. **Hypermedia (HATEOAS):**
   * Hypermedia as the Engine of Application State (HATEOAS) is an optional feature in REST APIs that allows resources to include links to related resources.
   * While HATEOAS is not commonly used in all REST APIs, some Java frameworks and libraries support it.
8. **Security:**
   * REST APIs often implement security measures like authentication and authorization to protect resources and ensure that only authorized clients can access certain endpoints.
   * Java provides various security libraries and frameworks, such as Spring Security, for implementing security in RESTful applications.

To create a REST API in Java, developers commonly use frameworks like Spring Boot, JAX-RS (part of Java EE), or Jersey. These frameworks provide abstractions and tools to simplify the development of RESTful services and make it easier to handle HTTP requests and responses in a Java application.

**KT0102 Functions**

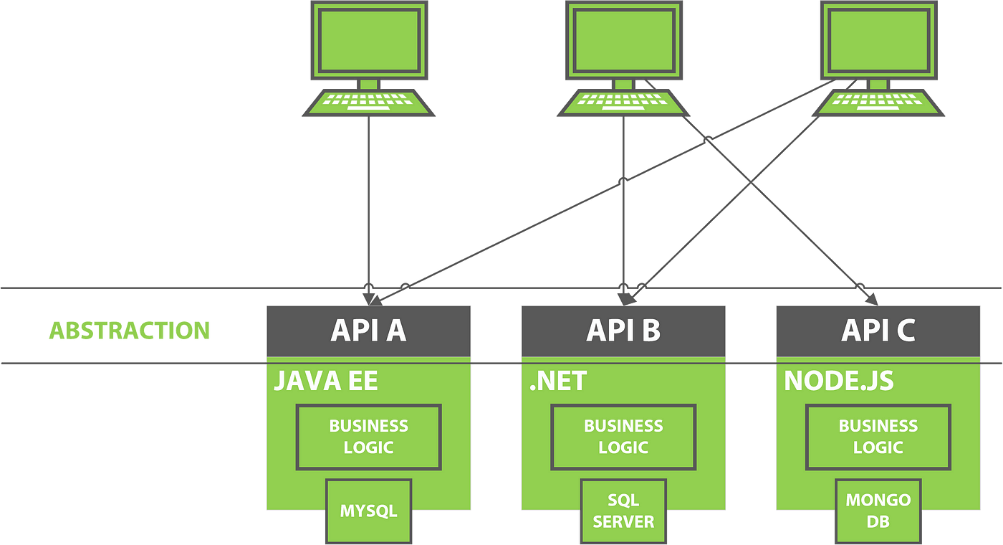


REST APIs (Representational State Transfer Application Programming Interfaces) for Java serve as interfaces for building web services that adhere to the principles of REST. These APIs enable communication between clients and servers using HTTP methods and standard conventions. Here are some of the essential functions and features of a REST API in the context of Java:

