**TABLE OF CONTENTS**

[**1. Introduction to Software Architecture**](#_heading=h.ntnx4a2rqjgi) **2**

[**2. Architecture Structures**](#_heading=h.30j0zll) **2**

[Module Structure:(In Detail)](#_heading=h.ecvjt5cu37z) **2**

[Component and Connector:(In Detail)](#_heading=h.fyd48vkrhwh6) **3**

[Allocation:](#_heading=h.o7x99dk7b0gc) **3**

[**3. Architecture Patterns**](#_heading=h.3znysh7) **3**

[Layered](#_heading=h.7t5xz7aihfr8)

[Client-Server](#_heading=h.oxigl1e1quh1)

[Master-slave](#_heading=h.piu7b53pnkzv)

[Pipes and filters](#_heading=h.x53v4dihdqxc)

[Broker](#_heading=h.cu2as7ndgpo5)

[Peer-to-peer](#_heading=h.x8p515p6alm2)

[Event bus](#_heading=h.u2oafvdk4dh4)

[MVC](#_heading=h.li0y80p0z0t5)

[Blackboard](#_heading=h.cu2as7ndgpo5)

**4**[**. What Makes a Good Architecture?**](#_heading=h.tpj0bd1mu9c) **12**

**5**[**. Why is Software architecture important?**](#_heading=h.ausylxg0ld24) **13**

**6**[**. Monolithic Architecture**](#_heading=h.fbx3w3cckbnb) **15**

[**7.Distributed Architecture**](#_heading=h.24krrheissxm)  **16**

[**8.Synchronous and Asynchronous communication**](#_heading=h.txeke28wvh3s) **17**

[**9.Microkernel Architecture**](#_heading=h.b66qol409od7) **18**

[**10.Serverless Architecture**](#_heading=h.oasrzlnf8ow4) **19**

[**11.Microservice Architecture**](#_heading=h.cyt3x01qm8x6) **20**

[**12.Monolithic Vs Microservices**](#_heading=h.cj09ogn3t8yw) **21**

[**13. C4 model**](#_heading=h.1bf61e7seur4)  **22**

# 1. Introduction to Software Architecture

Software architecture provides a design plan, a blueprint of a system and an abstraction to help manage the complexity of a system. It is the set of structures needed to describe the system, which consists of software elements, relations among them and properties of both.

Software architecture can be analyzed, designed, documented and implemented using known techniques that will support in achieving business goals.

# 2. Architecture Structures

Views and structure makes it easy to communicate meaningfully about architecture.

***View :*** A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.

Example: A module view is the representation of a structure, as documented by and used by some system stakeholders.

***Structure:*** A structure is the set of elements itself, as they exist in software or hardware.

Example: A module structure is the set of the system's modules and their organization.

**The three categories of architectural structures are:**

* **Module structure :**

1. The elements are modules, which are units of implementation.
2. Modules represent a code-based way of considering the system.
3. They are assigned areas of functional responsibility.
4. Module structures allow us to answer questions such as What is the primary functional responsibility assigned to each module?

* **Component and Connector structures:**

1. Here the elements are runtime components and connectors.
2. Component-and-connector structures help answer questions such as What are the major executing components and how do they interact?

* **Allocation structures:**

1. Allocation structures show the relationship between the software elements and the elements in one or more external environments in which the software is created and executed.
2. They answer questions such as What processor does each software element execute on?

##### **Module Structure:(In Detail)**

Module-based structures include the following:

**Decomposition structure:** The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood.

***Uses structure:*** The units are related by the *uses* relation. One unit uses another if the correctness of the first requires the presence of a correct version of the second.

***Layered structure:*** When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality. Layers are often designed as abstractions that hide implementation specifics below from the layers above.

***Class structure:***The module units in this structure are called classes. The relation is "inherits-from" or "is-an-instance-of." The class structure allows us to reason about re-use and the incremental addition of functionality.

##### **Component and Connector:(In Detail)**

These structures include the following.

* ***Process, or communicating processes***:

1. This deals with the dynamic aspects of a running system.
2. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations.

* ***Concurrency****:*

1. This component-and-connector structure determines opportunities for parallelism and the locations where resource contention may occur.
2. The units are components and the connectors are "logical threads."

* ***Shared data*, or *repository:***

1. This structure comprises components and connectors that create, store, and access persistent data.
2. It shows how data is produced and consumed by runtime software elements.

* ***Client-server:***

1. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work.
2. This is useful for separation of concerns, for physical distribution, and for load balancing.

