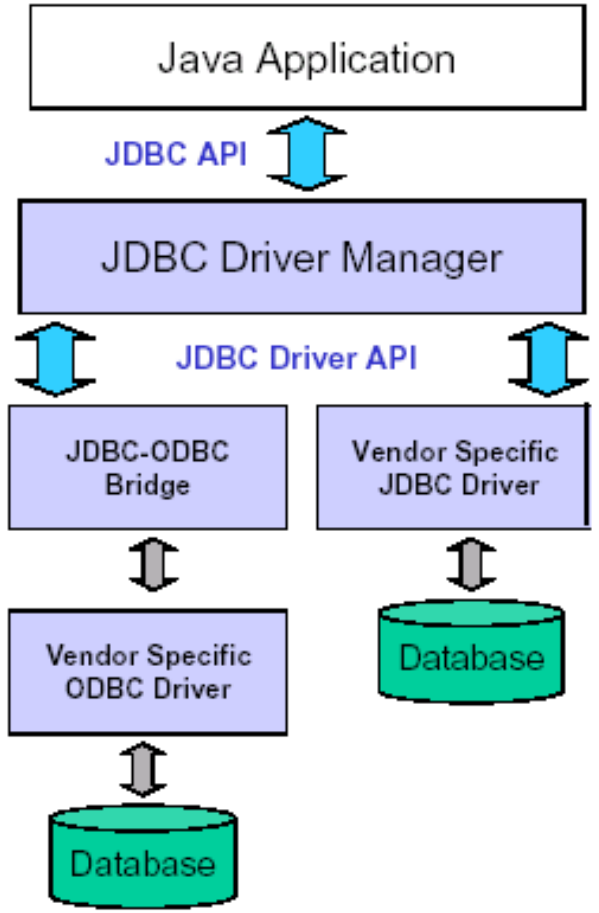
Software Applications with Databases

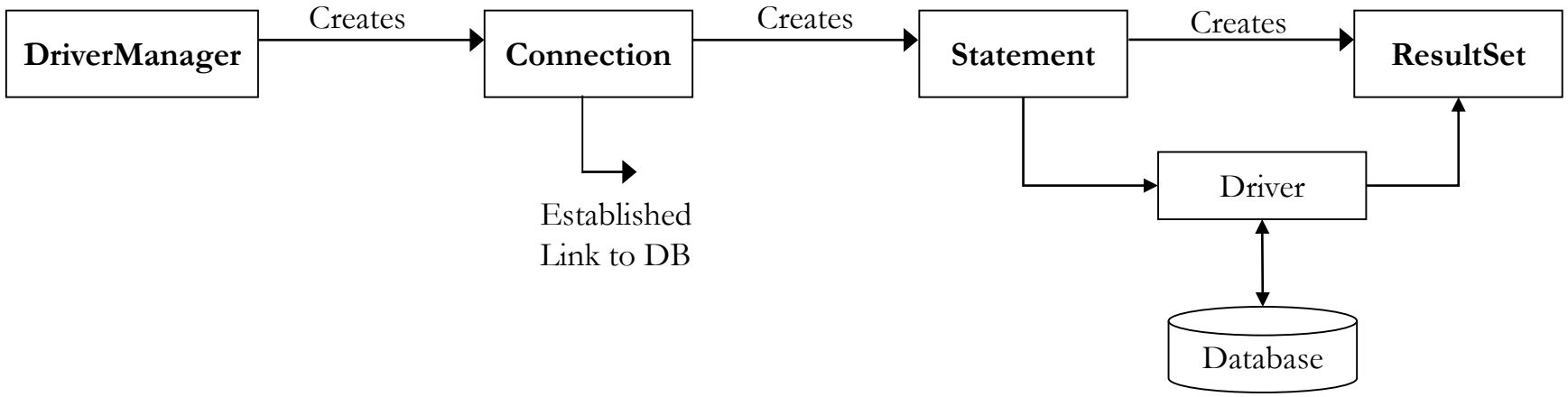
JDBC: Java Database Connectivity

- Standard library for Java programs to access a DBMS and execute SQL statements
  - JDBC API, a purely Java-based API
  - JDBC Driver Manager, which communicates with DBMS-specific drivers that perform the actual operations on databases
  - In 2 main packages **java.sql** and **javax.sql**



JDBC: Components

- **Driver Manager**: Load database drivers and manage connections
- **Driver**: Translate API calls into operations on database
- **Connection**: Create connection between Java application and data source
- **Statement**: Perform SQL statements (select, update, delete...)
- **ResultSet**: Contain data returned by executing a statement



Basic steps

- Import the necessary classes
- Load the JDBC driver
- Identify the data source via connection URL
- Establish a **Connection** object
- Create a **Statement** object
- Execute SQL query string using **Statement** object
- Retrieve data from the returned **ResultSet** object
- Close ResultSet, Statement, and Connection objects in order

JDBC Driver

- Loaded dynamically using Class.forName(String drivername)
- Example:
  - Load MySQL JDBC Driver: `Class.forName("com.mysql.cj.jdbc.Driver");`
  - Load Oracle JDBC Driver: `Class.forName("oracle.jdbc.driver.OracleDriver");`
  - Load SQLite JDBC Driver: `Class.forName("org.sqlite.JDBC");`

Connection URL

- Identify a database
- Syntax: <protocol>:<subprotocol>:<database>
  - protocol: Protocol used to access database (jdbc here)
  - subprotocol: Database driver
  - database: Name of database
- Example
  - "jdbc:sqlite:C:/Database/store.db"
  - "jdbc:odbc:Movies"

Connection URL

- Syntax in MySQL

```
jdbc:mysql://[host][,failoverhost...]
    [:port]/[database]
    [?propertyName1][=propertyValue1]
    [&propertyName2][=propertyValue2]...
```

- Example

```
"jdbc:mysql://localhost:3306/store?user=admin&password=123456"
```

Create Connection object

- Required to communicate with a database via JDBC
- Use methods in DriverManagaer
  - public static Connection getConnection(String url)
  - public static Connection getConnection(String url, Properties info)
  - public static Connection getConnection(String url, String user, String password)

```
try {
     Class.forName("org.sqlite.JDBC"); // Load the driver
     Connection conn = DriverManager.getConnection("jdbc:sqlite:store.db"); }
catch (ClassNotFoundException cnfe) {
     System.err.println("Driver is not loaded!");   }
catch (SQLException sqle) {
     System.err.println("Error connecting database! Database might not exist!");    }
```

Code Example for Oracle

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    String sourceURL = "jdbc:oracle:thin:@delilah.bus.albany.edu:1521:projects";
    String user = "goel";
    String password = "password";
    Connection conn = DriverManager.getConnection(sourceURL,user, password);
    System.out.println("Connected Connection"); }

catch (ClassNotFoundException cnfe) {
    System.err.println("Driver is not loaded!");   }
catch (SQLException sqle) {
    System.err.println("Error connecting database! Database might not exist!");
}
```

Statement object

- Used to create, retrieve, update & delete data (CRUD) from a table
  - Create: `Statement statement = connection.createStatement();`

    `PreparedStatement pstatement = connection.prepareStatement(sqlString);`
- Three types of statements
  - Statement: general
  - PreparedStatement: parameterized
  - CallableStatement: stored procedures

Executing Queries

- Methods of Statement interface
  - executeQuery(sql)
  - executeUpdate(sql)
- executeQuery: Retrieve data from a database to a ResultSet
  - Select commands
- executeUpdate: Insert, update, delete data; create, update, delete tables
  - Insert, update, delete, create table, drop table... commands
- setQueryTimeout: specify a maximum delay to wait for results

Data Definition Language

- Data definition language queries use executeUpdate
  - Returns the number of rows updated
- Example 1: Create a new table