1. **Resource Mapping:**
   * A REST API maps Java classes or objects to resources, allowing clients to interact with these resources using HTTP requests.
   * In Java, libraries like JAX-RS (Java API for RESTful Web Services) provide annotations (**@Path**, **@GET**, **@POST**, etc.) for mapping Java methods to specific URI paths and HTTP methods.
2. **HTTP Methods (Verbs):**
   * REST APIs use standard HTTP methods to perform actions on resources.
   * Java REST frameworks allow developers to define methods that correspond to HTTP methods (**@GET**, **@POST**, **@PUT**, **@DELETE**) for resource retrieval, creation, update, and deletion.
3. **Request Handling:**
   * REST APIs in Java frameworks receive incoming HTTP requests and route them to the appropriate resource or endpoint based on the URI and HTTP method.
   * Request parameters, headers, and bodies can be extracted and processed within the API.
4. **Response Generation:**
   * REST APIs generate HTTP responses with appropriate status codes and response bodies.
   * Java libraries, such as Jackson for JSON or JAXB for XML, help serialize Java objects into response representations.
5. **Resource Identification:**
   * Resources in a REST API are identified by unique URIs. Clients use these URIs to access or manipulate resources.
   * Java frameworks allow developers to define resource classes and annotate them with path segments.
6. **Content Negotiation:**
   * REST APIs support content negotiation, allowing clients to request data representations (e.g., JSON, XML) that best suit their needs.
   * Java APIs typically handle content negotiation by examining request headers (e.g., **Accept**) and generating responses accordingly.
7. **Error Handling:**
   * REST APIs provide error handling mechanisms, returning appropriate HTTP status codes (e.g., 400 for Bad Request, 404 for Not Found) in response to invalid requests.
   * Java frameworks offer mechanisms for customizing error responses.
8. **Pagination and Filtering:**
   * APIs often support pagination and filtering to retrieve subsets of data. Clients can specify query parameters to control the number of results and filter criteria.
   * Java APIs allow developers to parse and process query parameters to fetch and return filtered data.
9. **Authentication and Authorization:**
   * REST APIs can implement authentication mechanisms (e.g., OAuth, JWT) to secure access to resources.
   * Authorization checks can be applied to ensure that clients have the necessary permissions to perform specific actions.
   * Java security libraries, such as Spring Security, can be integrated to handle authentication and authorization.
10. **Versioning:**
    * APIs may support versioning to manage changes and updates to resource representations over time.
    * Java APIs can include version information in the URI or request headers.
11. **Caching and ETag Support:**
    * Caching mechanisms like ETags (Entity Tags) can be used to optimize resource retrieval by reducing the need for redundant requests.
    * Java APIs can support caching directives in response headers.
12. **Cross-Origin Resource Sharing (CORS):**
    * APIs can implement CORS headers to control which domains are allowed to access the API from the client-side (e.g., web browsers).
    * Java frameworks provide mechanisms for configuring CORS policies.
13. **Testing and Documentation:**
    * Testing tools and documentation generation are essential for developers and consumers of the API.
    * Java APIs can benefit from testing frameworks like JUnit and tools like Swagger or OpenAPI for documentation.

Java provides several libraries and frameworks for building RESTful APIs, including Spring Boot, JAX-RS (Jersey), and RESTEasy. These frameworks simplify the development of RESTful services and provide built-in support for many of the functions mentioned above.

**KT0103 Features**

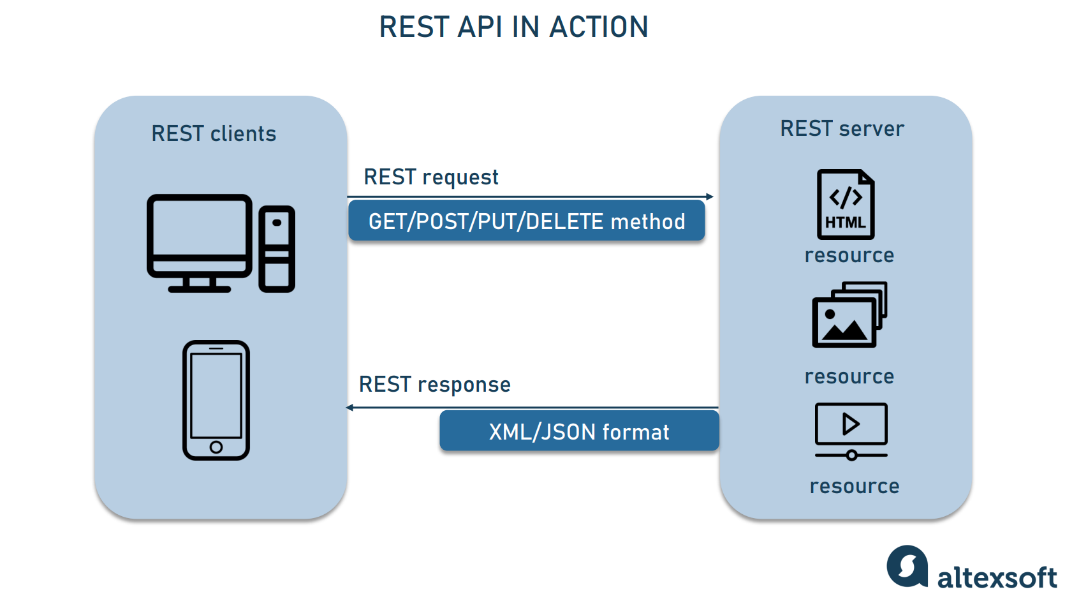


REST APIs (Representational State Transfer Application Programming Interfaces) for Java are designed to offer a set of features and characteristics that adhere to the principles of REST. These features help ensure that APIs are efficient, scalable, and easy to use. Here are some of the key features of REST APIs in the context of Java:

1. **Stateless Communication:**
   * RESTful APIs are inherently stateless, meaning that each request from a client to the server must contain all the information needed to understand and process the request.
   * Stateless communication simplifies server implementation and allows for horizontal scalability.
2. **Resource-Centric:**
   * REST APIs model the application's data and functionality as resources, which are identified by unique URIs (Uniform Resource Identifiers).
   * In Java, resources are often represented as Java objects or classes that can be mapped to URI paths using annotations (e.g., **@Path** in JAX-RS).
3. **Uniform Interface:**
   * REST APIs provide a uniform and consistent interface for interacting with resources. This simplifies client interactions and allows for easy discovery of resources and their capabilities.
   * HTTP methods (GET, POST, PUT, DELETE) serve as a standard set of operations to interact with resources.
4. **HTTP Methods (Verbs):**
   * REST APIs use standard HTTP methods (also known as HTTP verbs) to perform actions on resources.
   * Java REST frameworks, such as JAX-RS or Spring Boot, allow developers to map Java methods to specific HTTP methods for resource manipulation.
5. **Representation:**
   * Resources can have multiple representations, such as JSON, XML, HTML, or plain text. Clients specify their preferred representation using the **Accept** header.
   * Java libraries, like Jackson or JAXB, assist in serializing and deserializing representations.
6. **Stateless Server:**
   * RESTful servers do not maintain client state between requests. Each request is independent, which reduces server complexity and makes scaling easier.
   * Session management is typically avoided in RESTful APIs.
7. **Status Codes:**
   * HTTP status codes are used to indicate the outcome of a client's request. Common status codes include 200 (OK), 201 (Created), 204 (No Content), 400 (Bad Request), and 404 (Not Found).
   * Java REST frameworks handle status codes automatically based on the response.
8. **Content Negotiation:**
   * Clients can negotiate the content type they desire (e.g., JSON or XML) using the **Accept** header in their requests.
   * Java APIs can respond with the appropriate content type based on client preferences.
9. **Caching:**
   * APIs may include caching mechanisms to optimize resource retrieval by reducing redundant requests.
   * Java APIs can set cache-control headers in responses.
10. **Cross-Origin Resource Sharing (CORS):**
    * CORS headers can be included in responses to control which domains are allowed to access the API from the client-side (e.g., web browsers).
    * Java frameworks provide mechanisms to configure CORS policies.
11. **Error Handling:**
    * REST APIs provide error handling mechanisms, returning appropriate HTTP status codes (e.g., 400 for Bad Request, 404 for Not Found) in response to invalid requests.
    * Java frameworks allow customization of error responses.
12. **Security:**
    * APIs often implement security measures like authentication (e.g., OAuth, JWT) and authorization to protect resources.
    * Java security libraries, such as Spring Security, can be integrated for authentication and authorization.
13. **Testing and Documentation:**
    * Comprehensive testing and documentation are essential for developers and consumers of the API.
    * Java APIs can benefit from testing frameworks (e.g., JUnit) and tools (e.g., Swagger, OpenAPI) for documentation generation.

These features collectively define the characteristics and behavior of REST APIs in Java. When building or consuming RESTful services, adhering to these principles and features helps ensure that APIs are well-designed, easy to maintain, and interoperable with various clients and servers.

**KT0104 Understanding REST API - 10000 feet overview**



Understanding REST API at a high level, often referred to as a "10,000 feet overview," involves grasping the fundamental concepts and principles behind RESTful architecture without delving into detailed technical specifics. Here's a high-level overview of REST API:

**1. REST Defined:**

* REST stands for Representational State Transfer.
* It is an architectural style for designing networked applications, emphasizing simplicity, scalability, and statelessness.

**2. Resources:**

* In REST, everything is considered a resource, which can be any piece of data or functionality.
* Resources are identified by unique URIs (Uniform Resource Identifiers), which are analogous to URLs (Uniform Resource Locators).

**3. HTTP Methods (Verbs):**

* REST uses standard HTTP methods (also known as HTTP verbs) to perform actions on resources.
* Common HTTP methods include GET (retrieve), POST (create), PUT (update), and DELETE (delete).

**4. Stateless:**

* RESTful systems are stateless, meaning each request from a client to the server must contain all the information needed to understand and process the request.
* Servers do not store client state between requests.