##### **Allocation:**

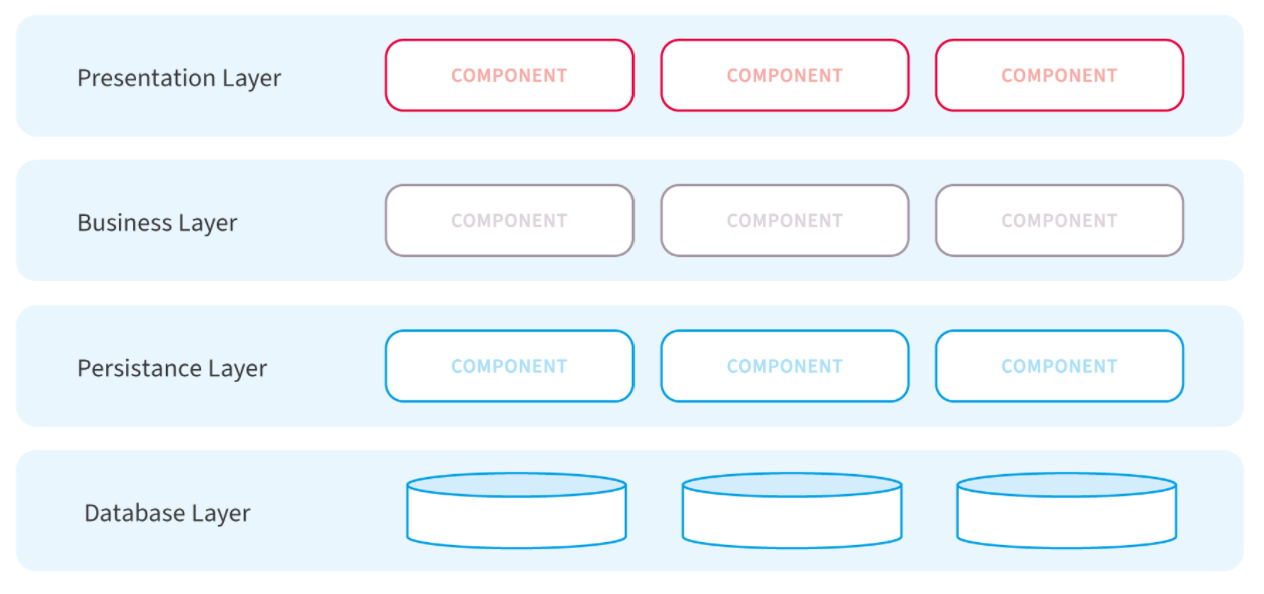
Allocation structures include the following.

* ***Deployment:*** The deployment structure shows how software is assigned to hardware-processing and communication elements.
* ***Implementation:*** This structure shows how software elements are mapped to the file structures in the system's development, integration, or configuration control environments.
* ***Work assignment:*** This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.

# 3. Architecture Patterns

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design patterns but have a broader scope.

### Layered Architecture:



**Definition :**

A layered architecture is a set of coherent related functionalities.

They are unidirectional as they use the service of the layer below it.

**Key characteristics of this pattern are as follows:**

* The outermost layer is where the data enters the system. The data passes through the subsequent layers to reach the innermost layer, which is the database layer.
* Simple implementations of this pattern have at least 3 layers, namely, a presentation layer, an application layer, and a data layer. Users access the presentation layer using a GUI, whereas the application layer runs the business logic. The data layer has a database for the storage and retrieval of data.

**Usage:** General desktop applications,E commerce web applications.

This pattern has the following **advantages**:

* Maintaining the software is easy since the tiers are segregated.
* Development teams find it easy to manage the software infrastructure, therefore, it’s easy to develop large-scale web and cloud-hosted apps.

There are a few **disadvantages** too, as follows:

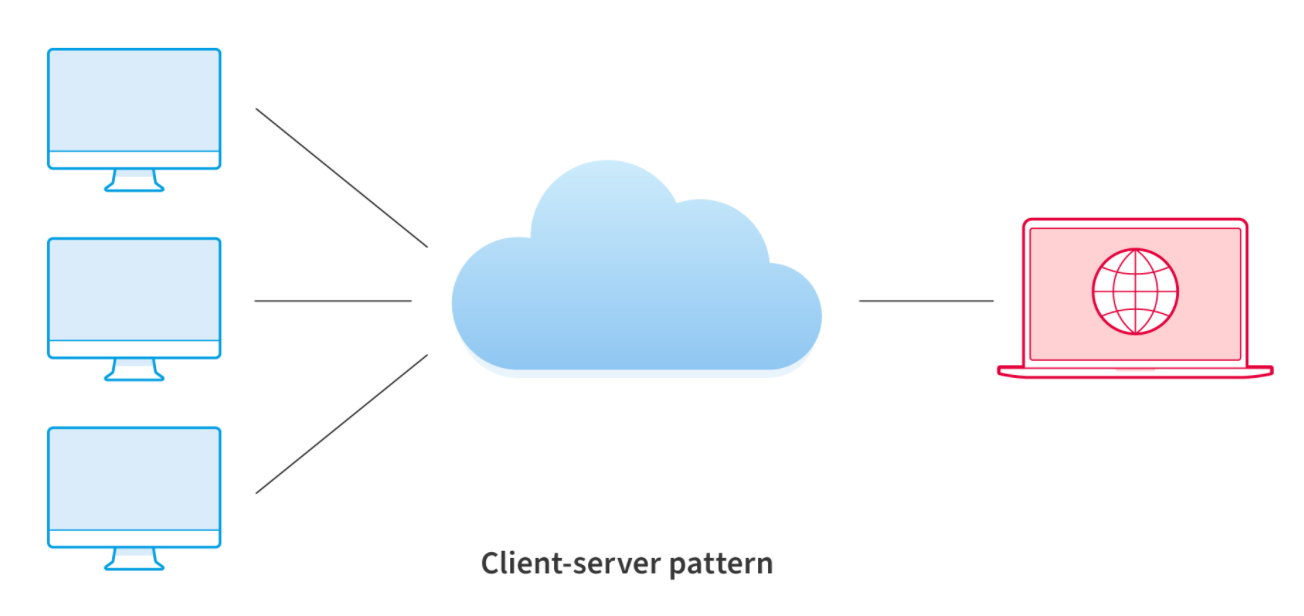
* The code can become too large.
* A considerable part of the code only passes data between layers instead of executing any business logic, which can adversely impact performance.