```
Statement statement = connection.createStatement();
String sqlString =
"Create Table Catalog"
+ "(Title Varchar(256) Primary Key Not Null,"+
+ "LeadActor Varchar(256) Not Null, LeadActress Varchar(256) Not Null,"
+ "Type Varchar(20) Not Null, ReleaseDate Date Not NULL )";
Statement.executeUpdate(sqlString);
```

Data Definition Language

- Example 2: Update table

```
Statement statement = connection.createStatement();
String sqlString =
 "Insert into Catalog"
+ "(Title, LeadActor, LeadActress, Type, ReleaseDate)”
+ "Values(‘Gone With The Wind’, ‘Clark Gable’, ‘Vivien Liegh’,"
+ "'Romantic', '02/18/2003'"
Statement.executeUpdate(sqlString);
```

  - executeUpdate returns a 1 since one row is added

Data Manipulation Language

- Data definition language queries use executeQuery
- Syntax: ResultSet executeQuery(String sqlString) throws SQLException
  - Return a ResultSet object which contains the results of the Query
- Example 1: Query a table

```
Statement statement = connection.createStatement();
String sqlString = "Select * From Catalog";
ResultSet rs = statement.executeQuery(sqlString);
```

ResultSet object

- ResultSet contains results returned by a query to the database
- We can scroll through each row and read data from all columns
- ResultSet provides various access methods that take a column index or column name and returns the data
- When the executeQuery method returns a ResultSet the cursor is placed before the first row of the data
  - Cursor refers to the set of rows returned by a query and is positioned on the row that is being accessed
  - Call next() to move the cursor to the first row of data
  - If the next row has data, next() returns true else it returns false and the cursor moves beyond the end of the data
- First column has index 1, not 0

ResultSet object

- Numerous functions to read data in cells
  - getShort(), getInt(), getLong()
  - getFloat(), getDouble()
  - getClob(), getBlob(),
  - getDate(), getTime(), getArray(), getString()

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String sourceURL = "jdbc:odbc:music";
Connection conn   = DriverManager.getConnection(sourceURL);

Statement statement = conn.createStatement();
String queryString = "SELECT recordingtitle, listprice FROM recordings";
ResultSet results = statement.executeQuery(queryString);

while (results.next()){
     System.out.println(results.getString("recordingtitle") +
                  "\t" + results.getFloat("listprice"));       }

results.close();
Statement.close
conn.close();
```

JDBC – Data Types

JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]
CHAR VARCHAR LONGVARCHAR	String

JDBC Type	Java Type
NUMERIC DECIMAL	BigDecimal
DATE	java.sql.Date
TIME TIMESTAMP	java.sql.Timestamp
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

PreparedStatement object

- PreparedStatement create a reusable statement precompiled by the database
- Processing time of a SQL query consists of
  - Parsing the SQL string
  - Checking the Syntax
  - Checking the Semantics
  - Parsing time is often longer than time required to run the query
- PreparedStatement is used to pass a SQL string to the database where it can be pre-processed for execution

PreparedStatement object

- Example

```
// Creating a prepared Statement
String sqlString = “UPDATE authors SET lastname = ? Authid = ?”;
PreparedStatement ps = connection.prepareStatement(sqlString);
ps.setString(1, “Allamaraju”);     // Sets first placeholder to Allamaraju
ps.setInt(2, 212);                 // Sets second placeholder to 212
ps.executeUpdate();                // Executes the update
```

Software design

- Architectural design
  - Divide the system into subsystems, components
  - Determine how they will interact
- Detailed design
  - Determine internal details of each subsystem, component
    - Class design
    - Algorithm design
- User interface design
- Database design

Software architecture

- Software architecture is the overall organization of a software system
  - How system is divided into sub-systems and components
    - Modules, packages, classes, functions...
  - How components and sub-systems communicate and interact
    - Connectors, interfaces, protocols
- Overall design goals
  - Efficiency: as small and fast as possible
  - Reusability: components can be reused in new versions or other systems
  - Maintainability: system can be changed, repaired, upgraded easily

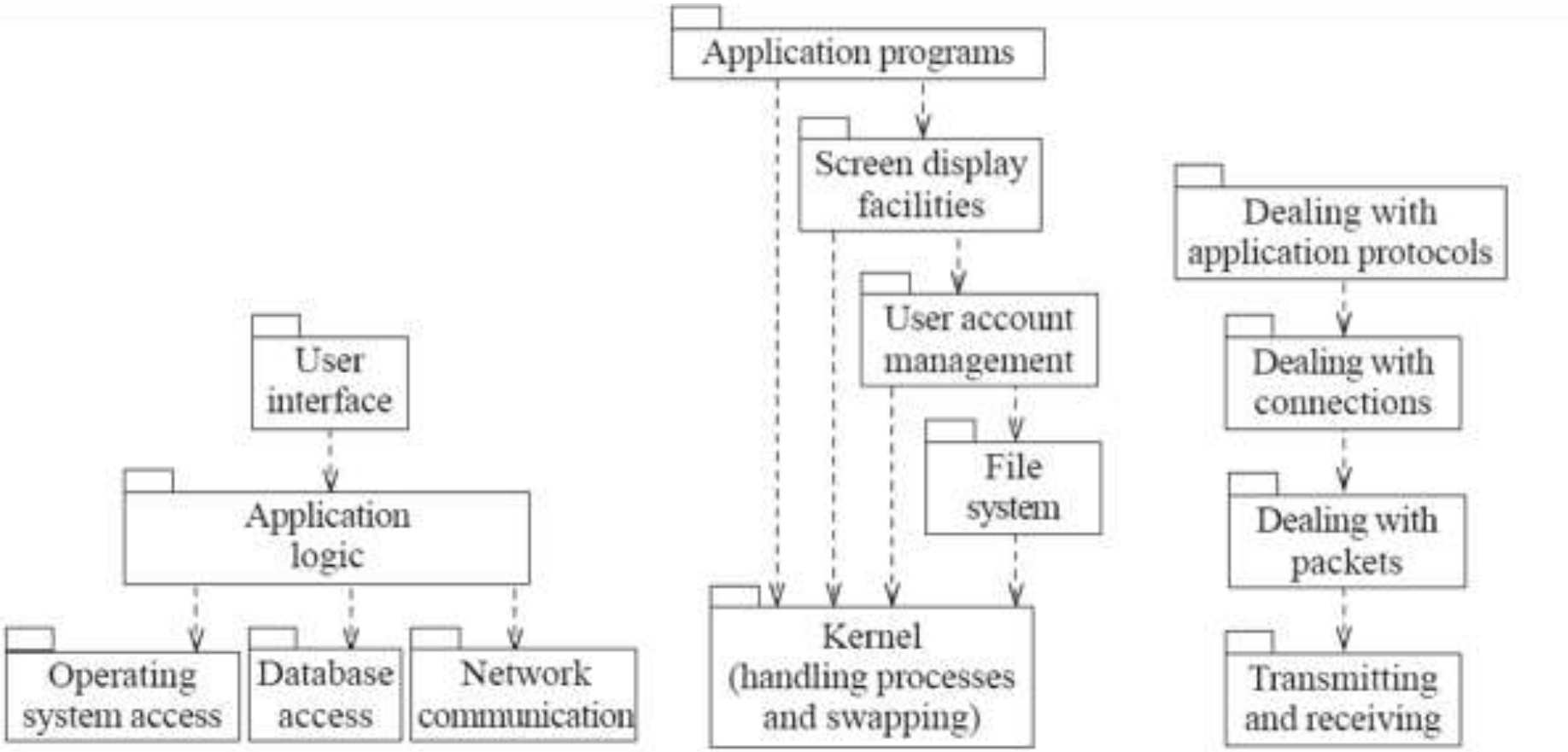
Design principles

- Divide and conquer
- Increase abstraction
- Increase reusability
- Decrease dependencies

Architectural design patterns

- Design patterns: recurring, workable design solutions
- Common architectural design patterns
  - Multi-layer (n-Tier)
  - Model-View-Controller (MVC)
  - Client/Server
    - Service-oriented
    - REST