**5. Representations:**

* Resources can have multiple representations (e.g., JSON, XML) to accommodate various client preferences.
* Clients specify their preferred representation using the **Accept** header in the request.

**6. Uniform Interface:**

* REST APIs provide a uniform and consistent interface for interacting with resources.
* A standard set of operations, such as GET for retrieval and POST for creation, simplifies client interactions.

**7. Status Codes:**

* HTTP status codes indicate the outcome of a client's request. For example, 200 means OK, 201 means Created, and 404 means Not Found.
* Status codes help clients understand the result of their actions.

**8. Stateless Servers:**

* RESTful servers do not maintain client state between requests. Each request is independent.
* Stateless design simplifies server architecture and supports scalability.

**9. Hypermedia (Optional):**

* Hypermedia as the Engine of Application State (HATEOAS) is an optional feature in REST APIs.
* It allows resources to include links to related resources, making API discovery and navigation easier.

**10. Practical Use Cases:** - REST APIs are widely used in web services, mobile app backends, and IoT (Internet of Things) applications. - They provide a flexible and efficient means of communication between clients and servers.

**11. Security and Authentication:** - REST APIs often implement security measures, including authentication and authorization, to protect resources. - Authentication mechanisms like OAuth and JWT are commonly used.

**12. Versioning (Optional):** - APIs may include versioning to manage changes and updates to resource representations over time. - Version information can be included in the URI or request headers.

This high-level overview captures the core principles and characteristics of REST API architecture. In practice, building and consuming RESTful services involves more detailed considerations, such as choosing appropriate HTTP status codes, designing resource URIs, handling request and response headers, and implementing secure communication. However, this overview provides a solid foundation for understanding the essence of RESTful design.

**KT0105 Using Docker to Launch MySQL - An Overview**

Using Docker to launch MySQL is a popular approach for quickly setting up and managing MySQL database instances in a containerized environment. Docker simplifies the process of provisioning, configuring, and running MySQL databases while providing isolation and portability. Here's an overview of how to use Docker to launch MySQL:

**1. Install Docker:**

* Ensure you have Docker installed on your system. You can download and install Docker from the official website (https://www.docker.com/get-started).

**2. Pull MySQL Image:**

* Docker images are pre-packaged containers that you can use to create instances of MySQL. To pull the official MySQL image from Docker Hub, open a terminal and run the following command:

docker pull mysql

**3. Run MySQL Container:**

* Once you have the MySQL image downloaded, you can create and run a MySQL container. Use the **docker run** command to do this. Here's a basic example:

docker run --name mysql-container -e MYSQL\_ROOT\_PASSWORD=my-secret-pw -d mysql

* + **--name mysql-container**: Specifies a name for the container (you can choose any name).
  + **-e MYSQL\_ROOT\_PASSWORD=my-secret-pw**: Sets the root password for MySQL. Replace **my-secret-pw** with your desired password.
  + **-d**: Runs the container in detached mode (in the background).

**4. Connect to MySQL:**

* Once the MySQL container is running, you can connect to it using a MySQL client or a database management tool. Use the following command to connect to the MySQL server inside the container:

docker exec -it mysql-container mysql -uroot -p

* + **-it**: Allocates a pseudo-TTY and allows interaction with the container.
  + **-uroot**: Specifies the MySQL user (root in this case).
  + **-p**: Prompts for the MySQL root password (enter the password you set in step 3).

**5. Interact with MySQL:**

* You can now interact with the MySQL server as you would with a standalone MySQL installation. Create databases, tables, insert data, and execute SQL queries using the MySQL client within the container.

**6. Stop and Remove the Container:**

* When you're done using the MySQL container, you can stop and remove it with the following commands:

docker stop mysql-container docker rm mysql-container

**7. Persist Data (Optional):**

* By default, data inside a Docker container is ephemeral, meaning it is lost when the container is removed. To persist MySQL data between container runs, you can use Docker volumes. You'll need to create a volume and mount it to the MySQL container's data directory.

This overview provides the basic steps to get started with Docker and MySQL. In real-world scenarios, you may need to customize the MySQL container configuration further, such as specifying custom database initialization scripts or exposing specific ports for remote access. Docker provides a flexible and convenient way to manage MySQL instances in various development and deployment environments.