### 

### 

### 

### Client-Server Architecture:

******

**Definition:**

This pattern consists of two parties: a **server** and multiple **clients**.

The server component will provide services to multiple client components.

Clients request services from the server and the server provides relevant services to those clients.

The following are key **characteristics** of this pattern:

* Client components send requests to the server, which processes them and responds back.
* When a server accepts a request from a client, it opens a connection with the client over a specific protocol.
* Servers can be stateful or stateless.

**Usage:** Online applications such as email, document sharing and banking.

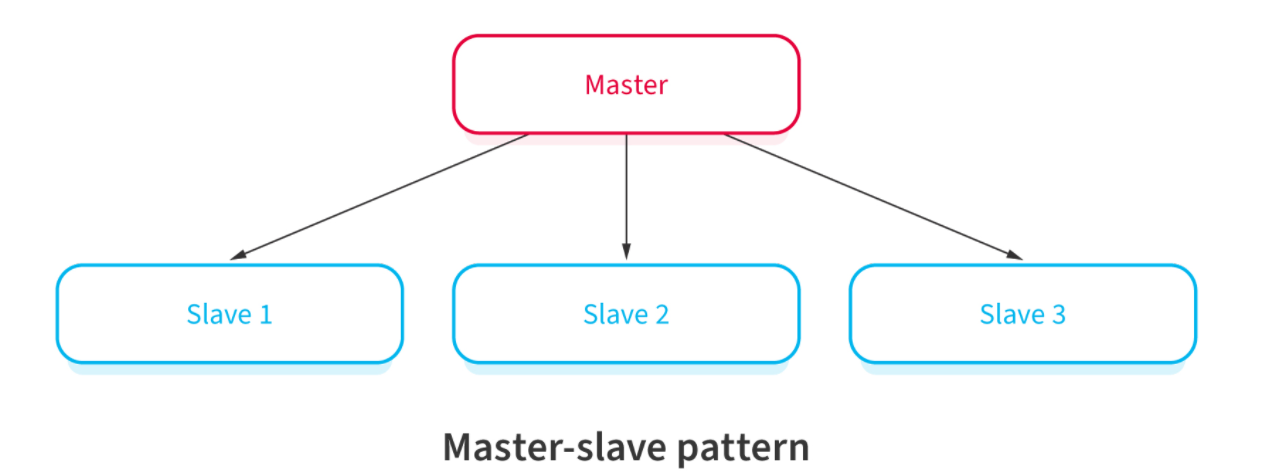
The pattern has several **advantages**, as follows:

* Clients access data from a server using authorized access, which improves the sharing of data.
* Accesses a service via a user interface due to which there’s no need to run terminal sessions or command prompts.
* Client-server applications can be built irrespective of the platform or technology stack.
* This is a distributed model with specific responsibilities for each component, which makes maintenance easier.

Some disadvantages of the client-server architecture are as follows:

* The server can be overloaded when there are too many requests.
* A central server to support multiple clients represents a ‘single point of failure’.

### Master-Slave Architecture:

******

**Definition:**

This pattern consists of two parties; **master** and **slaves**.

The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

It is useful when clients make multiple instances of the same request. The requests need simultaneous handling.

The key **characteristics** are:

* The master launches slaves when it receives simultaneous requests.
* The slaves work in parallel, and the operation is complete only when all slaves complete processing their respective requests.

**Usage:** In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.

**Advantages** of this pattern are the following:

* Applications read from slaves without any impact on the master.
* Taking a slave offline and the later synchronization with the master requires no downtime.

There are a few **disadvantages** to this pattern:

* This pattern doesn’t support automated fail-over systems since a slave needs to be manually promoted to a master if the original master fails.
* Failure of a master typically requires downtime and restart, moreover, data loss can happen in such cases.

### ***Pipe-filter Architecture:***

**Definition :**

This pattern can be used to structure systems which produce and process a stream of data.

Each processing step is enclosed within a **filter** component.

Data to be processed is passed through **pipes**.

These pipes can be used for buffering or for synchronization purposes.

Below are the **characteristics**:

* The code for each task is relatively small. You treat it as one independent ‘filter’.
* You can deploy, maintain, scale, and reuse code in each filter.
* The stream of data that each filter processes passes through ‘pipes’.