Multi-layer Architecture

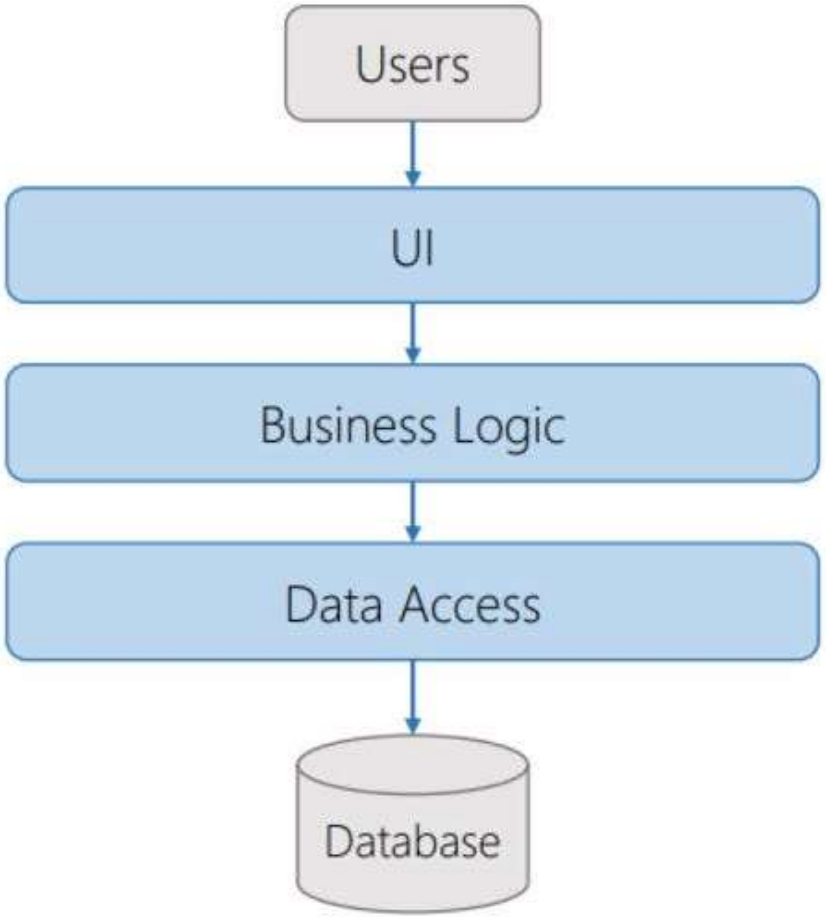


(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

3-tier Architecture



Example: Online shopping app

- Data Access
  - Load/save data objects to databases
- Business Logic
  - Main application (setup data access, create views)
  - Data models for products, orders, customers
  - Controllers to update UI, access DB
- User Interface
  - Different views: Login, Main, Add a new product, Create an order...

Design principles of Multi-layer Architecture

- Divide and conquer
  - Layers can be independently designed
- Increase abstraction
  - Higher layers do not need to know the implementation details of lower layers
- Increase reusability
  - Lower layers can often be designed generically for resue
- Decrease dependencies
  - Layers only interact through API (Application Programming Interfaces)
  - Components of each layer mostly interact to each other

Advantages

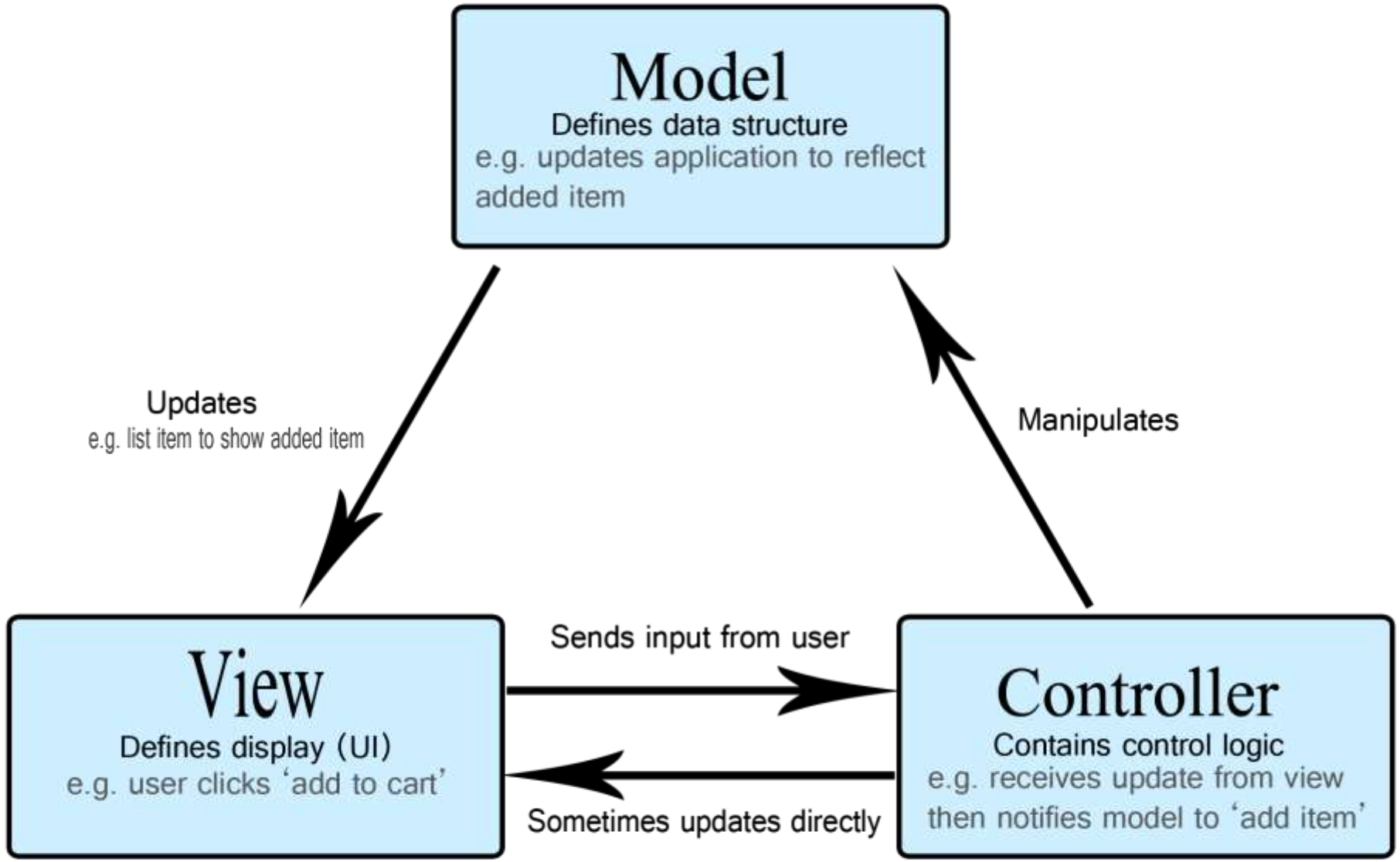
- Maintainable and testable
  - Changes and bug fixes of one layer typically occur on that layer only
  - Each layer can be tested independently
- Easy to update and enhance layers separately.
  - Add a different view: mobile, web, desktop...
  - Change data management system (e.g. from SQLite to MySQL) without changes to other layers
- Easy to assign different people to different tasks
  - Back-end development, front-end (UI/UX) design...

