**LANGUAGES**

**DATABASE  
  
DESIGN**

**REST API**

|  |  |
| --- | --- |
| **REST API** | |
| REST API | Interface for two computer systems |
|  | REST APIs communicate via HTTP requests to perform standard database functions like creating, reading, updating, and deleting records |
|  | Formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text |
| API | Allows for an application to access a resource in another application or service |
| REST | REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways. |

|  |  |
| --- | --- |
| **GET** | Retrieve Data |
|  | Cacheable, stay in history, bookmarked, length restrictions |
| **POST** | Submit something often causing a change of state or side effect on server |
|  | Not cached, not in history can't be bookmarked |
| **PUT** | Update something |
| **DELETE** | Delete something |
| **CONNECT** | Create tunnel to server |
| **OPTIONS** | Describe communication options for resource |
| **TRACE** | Message loop back test along path to target resource |
| **PATCH** | Partial modifications to resource |
| **OPTIONS** | describes the communication options for the target resource |
| **HEAD** | Like GET but no request body |

|  |  |
| --- | --- |
| **Uniform interface** | All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn’t be too large but should contain every piece of information that the client might need. |
| **Client-server decoupling** | In REST API design, client and server applications must be completely independent of each other. The only information that the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP. |
| **Statelessness** | REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren’t allowed to store any data related to a client request. |
| **Cacheability** | When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side. |
| **Layered system architecture** | In REST APIs, the calls and responses go through different layers. As a rule of thumb, don’t assume that the client, and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary. |

**DESIGN PATTERNS**

**Solid**

|  |  |
| --- | --- |
| Single Responsibility Principle | A class should have one and only one reason to change, meaning that a class should have only one job. |
| [Open/Closed Principle](https://stackify.com/solid-design-open-closed-principle/) | Objects or entities should be open for extension but closed for modification. |
| [Liskov Substitution Principle](https://stackify.com/solid-design-liskov-substitution-principle/) | Every subclass or derived class should be substitutable for their base or parent class. |
| [Interface Segregation Principle](https://stackify.com/interface-segregation-principle/) | A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use. |
| [Dependency Inversion](https://stackify.com/dependency-inversion-principle/) | Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions. |

**Design**

**A picture containing text, screenshot, font, number

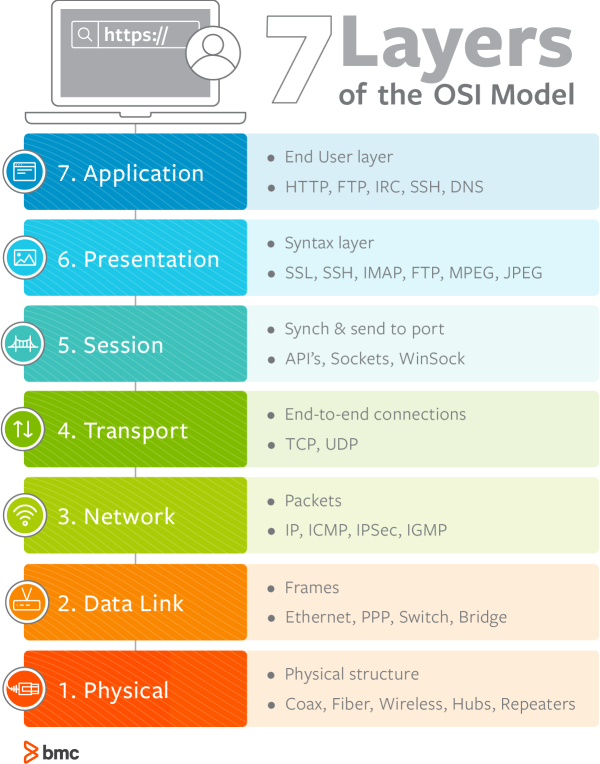
Description automatically generated**

|  |  |
| --- | --- |
| **Creational Design Patterns** | |
| **Pattern Name** | **Description** |
| Singleton | The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created. |
| Factory | The factory pattern takes out the responsibility of instantiating a object from the class to a Factory class. |
| Abstract Factory | Allows us to create a Factory for factory classes. |
| Builder | Creating an object step by step and a method to finally get the object instance. |
| Prototype | Creating a new object instance from another similar instance and then modify according to our requirements. |