**Internal Assessment Criteria and Weight**

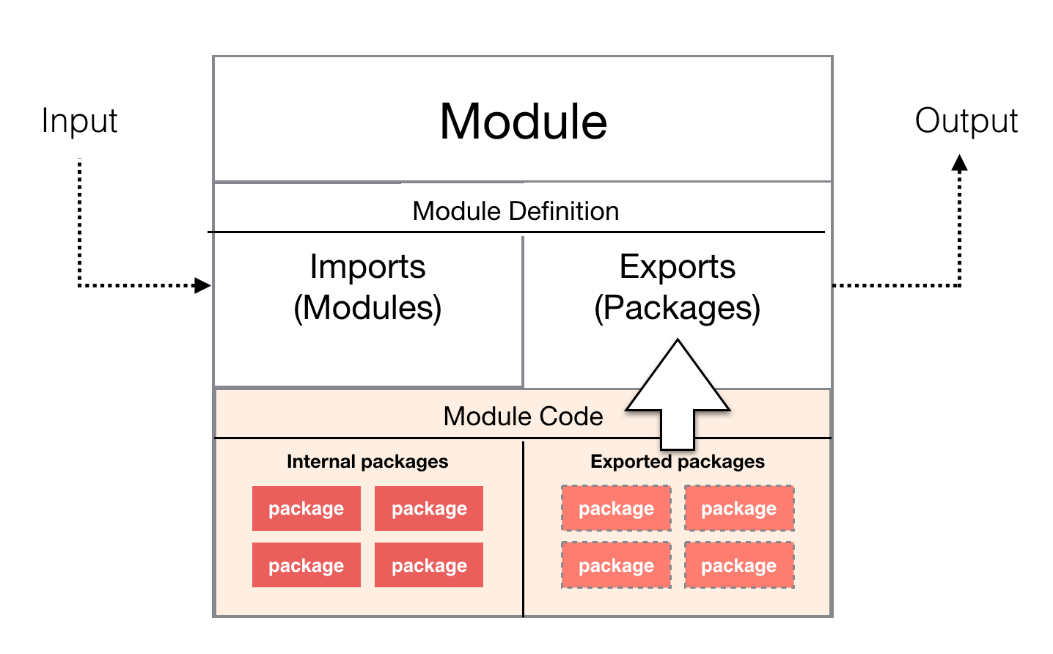
1. IAC0101 Definitions, functions and features of REST API in Java are stated.

**Session 2:** **KM-05-KT02 :Java Modularization**

Topic elements to be covered include:

* KT0201 Definition
* KT0202 Functions
* KT0203 Features
* KT0204 Java New Features
  + - Local Variable Type Inference
    - Switch Expression
    - Text Blocks
    - Records

**KT0201 Definition**



Java modularization, also known as the Java Platform Module System (JPMS), is a feature introduced in Java 9 to enhance the organization, maintainability, and encapsulation of Java applications and libraries. It allows you to partition a Java application or library into discrete modules, each with its own well-defined scope and dependencies. Here's a definition of Java modularization:

**Java Modularization (JPMS):** Java modularization, or the Java Platform Module System (JPMS), is a feature in Java that enables developers to break down large monolithic codebases into smaller, self-contained modules. Each module encapsulates a specific set of related classes and resources, and it defines clear boundaries for access to its internal code. Modularization offers several benefits, including improved code organization, stronger encapsulation, better dependency management, and enhanced maintainability.

Key components and concepts of Java modularization include:

* **Module:** A module is a self-contained unit of code that packages related classes, interfaces, and resources. It defines its dependencies on other modules and specifies which parts of its code are accessible to other modules (exports) and which are encapsulated (internal).
* **Module Descriptor:** Each module contains a **module-info.java** file, known as the module descriptor, which defines the module's name, dependencies, and access control rules. It also declares the packages it exports and the services it provides or consumes.
* **Module Path:** Java applications use the module path to locate and load modules. It is similar to the classpath but supports module resolution. Modules can be explicitly added to the module path, allowing the Java runtime to resolve and load dependencies.
* **Strong Encapsulation:** Modules enforce strong encapsulation, meaning that only explicitly exported packages are accessible from outside the module. This helps prevent unintended access to internal implementation details.
* **Module System Commands:** The Java Platform Module System introduces new command-line tools, such as **jlink** for creating custom runtime images and **jdeps** for analyzing dependencies between modules.
* **Improved Dependency Management:** Modularization simplifies dependency management by explicitly specifying module dependencies. This reduces classpath-related issues and makes it easier to manage complex projects.
* **Migration:** Existing Java codebases can be gradually modularized by introducing module descriptors and organizing code into modules. This allows developers to modernize and modularize legacy applications over time.