Disadvantages

- Implementation code can skip past layers to create complex interdependencies
- System might have too many modules, thus, difficult to organize
- Multi layers might reduce performance
  - Much of code is for passing data through layers without any transformation
- Layer isolation might be difficult for understanding the architecture without understanding every module

Model-View-Controller (MVC)

- An architectural pattern used to help separate the user interface from other parts of the system
  - Model contains the underlying classes for data
  - View contains objects used to render the appearance of the data in the model on the user interface
  - Controller contains the objects that control and handle the user interaction with the view and the model

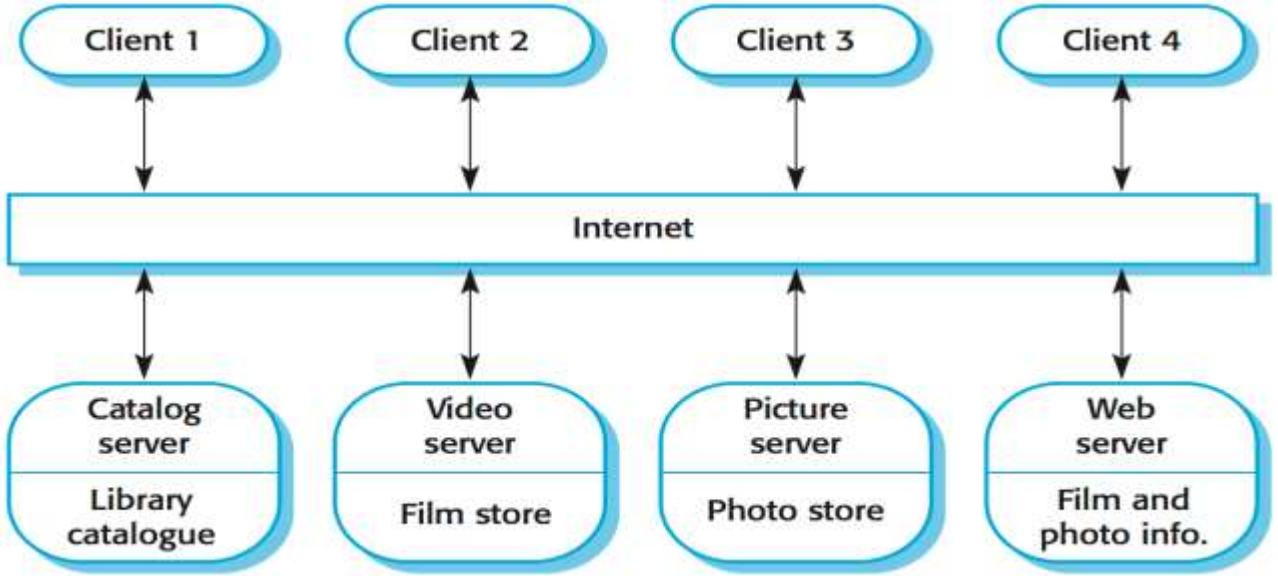

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Design principles of MVC

- Divide and conquer
  - Three components can be independently designed, implemented, tested
- Increase reusability
  - The view and controller normally make extensive use of reusable components for various kinds of UI controls
  - A data model can be used for multiple views and controllers
- Decrease dependencies
  - Views and data models do not interact directly

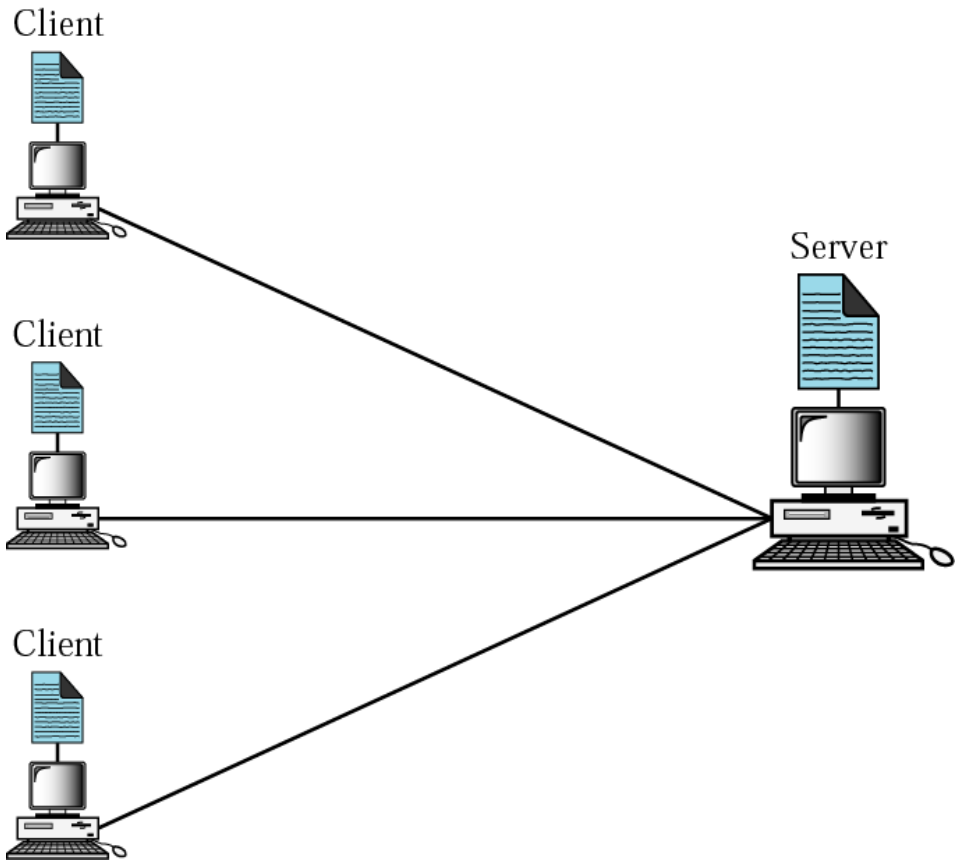
Client/Server Architecture

- System is divided into **client** and **server** components
  - Client and server communicate via **protocols**
  - Client and server can be designed and developed separately



Client-Server Relationship

- Servers
  - Run in background (i.e. infinite)
  - Provide service to any client
  - Typically specialize in providing a certain type of service
  - Listen to a well-known port and passively open connection
- Clients
  - Communicate when needed
  - Actively connect to Server's socket

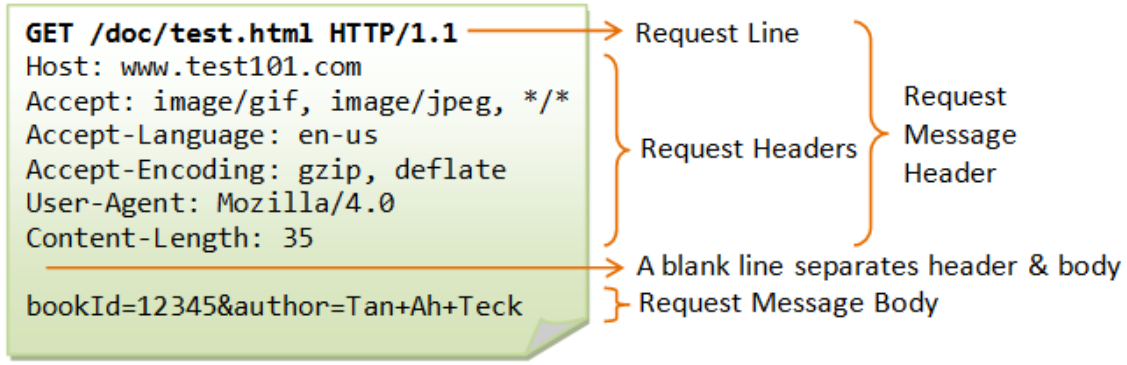


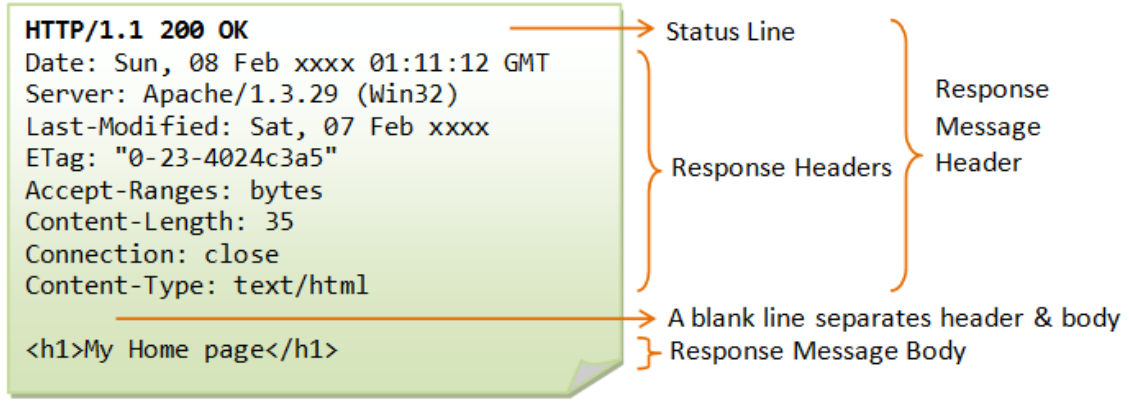
Operation mode of servers

- Iterative: serve one client at-a-time
  - clients wait in a queue
- Concurrent: serve multiple clients concurrently and independently.