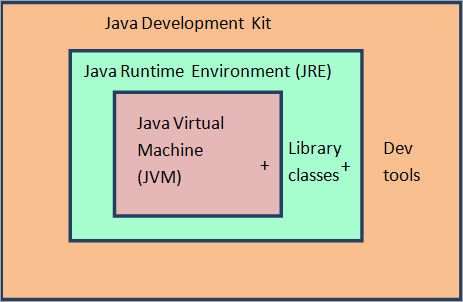
|  |  |
| --- | --- |
| **Structural Design Patterns** | |
| Adapter | Provides an interface between two unrelated entities so that they can work together. |
| Composite | Used when we have to implement a part-whole hierarchy. For example, a diagram made of other pieces such as circle, square, triangle, etc. |
| Proxy | Provide a surrogate or placeholder for another object to control access to it. |
| Flyweight | Caching and reusing object instances, used with immutable objects. For example, string pool. |
| Facade | Creating a wrapper interfaces on top of existing interfaces to help client applications. |
| Bridge | The bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client program. |
| Decorator | The decorator design pattern is used to modify the functionality of an object at runtime. |

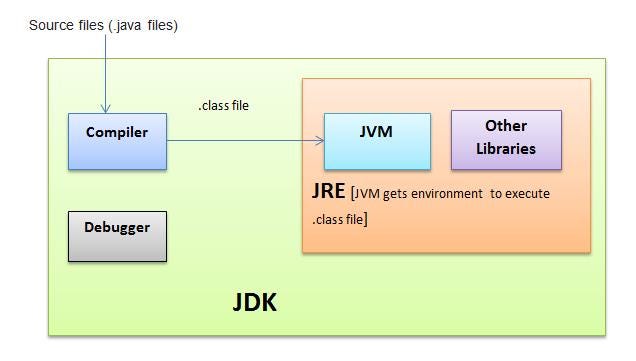
|  |  |
| --- | --- |
| **Behavioral Design Patterns** | |
| Template Method | used to create a template method stub and defer some of the steps of implementation to the subclasses. |
| Mediator | used to provide a centralized communication medium between different objects in a system. |
| Chain of Responsibility | used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. |
| Observer | useful when you are interested in the state of an object and want to get notified whenever there is any change. |
| Strategy | Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime. |
| Command | Command Pattern is used to implement lose coupling in a request-response model. |
| **State** | State design pattern is used when an Object change it’s behavior based on it’s internal state. |
| Visitor | Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. |
| Interpreter | defines a grammatical representation for a language and provides an interpreter to deal with this grammar. |
| **Iterator** | used to provide a standard way to traverse through a group of Objects. |
| **Memento** | The memento design pattern is used when we want to save the state of an object so that we can restore later on |

**OSI**



**SORT**

**Java**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **How it Works** | |  |  | public | All classes |
| Hardware | OS | JVM |  | private | Declared class |
| my code | bytecode | jvm |  | default | Same Package |
| .java -> | java compiler -> | .class |  | protected | [Same Package and Subclass](https://www.w3schools.com/java/java_inheritance.asp) |

|  |  |  |
| --- | --- | --- |
| **Java Language** | | |
| Portable | Multithreaded | High performance |
| Object-oriented | Architecture-neutral | Distributed |
| Robust | Interpreted | Dynamic |

|  |  |
| --- | --- |
| **Static (Belongs to Type)** | |
| Shared Memory, Access without instatiation, cant access non-static | |
| Blocks | Runs once do computation for variables |
| Variables | Single copy all classes |
| Methods | Access without creating class |
| Classes | For an inner nested class |

|  |  |  |
| --- | --- | --- |
| **OOP** | | |
| Inheritance | I do all the time | extends |
| Interface |  | implements |
| Abstract Class |  | extends |
| Method Overloading | Same name but different inputs |  |
| Polymorphism | Child can override parent method |  |
|  | Interface Animal - Dog Horse Sound |  |

|  |  |
| --- | --- |
| **Version** | **Features** |
| **Java 8** | Lambdas, Collections and Streams |
| **Java 9** | Collections new helper methods (Easily make set, list Map) |
|  | Interfaces got private methods (sensitive code) |
|  | HTTP Client |
| **Java 10** | var detect type |
| **Java 11** | HTTP Client- finalized |
|  | Strings got methods is blank and strip |
| **Java 13** | Switch statements can return a value and lambda syntax |
| **Java 14** | standarized switch |
| **Java 15** | Strings in multiline |
| **Java 16** | Patterm matching for instance of |
| **Java 17 (LTS)** |  |
| **Java 18** | UTF-8 by default |
|  | Simple Web Server (jwebserver) |
| **Java 19** | virtual threads |
| **Java 20** |  |