## Usage: Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.Workflows in bioinformatics.

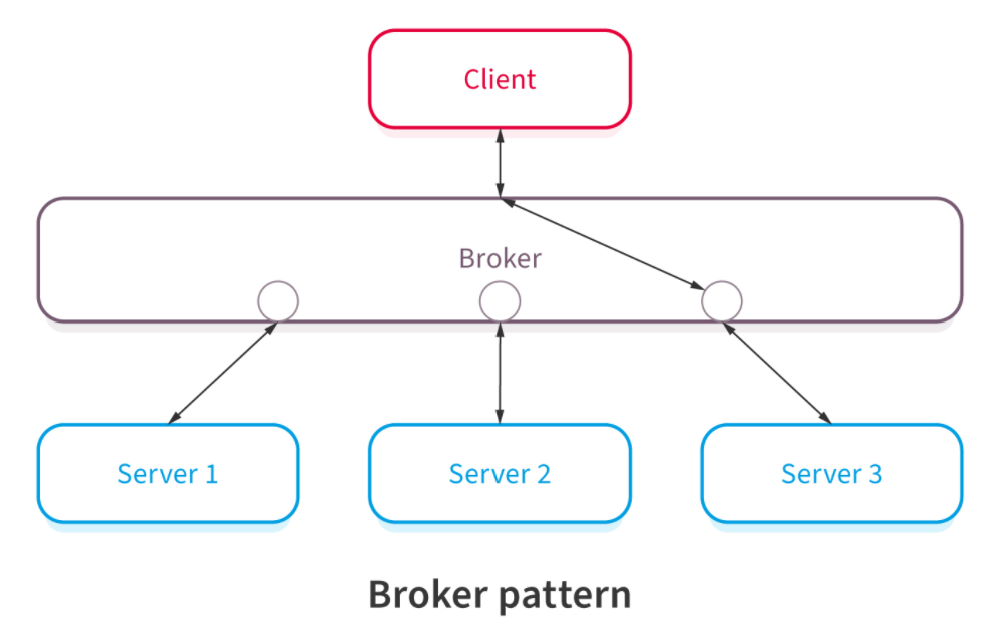
Below are the following **advantages**:

* There are repetitive steps such as reading the source code, parsing, generating code, etc. These can be easily organized as separate filters.
* Each filter can perform its processing in parallel if the data input is arranged as streams using pipes.
* It’s a resilient model since the pipeline can reschedule the work and assign to another instance of that filter.

Few **disadvantages** are:

* This pattern is complex.
* Data loss between filters is possible in case of failures unless you use a reliable infrastructure

### Broker Architecture Pattern:

******

**Definition :**

This pattern is used to structure distributed systems with decoupled components.

These components can interact with each other by remote service invocations.

A broker component is responsible for the coordination of communication among components.

Servers publish their capabilities to a broker.

Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

**Characteristics** of Broker Pattern are:

* A broker component coordinates requests and responses between clients and servers.
* The broker has the details of the servers and the individual services they provide.
* The main components of the broker architectural pattern are clients, servers, and brokers. It also has bridges and proxies for clients and servers.
* Clients send requests, and the broker finds the right server to route the request to.
* It also sends the responses back to the clients

**Usage**:Message broker software such as [ActiveMQ](https://en.wikipedia.org/wiki/Apache_ActiveMQ), [Apache Kafka](https://en.wikipedia.org/wiki/Apache_Kafka), [RabbitMQ](https://en.wikipedia.org/wiki/RabbitMQ) and [JBoss Messaging](https://en.wikipedia.org/wiki/JBoss_Messaging).