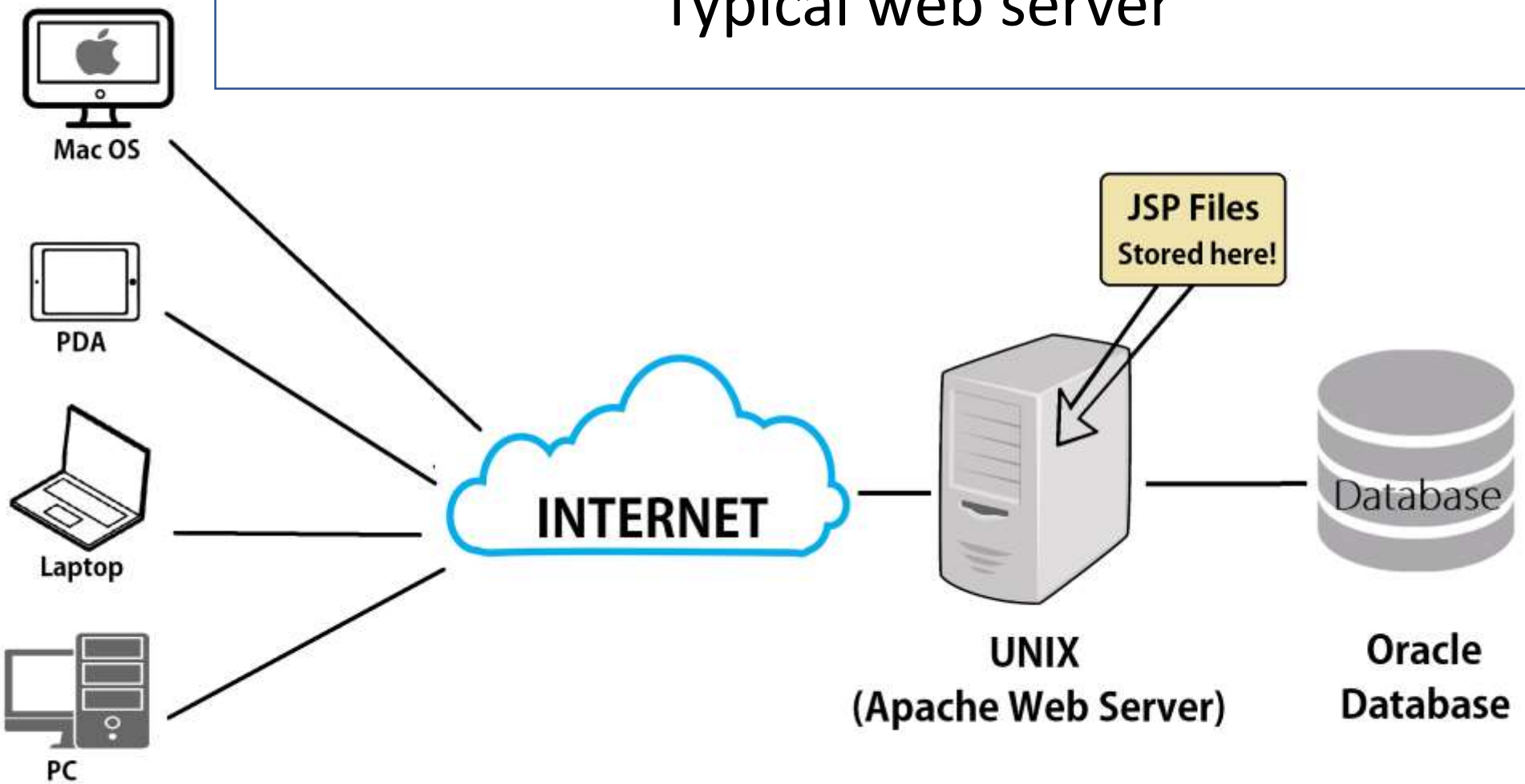
Example: HTTP protocol

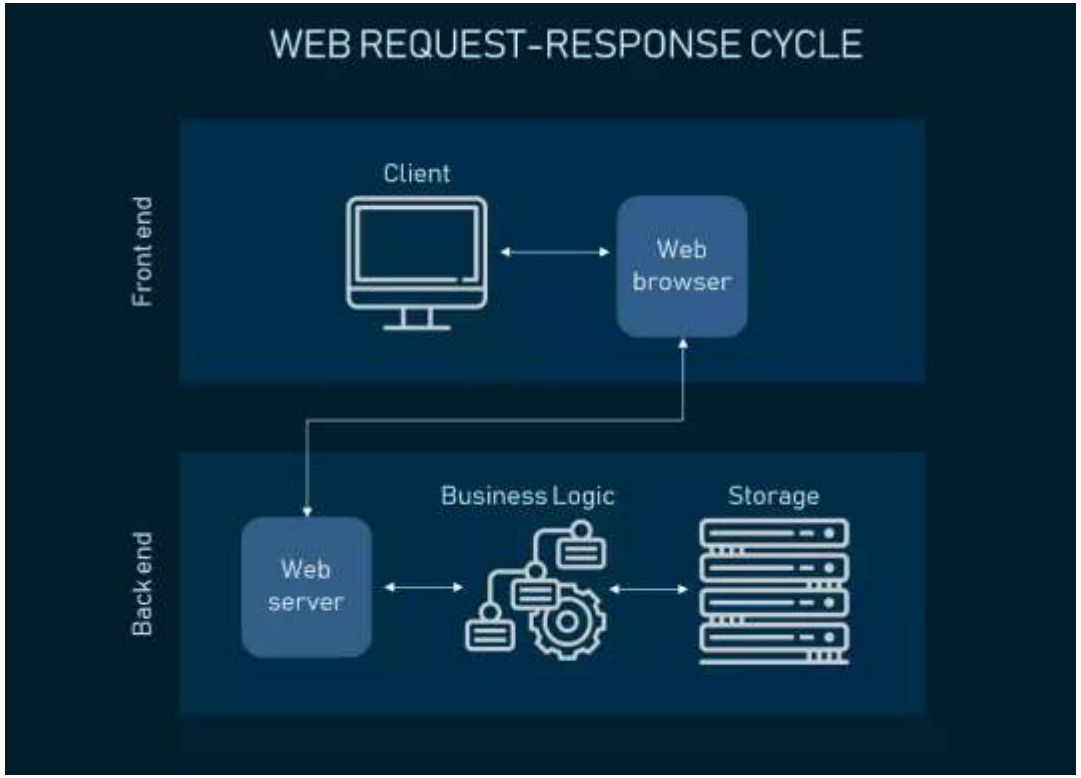
- Clients: web browser: Chrome, Safari, FireFox, Edge...
- Servers: web server (Apache HTTP Server, Microsoft Internet Information Services - IIS)
- Commands: GET, PUT, POST