**Concepts**

|  |
| --- |
| **AGILE** (A mindset) |
| Individuals and interactions over processes and tools |
| Working software over comprehensive documentation |
| Customer collaboration over contract negotiation |
| Responding to change over following a plan |

|  |
| --- |
| **AGILE 12 Principles** |
| Highest priority is to satisfy the customer through early and continuous delivery of valuable software. |
| Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage. |
| Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. |
| Business people and developers must work together daily throughout the project. |
| Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. |
| The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. |
| Working software is the primary measure of progress. |
| Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. |
| Continuous attention to technical excellence and good design enhances agility. |
| Simplicity–the art of maximizing the amount of work not done–is essential. |
| The best architectures, requirements, and designs emerge from self-organizing teams. |

**Scrum**

Scrum is a management framework that teams use to self-organize and work towards a common goal. It describes a set of meetings, tools, and roles for efficient project delivery. Much like a sports team practicing for a big match, Scrum practices allow teams to self-manage, learn from experience, and adapt to change.

|  |  |  |
| --- | --- | --- |
| **Scrum methodology** | | |
| **Principles** | **Artifacts** | **Roles** |
| Transparency | Product Backlog | Product Owner |
| Reflection | Sprint Backlog | Scrum Leader |
| Adaption | Increment- Step towards goal | Scrum Team |
|  |  |  |
| **Events** | **Importance** |  |
| Sprint Planning | Happier more productive |  |
| Sprint | Better ROI |  |
| Daily Scrum or stand-up | Metrics |  |
| Sprint Review |  |  |
| Sprint Retrospective |  |  |

**Frameworks**

**Spring**

Build enterprise level applications

* Focus on business issue and let Spring handle rest

Overview

* Build simple classes (like a business service) with annotations (@service) that define what they are
  + This allows Spring to manage lifecyle of a class
* Allows for dependency injection which allows objects to define own dependencies that the Spring container injects into them
* Can build loosely coupled components that are ideal for [microservices](https://www.ibm.com/topics/microservices) and distributed network applications

**Spring Boot**

Java Spring Boot is an open-source tool that makes it easier to use Java-based frameworks to create microservices and web apps.

* Convention over configuration
* Standalone

Overview

* Large body of pre-written code
* Helps remove configuration, build and deploy steps

Setup

* Maven- Put dependencies in POM.xml
* Gradle- Build tool
* Uses pom.xml
* Add <parent> that has springboot as its parent (Define configuration here and child inherits)

Annotations

* @SpringBootApplication: Main
* @service
* @controller
* @RequestMapping (/api/hi)
* @autowired- Needs dependency injection

**Spring Initializer**

* Quick start
  + Metadata and dependencies you need
  + Java version and Language
* Core: Security
* Web: Web, Web Service Web Socket
* Cloud Messaging:
* Basicaly like NPM