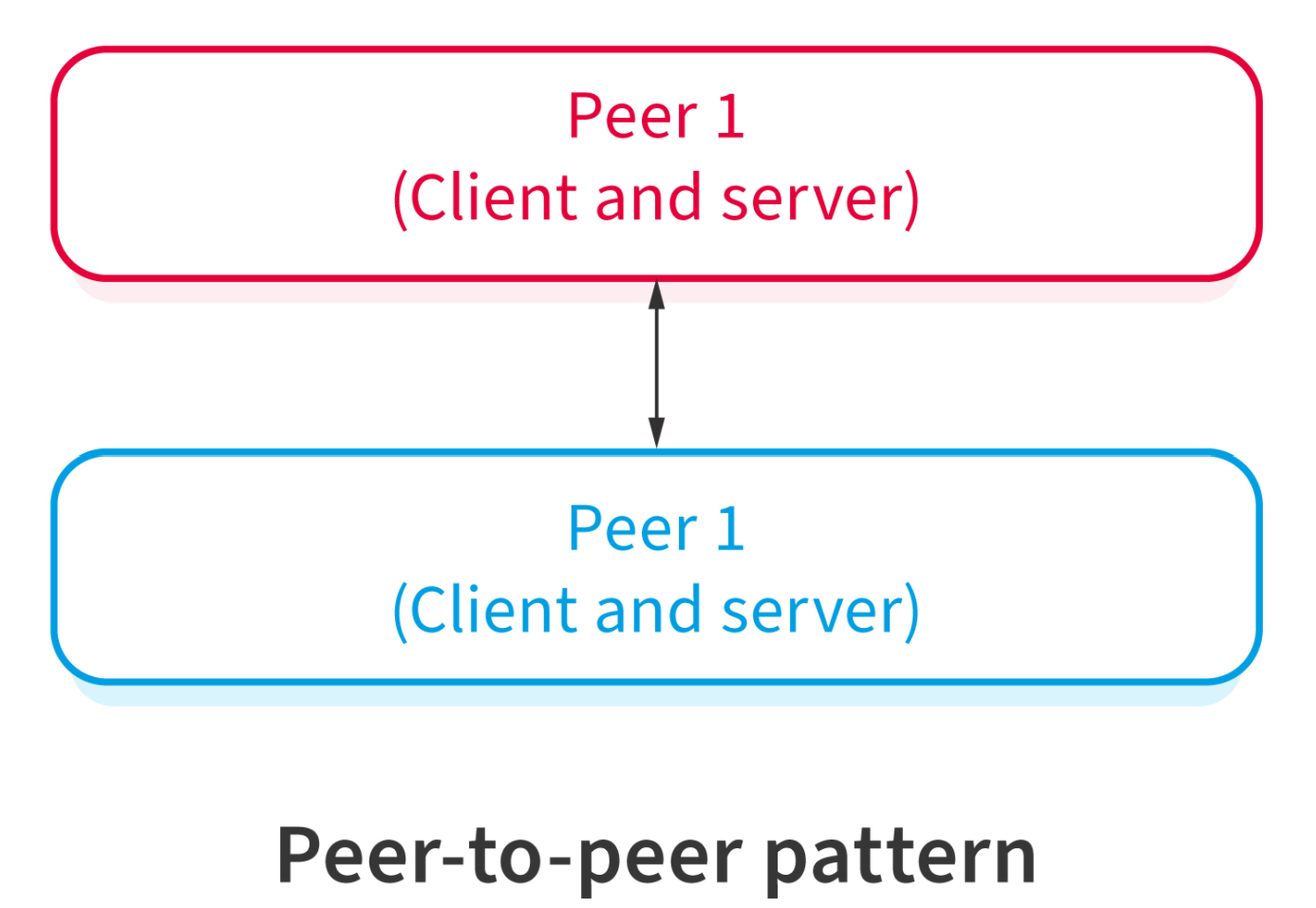
The pattern has a few distinct **advantages** :

* Developers face no constraints due to the distributed environment, they simply use a broker.
* This pattern helps using object-oriented technology in a distributed environment.

**Disadvantages** are:

* Messages not guaranteed delivery. Some may be dropped.
* No "discovery". Relationships between components need to be more managed.

### Peer-to-peer Architecture Pattern:

******

**Definition:**

Each individual component is known as peers.

Peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers.

A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

Key **characteristics** of the P2P pattern are as follows:

* There isn’t one central server, with each node having equal capabilities.
* Each computer can function as a client or a server.
* When more computers join the network, the overall capacity of the network increases.

## Usage: File-sharing networks such as [Gnutella](https://en.wikipedia.org/wiki/Gnutella) and [G2](https://en.wikipedia.org/wiki/Gnutella2),Multimedia protocols such as [P2PTV](https://en.wikipedia.org/wiki/P2PTV) and [PDTP](https://en.wikipedia.org/wiki/Peer_Distributed_Transfer_Protocol) and Cryptocurrency-based products such as [Bitcoin](https://bitcoin.org/en/) and [Blockchain](https://www.blockchain.com/)

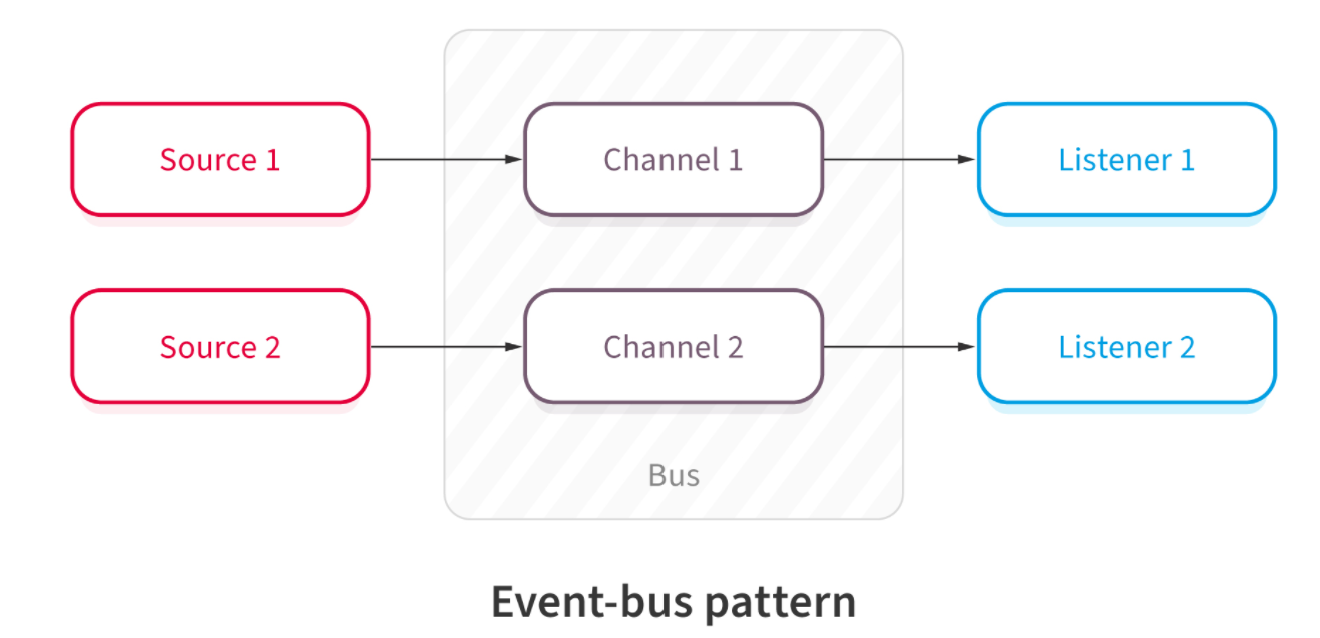
**Advantages** of a P2P network are as follows:

* P2P networks are decentralized, therefore, they are more secure.
* Hackers can’t destroy the network by compromising just one server.

**Disadvantages** of Peer to Peer Computing:

* It is difficult to backup the data as it is stored in different computer systems and there is no central server.
* It is difficult to provide overall security in the peer to peer network as each system is independent and contains its own data.

### Event Bus Architecture Pattern:

**

**Definition:**

This pattern primarily deals with events and has 4 major components; event source, event listener, channel and event bus.

Sources publish messages to particular channels on an event bus.

Listeners subscribe to particular channels.

Listeners are notified of messages that are published to a channel to which they have subscribed before.

It has the following **characteristics**:

* A central agent, which is an event-bus, accepts the input.
* Different components handle different functions, therefore, the event-bus routes the data to the appropriate module.
* Modules that don’t receive any data pertaining to their function will remain inactive.

**Usage:** Android development,Notification services

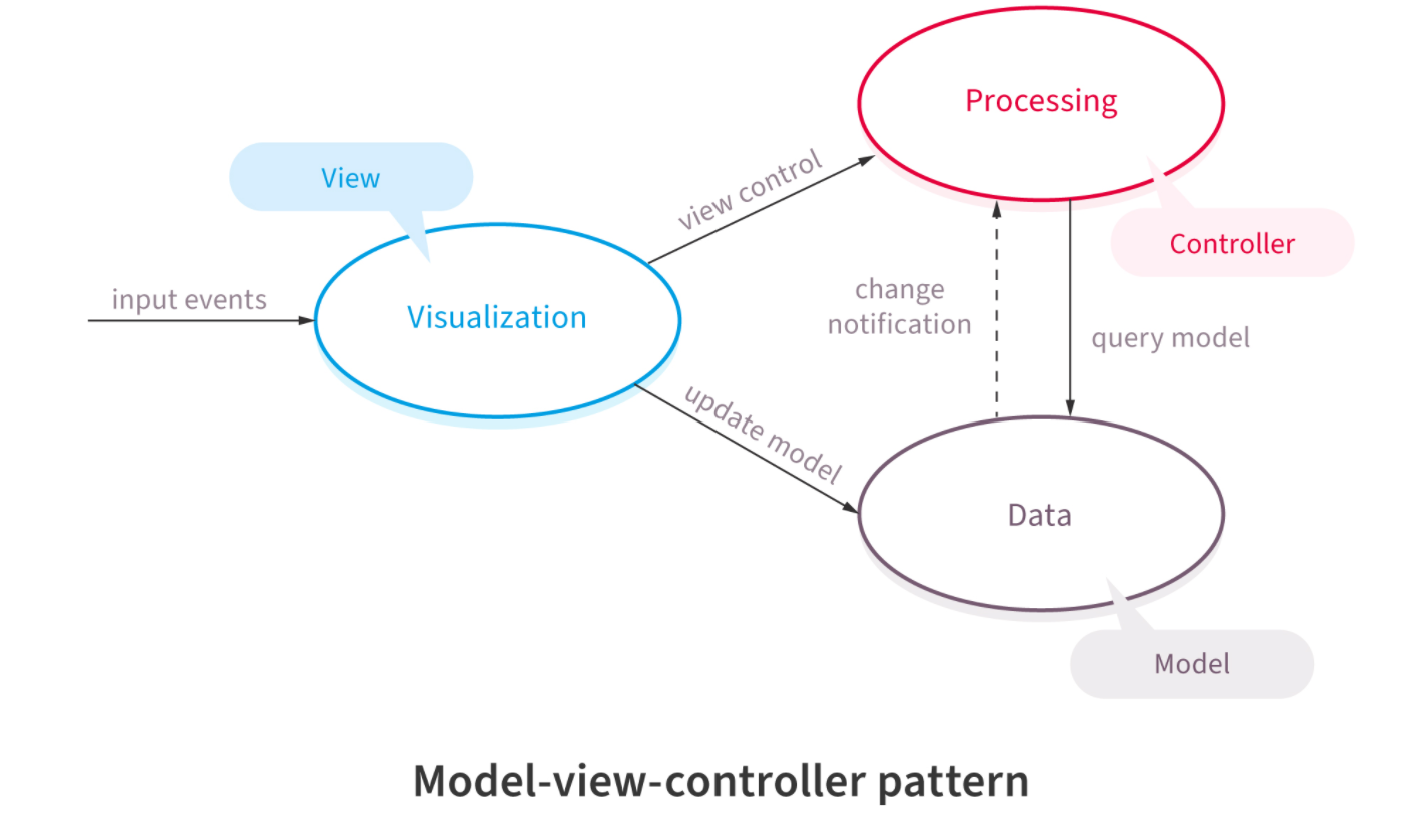
The **advantages** of this pattern are as follows:

* This pattern helps developers handle complexity.
* It’s a scalable architecture pattern.
* This is an extensible architecture, new functionalities will only require a new type of events.