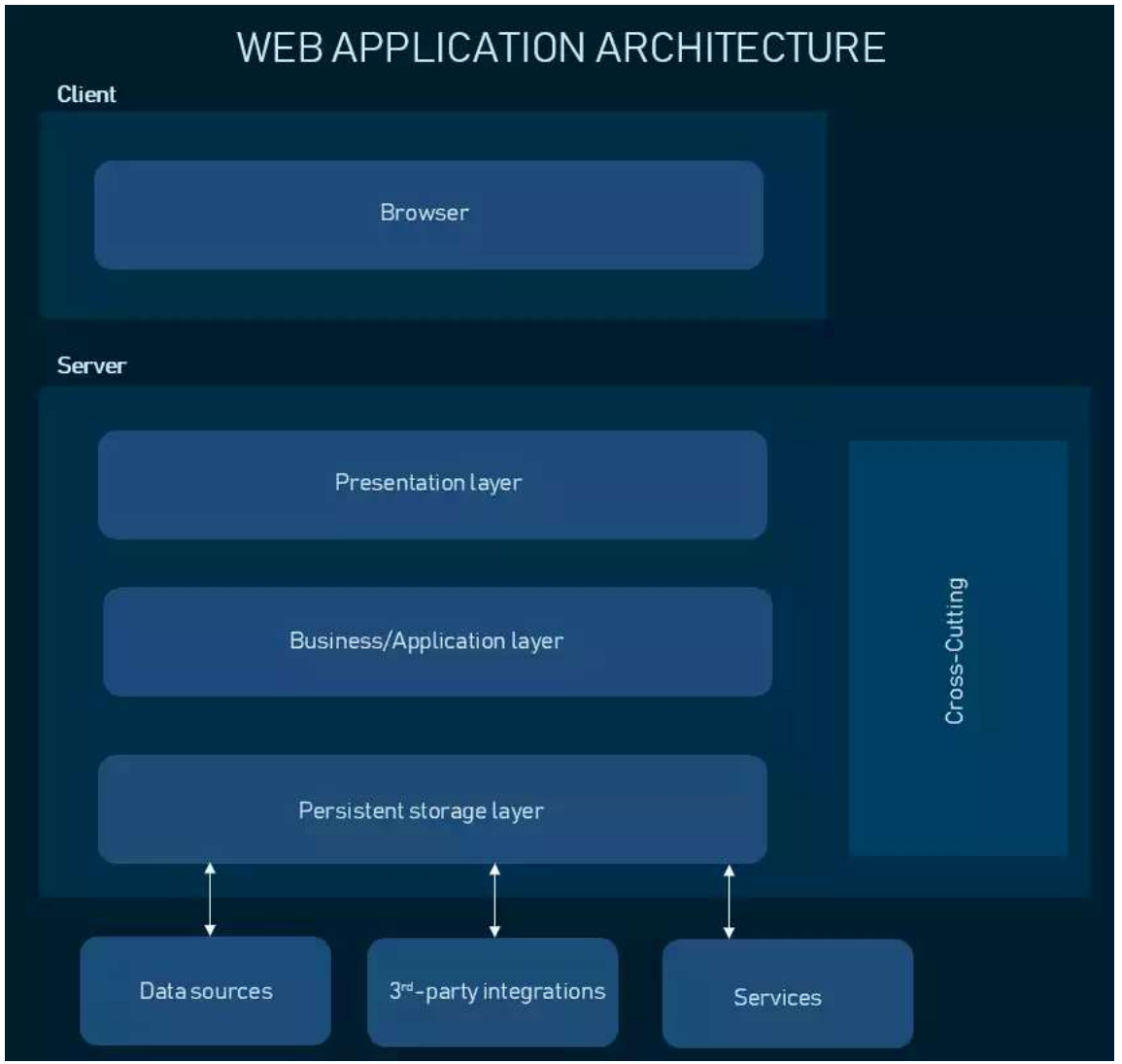


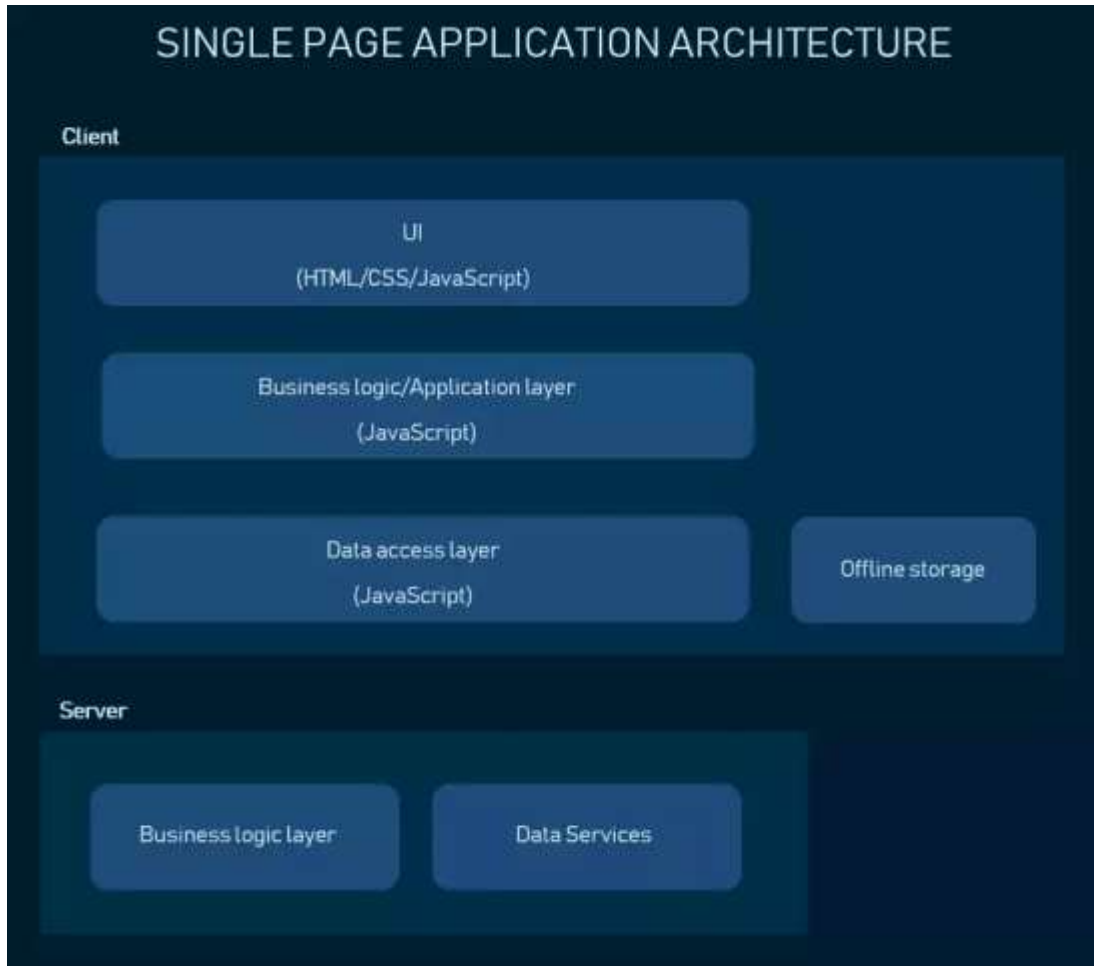


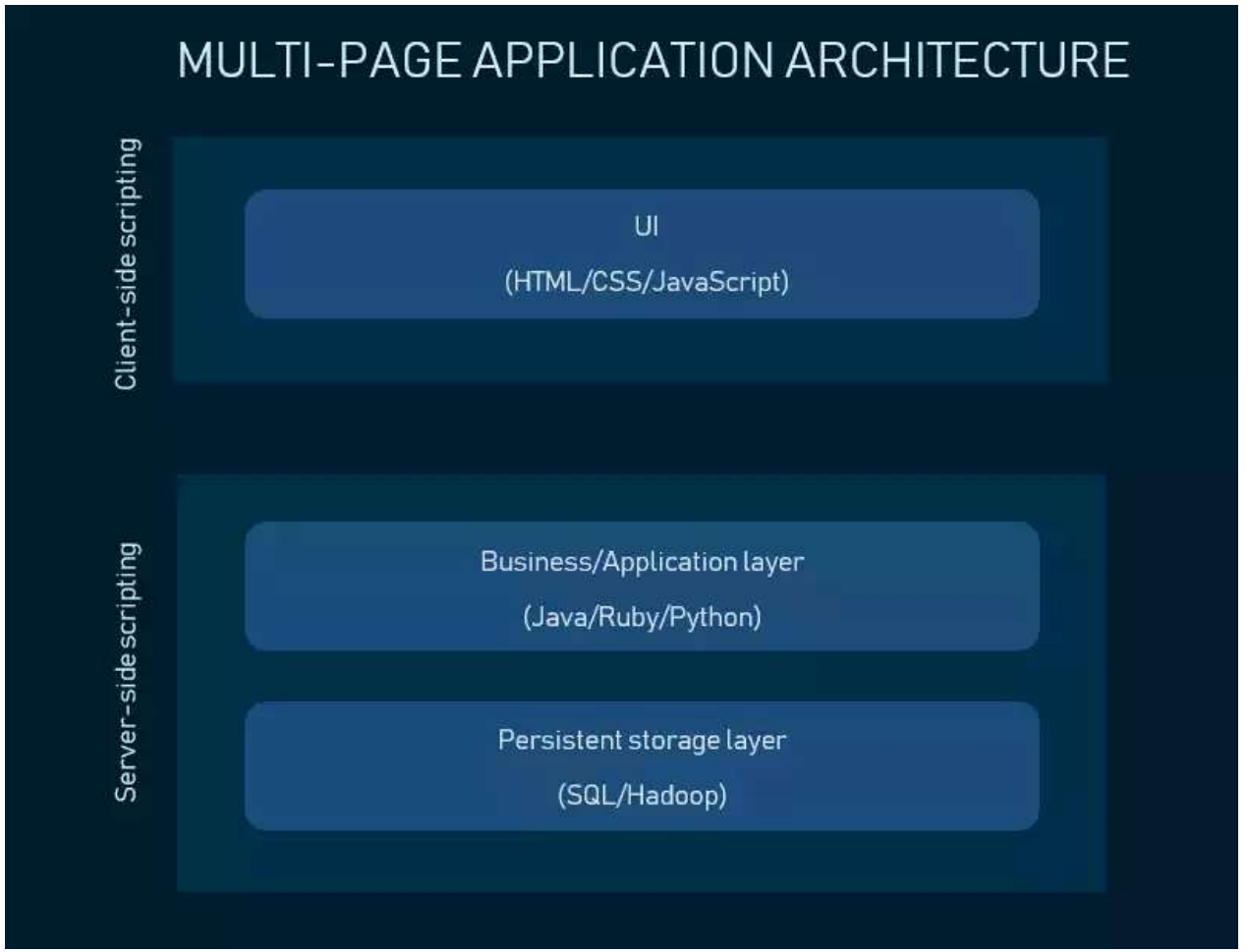
Typical web server




WEB REQUEST-RESPONSE CYCLE
Front end
Client
Web browser
Back end
Business Logic
Storage
Web server

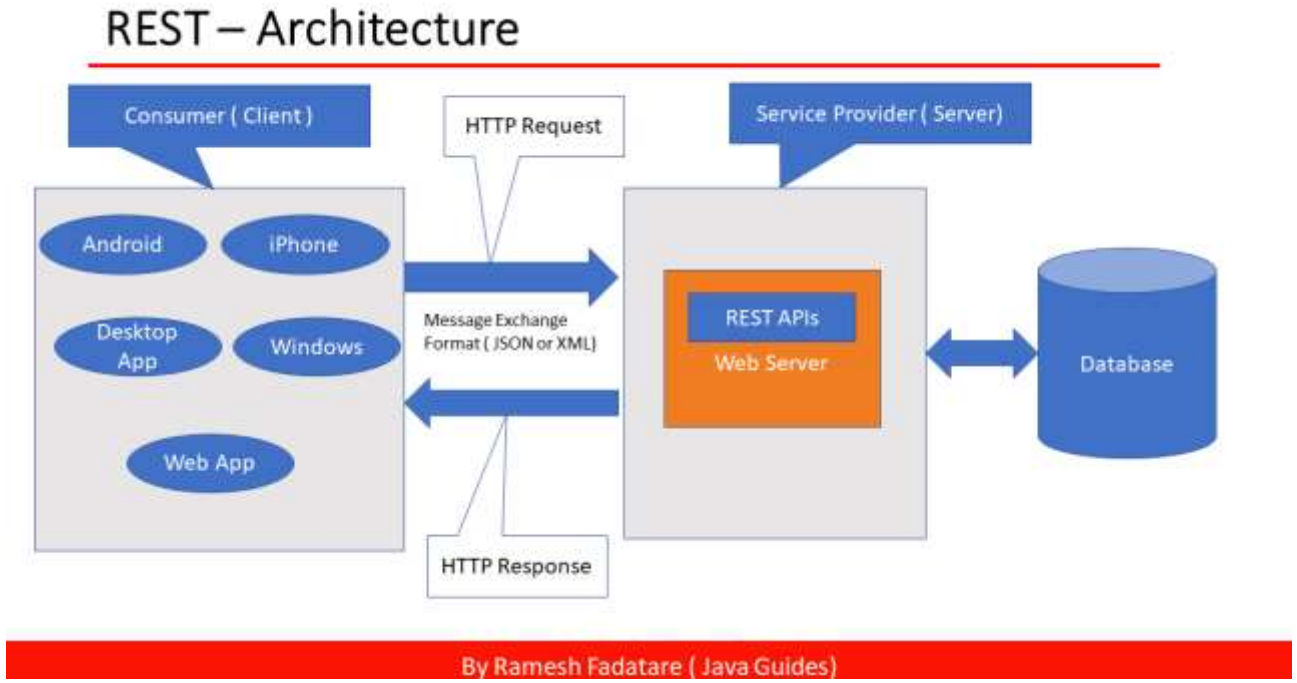

WEB APPLICATION ARCHITECTURE
Client
Browser
Server
Presentation layer
Business/Application layer
Persistent storage layer
Cross-Cutting
Data sources
3rd-party integrations
Services


SINGLE PAGE APPLICATION ARCHITECTURE
Client
UI
(HTML/CSS/JavaScript)
Business logic/Application layer
(JavaScript)
Data access layer
(JavaScript)
Offline storage
Server
Business logic layer
Data Services


MULTI-PAGE APPLICATION ARCHITECTURE
Client-side scripting
UI
(HTML/CSS/JavaScript)
Server-side scripting
Business/Application layer
(Java/Ruby/Python)
Persistent storage layer
(SQL/Hadoop)