Java modularization is particularly useful for large-scale applications and libraries, where it enhances code organization, promotes code reuse, and simplifies maintenance. It also helps address classpath-related issues and supports the creation of custom runtime images, making Java applications more efficient and compact.

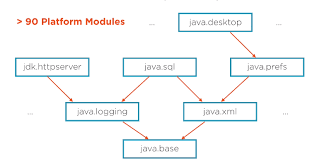
**KT0202 Functions**

Java modularization, introduced with the Java Platform Module System (JPMS) in Java 9, offers several functions and benefits for developers and the Java ecosystem as a whole. These functions of Java modularization are designed to improve code organization, maintainability, and reliability. Here are the key functions of Java modularization:

1. **Code Organization:**
   * Java modularization allows you to organize your code into self-contained modules. Each module encapsulates related classes, interfaces, and resources.
   * This function helps developers create a clear and structured project layout, making it easier to understand, navigate, and maintain code.
2. **Encapsulation:**
   * Modules enforce strong encapsulation, which means that internal details of a module are hidden from other modules by default.
   * Encapsulation prevents unintended access to implementation details, reducing the risk of code coupling and promoting modularity.
3. **Dependency Management:**
   * Java modules explicitly declare their dependencies on other modules in their module descriptors (**module-info.java** files).
   * This function simplifies dependency management and ensures that the correct versions of dependencies are used, reducing classpath-related issues.
4. **Access Control:**
   * Modules specify which packages are exported for use by other modules and which packages are encapsulated and not accessible externally.
   * Access control allows fine-grained control over the visibility of classes and packages, enhancing security and reducing the risk of unintended dependencies.
5. **Strong Module Boundaries:**
   * Modules provide clear and well-defined boundaries between different parts of an application, making it easier to reason about and test code.
   * Strong module boundaries promote isolation and reduce the risk of "classloader hell" and classpath-related issues.
6. **Custom Runtime Images:**
   * Java modularization enables the creation of custom runtime images using tools like **jlink**. These images contain only the modules required for an application, resulting in smaller and more efficient deployments.
7. **Improved Maintenance:**
   * Modular codebases are typically easier to maintain and evolve over time. Changes to one module are less likely to impact other modules.
   * This function enhances the long-term maintainability of Java applications and libraries.
8. **Migration Support:**
   * Java modularization allows for a gradual migration process. Developers can modularize existing codebases over time by introducing module descriptors and partitioning code into modules.
   * This function supports the modernization of legacy Java applications.
9. **Improved Dependency Analysis:**
   * Tools like **jdeps** help analyze dependencies between modules and identify potential issues in the codebase.
   * Improved dependency analysis assists in keeping the codebase clean and modular.
10. **Module Versioning (Java 11+):**
    * In Java 11 and later versions, module versioning support is introduced, allowing modules to specify version information. This helps manage compatibility between modules and ensures that the correct versions are used.

Java modularization is a valuable feature for projects of various sizes, as it provides a structured and scalable approach to organizing and managing code. It fosters good software engineering practices, enhances code quality, and contributes to the long-term maintainability of Java applications and libraries.