Some **disadvantages** of this pattern are as follows:

* Testing of interdependent components is an elaborate process.
* If different components handle the same event require complex treatment to error-handling.
* Some amount of messaging overhead is typical of this pattern.

### Model-View-Controller Architecture Pattern:

******

**Definition :**

This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

model — contains the core functionality and data.

view — displays the information to the user

controller — handles the input from the user

Following are its’ **characteristics**:

* There are three building blocks here, namely, model, view, and controller.
* The application data resides in the model.
* Users see the application data through the view, however, the view can’t influence what the user will do with the data.
* The controller is the building block between the model and the view. View triggers events, subsequently, the controller acts on it. The action is typically a method call to the model. The response is shown in the view.

## Usage: Architecture for World Wide Web applications in major programming languages,Web frameworks such as [Django](https://en.wikipedia.org/wiki/Django_(web_framework)) and [Rails](https://en.wikipedia.org/wiki/Ruby_on_Rails).

Its’ **advantages** are as follows:

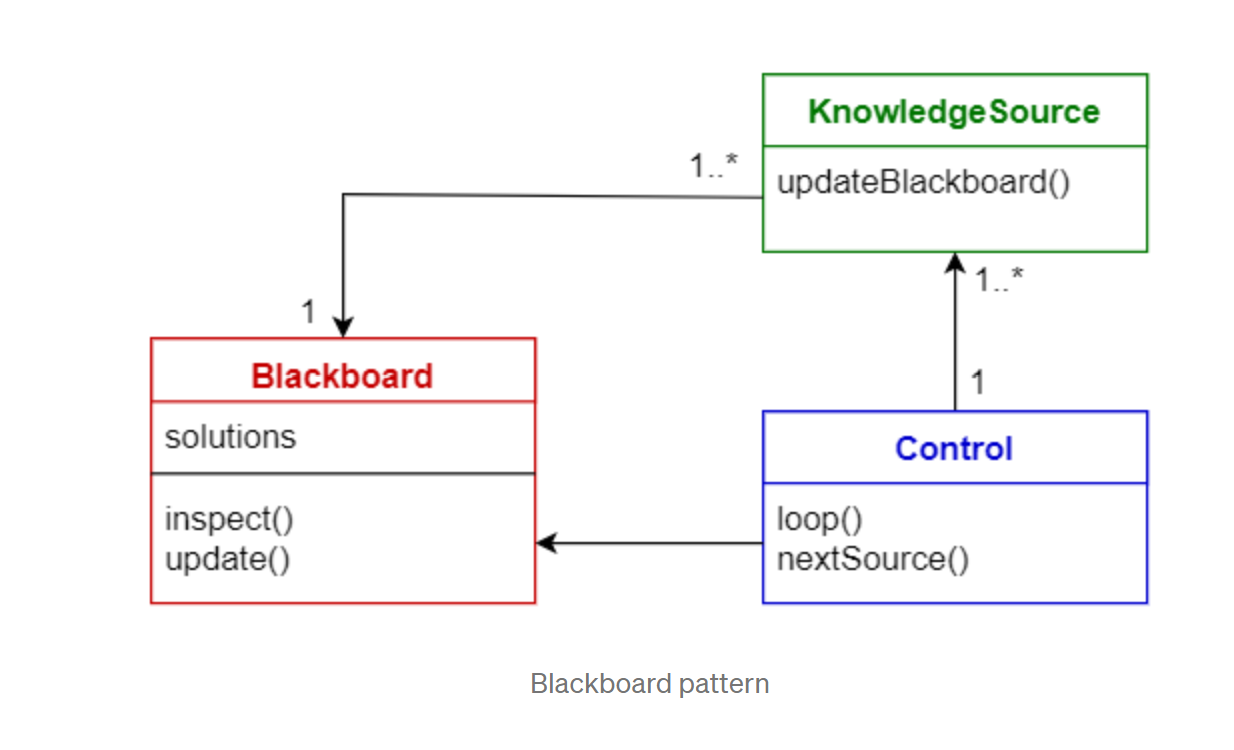
* Using this model expedites the development.
* Development teams can present multiple views to users.
* Changes to the UI are common in web applications, however, the MVC pattern doesn’t need changes for it.
* The model doesn’t format data before presenting to users, therefore, you can use this pattern with any interface.

**Disadvantages.:**

* With this pattern, the code has new layers, making it harder to navigate the code.
* There is typically a learning curve for this pattern, and developers need to know multiple technologies.

### 

### Blackboard Architecture:

******

**Definition :**

This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.

blackboard — a structured global memory containing objects from the solution space

knowledge source — specialized modules with their own representation

control component — selects, configures and executes modules.