REST Architecture

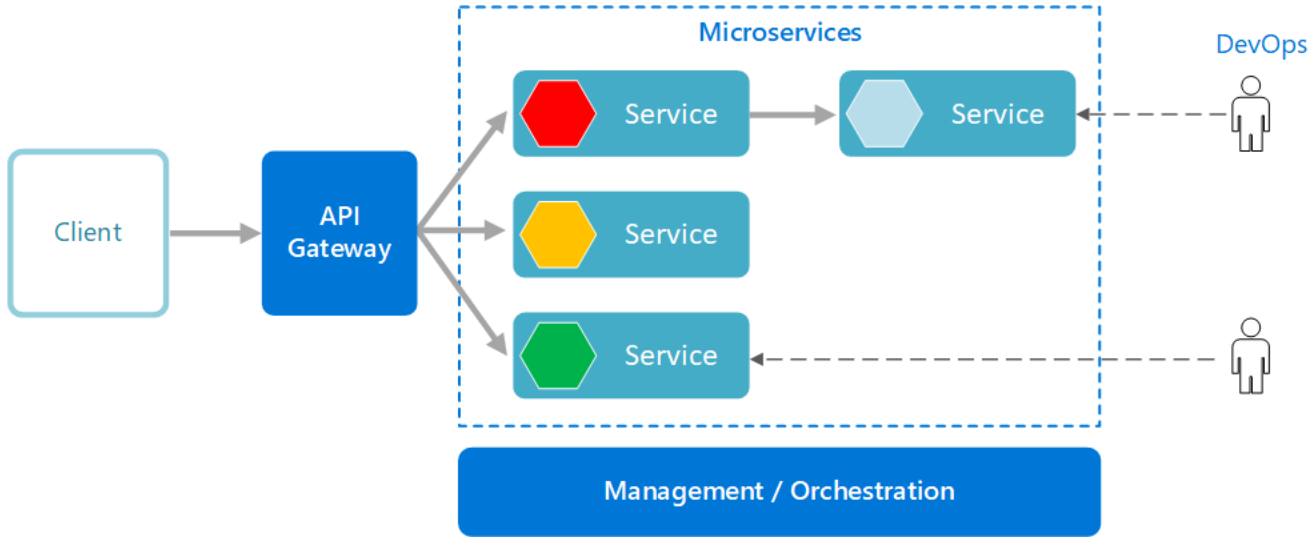
- REST: Representational State Transfer
  - Architectual design pattern for web-based application



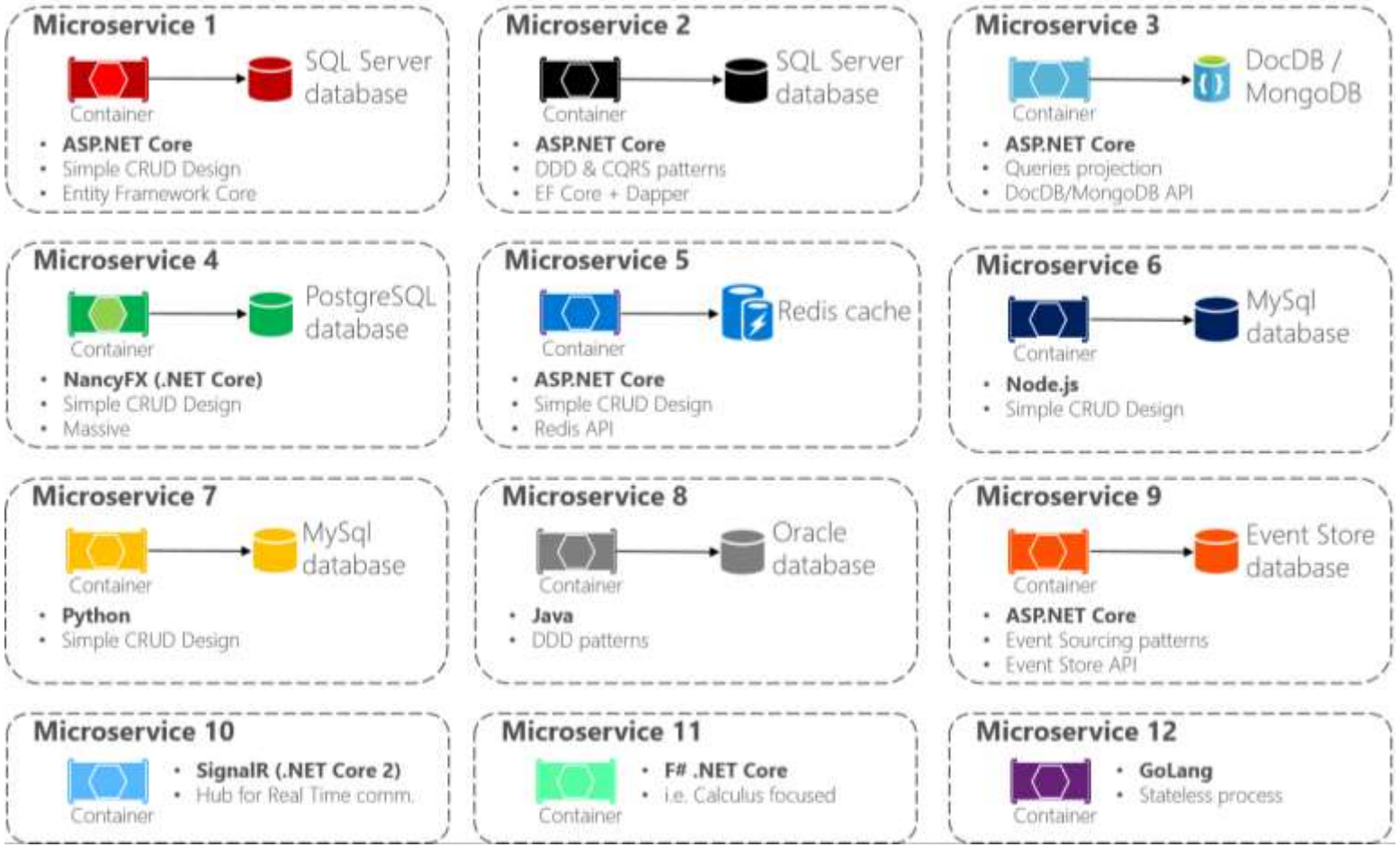
REST Architecture

- Client/server communication is stateless
- Use standard HTTP protocol:
  - GET command to retrieve data (from server to client)
  - POST command to update data (from client to server)
- Client requests for resources (typical data objects) via common interface with server RESTful API endpoints
  - Example:
    - GET /product/100 — Request a product with ID 100
    - GET /product/search?price < 100 — Request all products with price < 100
- Data objects can be in multiple representations: JSON, XML, HTML...

Microservice Architecture



Microservice Architecture



Network Programming

- Sockets are used to exchange messages between network nodes
- Message formats are designed in protocol specification

What is a socket?

- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured, the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

TCP sockets

- Stream-based
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

UDP sockets

- Packet-based
  - unreliable delivery; data can be lost, although this is unusual
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

Ports

- A socket provides an interface to send data to/from the network through a port
- Each host has 65,536 ports
- Some ports are reserved for specific services
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP

The Java.net.Socket Class

- Each Socket object is associated with exactly one remote host Connection is accomplished via construction

  `public Socket(String host, int port)`: connect to specified host/port

  `public Socket(InetAddress address, int port)`: connect to specified IP address and port

- To close a socket:
  - `public void close()`

The Java.net.Socket Class

- Sending and receiving data via output and input streams.

  public InputStream getInputStream() throws IOException

  public OutputStream getOutputStream() throws IOException

- To close a socket:
  - public void close() throws IOException

The Java.net.ServerSocket Class

- A ServerSocket object represents a server socket on a particular port

```
public ServerSocket(int port)
public ServerSocket(int port, int backlog)
public ServerSocket(int port, int backlog, InetAddress bindAddr)
```

  - backlog is the maximum size of the queue of connection requests
- Method accept() listens for incoming connections
  - accept() blocks until a connection is detected.
  - Then accept() returns a java.net.Socket object that is used to perform the actual communication with the client.

TCP Sockets - Server

- Create a ServerSocket object
  ServerSocket servSocket = new ServerSocket(<port>);
- Put the server into a waiting state
  Socket link = servSocket.accept();
- Set up input and output streams
- Send and receive data
  out.println(awaiting data…);
  String input = in.readLine();
- Close the connection
  link.close()

TCP Sockets - Client

- Establish a connection to the server
  Socket link = new Socket(<server>,<port>);
- Set up input and output streams
- Send and receive data
- Close the connection

Set up input and output streams

- Once a socket has connected
  - send data via the output stream
  - receive data via input stream

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(link.getInputStream()));

PrintWriter out = new PrintWriter(link.getOutputStream(),true);
```