**KT0203 Features**



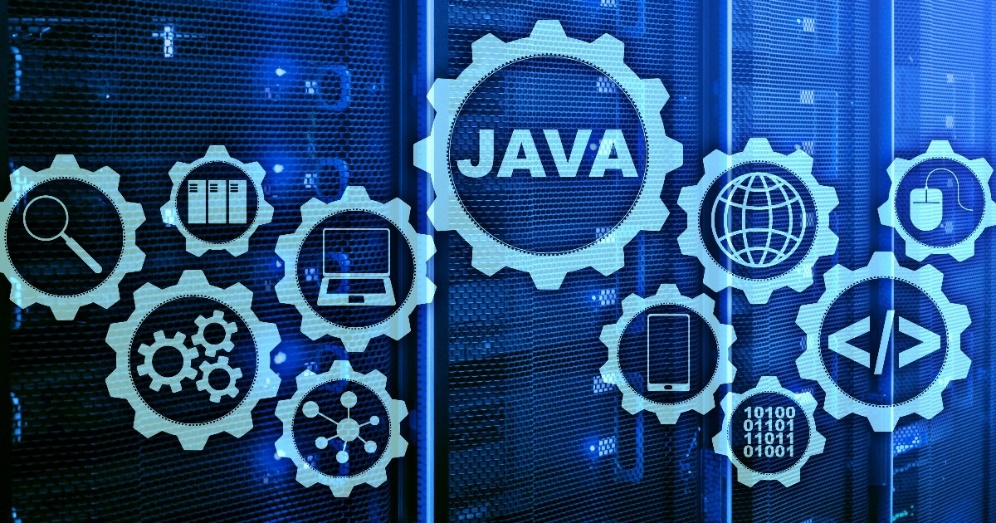
Java modularization, introduced with the Java Platform Module System (JPMS), brings several important features to the Java ecosystem. These features enhance code organization, maintainability, and reliability while allowing developers to build more scalable and efficient applications. Here are the key features of Java modularization:

1. **Modular Code Organization:**
   * Java modules allow you to organize your codebase into self-contained and well-defined modules.
   * Each module encapsulates a set of related classes, interfaces, and resources, promoting better code organization.
2. **Strong Encapsulation:**
   * Modules enforce strong encapsulation, meaning that the internal details of a module are hidden from other modules by default.
   * Encapsulation helps prevent unintended access to internal implementation details and reduces the risk of coupling.
3. **Explicit Dependencies:**
   * Java modules explicitly declare their dependencies on other modules in their module descriptors (**module-info.java** files).
   * This feature simplifies dependency management and ensures that the correct versions of dependencies are used.
4. **Access Control:**
   * Modules specify which packages are exported for use by other modules and which packages are encapsulated and not accessible externally.
   * Access control allows fine-grained control over class visibility and prevents unauthorized access to internal classes and APIs.
5. **Strong Module Boundaries:**
   * Modules provide clear and well-defined boundaries between different parts of an application.
   * Strong module boundaries reduce the risk of classpath-related issues and promote isolation between modules.
6. **Custom Runtime Images (Java 9+):**
   * Java modularization enables the creation of custom runtime images using tools like **jlink**.
   * Custom runtime images contain only the modules required for an application, resulting in smaller and more efficient deployments.
7. **Improved Dependency Management:**
   * Modules eliminate classpath-related issues, as they explicitly specify their dependencies.
   * Dependency management becomes more reliable and straightforward with modularization.
8. **Enhanced Security:**
   * Strong encapsulation and access control mechanisms enhance security by preventing unauthorized access to internal code.
   * Encapsulation reduces the risk of malicious code interfering with internal implementations.
9. **Clear Module Declarations:**
   * Module descriptors (**module-info.java**) provide clear and declarative information about a module's dependencies, exports, and services.
   * This feature improves code documentation and helps developers understand module interactions.
10. **Migration Support:**
    * Java modularization allows for gradual migration. Existing codebases can be modularized incrementally by introducing module descriptors and partitioning code into modules.
    * This feature supports the modernization of legacy Java applications.
11. **Improved Dependency Analysis:**
    * Tools like **jdeps** help analyze dependencies between modules, identify potential issues, and generate reports.
    * Improved dependency analysis assists in keeping the codebase clean and modular.
12. **Module Versioning (Java 11+):**
    * In Java 11 and later versions, module versioning support is introduced, allowing modules to specify version information.
    * Module versioning helps manage compatibility between modules and ensures that the correct versions are used.

These features collectively make Java modularization a valuable addition to the Java platform, enabling developers to build more organized, maintainable, and scalable applications and libraries while enhancing security and dependency management.