**Spring Boot CLI**

* Run groovy scripts less boilerplate code

**Spring MVC**

* Map Routes to a controller (Java Class)
* Convert to a response

**A picture containing text, screenshot, font, diagram

Description automatically generated**

|  |  |  |
| --- | --- | --- |
|  | **Spring** | **Spring Boot** |
| What is it? | An open-source web application framework based on Java. | An extension or module built on the Spring framework. |
| What does it do? | Provides a flexible, completely configurable environment using tools and libraries of prebuilt code to create customized, loosely coupled web apps. | Provides the ability to create, standalone Spring applications that can just run immediately without the need for annotations, XML configuration, or writing lots of additional code. |
| When should I use it? | Use Spring when you want: | Use Spring Boot when you want: |
| Flexibility | Ease of use |
| An unopinionated approach.\* | An opinionated approach.\* |
| To remove dependencies from your custom code. | To get quality apps running quickly and reduce development time. |
| To implement a very unique configuration. | To avoid writing boilerplate code or configuring XML. |
| To develop enterprise applications. | To develop REST APIs. |
|  |  |
| What's its key feature? | Dependency injection | Autoconfiguration |
| Does it have embedded servers? | No. In Spring, you'll need to set up the servers explicitly. | Yes, Spring Boot comes with built-in HTTP servers like Tomcat and Jetty. |
| How is it configured? | The Spring framework provides flexibility, but its configuration has to be built manually. | Spring Boot configures Spring and other third-party frameworks automatically by the default "convention over configuration" principle. |
| Do I need to know how to work with XML? | In Spring, knowledge of XML configuration is required. | Spring Boot does not require XML configuration. |
| Are there CLI tools for dev/testing apps? | The Spring framework alone doesn't provide CLI tools for developing or testing apps. | As a Spring module, Spring Boot has a CLI tool for developing and testing Spring-based apps. |

**AWS  
DynamoDB**

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability

|  |  |  |
| --- | --- | --- |
|  | **Explanation** | **Example** |
| **Table** | Like SQL | Users |
| **Item** | Group of Attributes | A current user |
| **Attribute** | Fundament Data Element | userID |

|  |  |
| --- | --- |
| **Primary Key** | Uniquely identify an Item uses internal hash function |
| **Partition Key** | Simple Primary Key (One attribute) |
| **Composite Primary Key** | Partition key and sort key |

|  |  |
| --- | --- |
| **PutItem** | Create an item |
| **GetItem** | Read an item |
| **UpdateItem** | Update an item |
| **DeleteItem** | Delete an item |
| **BatchGetItem** | Read up to 100 items from one or more tables |
| **BatchWriteItem** | Create or delete up to 25 items in one or more tables |

|  |  |
| --- | --- |
| **Scalar Types** | A scalar type can represent exactly one value |
|  | number, string, binary, Boolean, and null |
| **Document Types** | A document type can represent a complex structure with nested attributes like JSON |
|  | The document types are list and map |
| **Set Types** | A set can be multiple scalar values. |
|  | The set types are string set, number set, and binary set. |

**Lambda**

**A picture containing text, screenshot, diagram, font

Description automatically generated**

**A computer screen shot of a diagram

Description automatically generated with low confidenceA screenshot of a computer

Description automatically generated with medium confidence**

**Database**

**SQL vs NoSQL**

|  |  |  |
| --- | --- | --- |
|  | **SQL** | **NoSQL** |
| **Overview** | Relational Data | Non-relational structure |
|  |  | Utilizes a dynamic schema |
| **Scaling** | Vertical or Sharding | NoSQL databases scale better horizontally |
|  |  | Add additonal servers for load |
| **Properties** | ACID | **Consistency:** Every request receives either the most recent result or an error |
|  |  | **Availability:** Every request has a non-error result. |
|  |  | **Partition tolerance:** Any delays or losses between nodes do not interrupt the system operation. |
| **Structure** | Rows and Table | **Column-oriented:** where data is stored in cells grouped in a virtually unlimited number of columns rather than rows. |
|  |  | **Key-value stores:** which use an associative array (also known as a dictionary or map) as their data model |
|  |  | **Document stores:** which use documents to hold and encode data in standard formats, including XML, YAML, JSON (JavaScript Object Notation) and BSON. A benefit is that documents within a single database can have different data types. |
|  |  | **Graph databases**: which represent data on a graph that shows how different sets of data relate to each other |

**Normal Forms**

|  |  |  |
| --- | --- | --- |
| **1NF** | Each Column Unique | Each Cell a Single Value |
| **2NF** | Each Column directly related to primary key |  |
| **3NF** | All non-key attributes independent of each other | Each column directly related to primary key and not other columns |

**ACID**

|  |  |
| --- | --- |
| **Atomicity** | Each statement in a transaction CRUD is treated as a single unit. Either the entire statement is executed, or none of it is executed. |
| **Consistency** | ensures that transactions only make changes to tables in predefined, predictable ways |
| **Isolation** | when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions don't interfere with or affect one another. Each request can occur as though they were occurring one by one, even though they're actually occurring simultaneously |
| **Durability** | ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure. |