**KT0204 Java New Features**

* **Local Variable Type Inference**
* **Switch Expression**
* **Text Blocks**
* **Records**

****

Java has introduced several new features and enhancements in recent versions to improve code readability, maintainability, and developer productivity. Here are some of the notable features introduced in Java:

1. **Local Variable Type Inference (var)**:
   * Starting with Java 10, you can use the **var** keyword to declare local variables with inferred types. The type is determined by the compiler based on the assigned value.
   * This feature reduces verbosity and enhances code readability for local variables, especially when working with complex types.

var name = "John"; // Inferred as String var numbers = List.of(1, 2, 3); // Inferred as List<Integer>

1. **Switch Expression (Java 12+)**:
   * In Java 12 and later versions, the traditional **switch** statement is enhanced to support switch expressions.
   * Switch expressions allow you to assign a value to a variable directly based on the result of a switch case.

int dayOfWeek = switch (day) { case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" -> 1; case "Saturday" -> 2; case "Sunday" -> 3; default -> throw new IllegalArgumentException("Invalid day"); };

1. **Text Blocks (Java 13+)**:
   * Java 13 introduced text blocks, a new way to write multi-line strings with improved readability and reduced escape sequences.
   * Text blocks are enclosed in triple double-quotes **"""** and maintain the formatting within the block.

String html = """ <html> <body> <p>Hello, world!</p> </body> </html> """;

1. **Records (Java 16+)**:
   * Records are a concise way to declare simple data classes, reducing boilerplate code.
   * Records automatically generate constructors, **equals()**, **hashCode()**, and **toString()** methods based on the declared fields.

record Point(int x, int y) {} Point p1 = new Point(1, 2);

These features contribute to making Java code more concise, expressive, and readable. They enhance developer productivity by reducing the need for boilerplate code and allowing developers to focus on the logic of their applications. It's important to note that the availability of these features depends on the Java version you are using, so you should check the Java version compatibility for each feature.

**Internal Assessment Criteria and Weight**

1. IAC0201 Definitions, functions and features of Java modularization are stated.

**References**

***REST API References:***

1. ***"RESTful Web Services" by Leonard Richardson and Sam Ruby:***
   * *This book provides a comprehensive introduction to RESTful web services and includes practical examples and best practices.*
   * [*Link to Book*](http://shop.oreilly.com/product/9780596529260.do)
2. ***RESTful API Design Guidelines (Microsoft):***
   * *Microsoft offers a set of guidelines and best practices for designing RESTful APIs, covering topics like resource naming, versioning, and security.*
   * [*Microsoft REST API Guidelines*](https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md)
3. ***"REST API Tutorial" by RESTfulAPI.net:***
   * *This online tutorial offers a step-by-step guide to creating RESTful APIs, covering key concepts and practical implementation.*
   * [*RESTfulAPI.net Tutorial*](https://restfulapi.net/)
4. ***Swagger and OpenAPI Specification:***
   * *Swagger is a popular tool for API documentation and design. The OpenAPI Specification (formerly Swagger Specification) is a format for describing RESTful APIs.*
   * [*Swagger*](https://swagger.io/)
   * [*OpenAPI Specification*](https://swagger.io/specification/)
5. ***REST API Best Practices (RESTfulAPI.net):***
   * *This resource provides a collection of best practices for designing RESTful APIs, with a focus on simplicity and consistency.*
   * [*REST API Best Practices*](https://restfulapi.net/rest-api-design-best-practices/)

***Java Modularization References:***

1. ***"Java 9 Modularity" by Sander Mak and Paul Bakker:***
   * *This book offers a comprehensive guide to Java modularization (JPMS) in Java 9 and later versions. It covers module creation, dependencies, and migration from non-modular to modular code.*
   * [*Link to Book*](https://www.oreilly.com/library/view/java-9-modularity/9781491954172/)
2. ***"Java 9 Modularity Revealed" by Alex Buckley:***
   * *This book provides insights into the design and implementation of the Java Platform Module System (JPMS) in Java 9.*
   * [*Link to Book*](https://www.apress.com/gp/book/9781484227412)
3. ***Java Platform, Standard Edition 9 Documentation (Oracle):***
   * *Oracle's official documentation provides in-depth information about Java modularization, module descriptors, and module-related tools.*
   * [*Java 9 Documentation - Modules*](https://docs.oracle.com/javase/9/docs/technotes/guides/module/index.html)
4. ***"Java Modules" (Baeldung):***
   * *Baeldung offers a series of tutorials and articles on Java modularization, covering various aspects of module development and migration.*
   * [*Baeldung Java Modules Tutorials*](https://www.baeldung.com/java-modules)
5. ***"Migrating to Modules" (Oracle):***
   * *This Oracle article provides guidance on migrating existing codebases to Java modules and handling common migration challenges.*
   * [*Migrating to Modules*](https://docs.oracle.com/en/java/javase/16/migrate/)