All the components have access to the blackboard.

Components may produce new data objects that are added to the blackboard.

Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

**Characteristics** are as follows:

* When you deal with an emerging domain like AI or ‘Machine Learning’ (ML), you don’t necessarily have a settled architecture pattern to use. You start with the blackboard pattern, subsequently, when the domain matures, you adopt a different architecture pattern.
* The application system stores the relevant information in the blackboard.
* The knowledge resources could be algorithms in the AI or ML context that collect information and update the blackboard.
* The controller reads from the blackboard and updates the application ‘assets’, for e.g., robots.

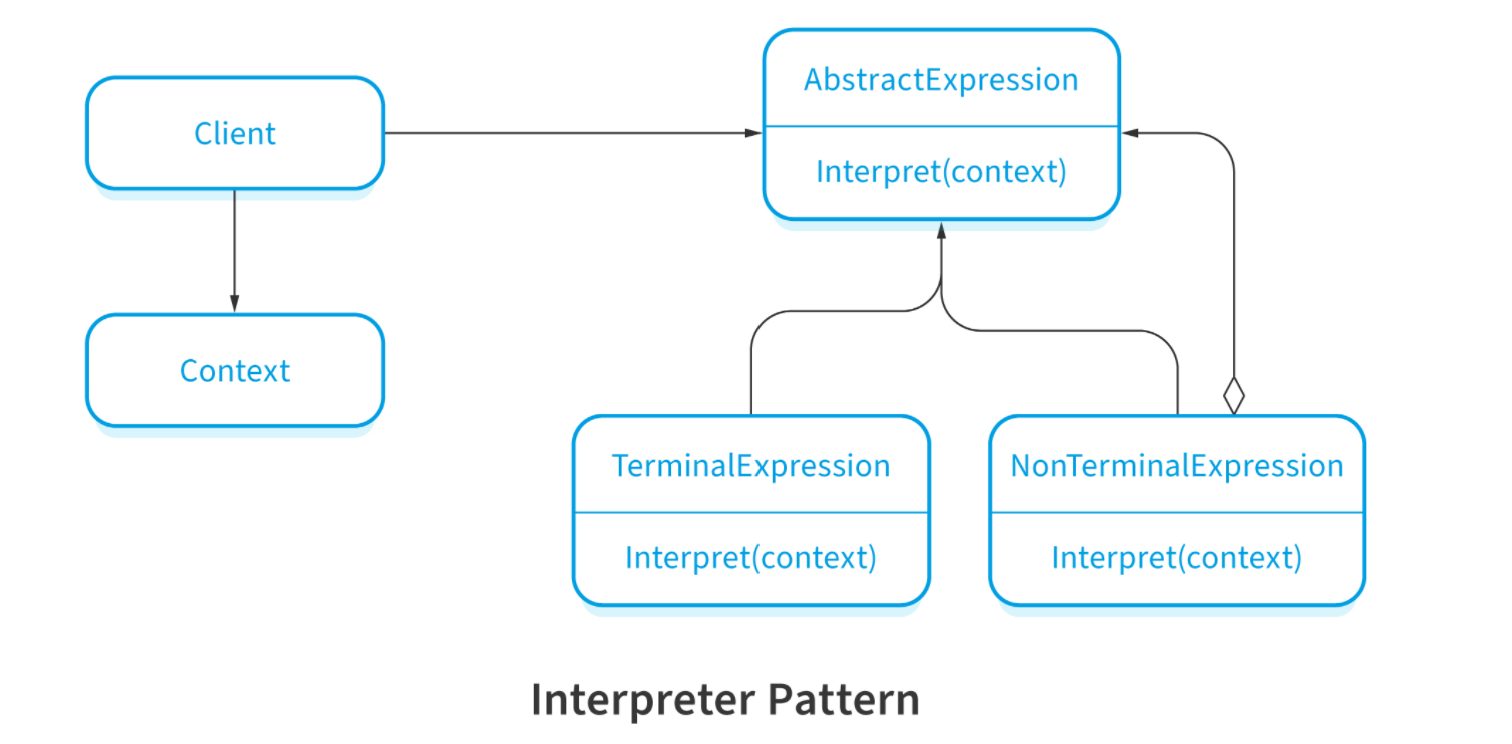
## Usage: Speech recognition, Vehicle identification and tracking, Protein structure identification,Sonar signals interpretation.

Few **advantages**, as follows:

* The pattern facilitates experiments.
* You can reuse knowledge resources like algorithms.

Disadvantages as follows.:

* It’s an intermediate arrangement. Ultimately, you will need to arrive at a suitable architecture pattern, however, you don’t have certainty that you will find the right answer.
* All communication within the system happens via the blackboard, therefore, the application can’t handle parallel processing.
* Testing can be hard

***Interpreter Architecture:***

**Definition :**

This pattern is used for designing a component that interprets programs written in a dedicated language.

It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language.

The basic idea is to have a class for each symbol of the language.

It works as follows:

* You implement an interface that aids in interpreting given contexts in a programming language.
* The pattern uses a hierarchy of expressions.
* It also uses a tree structure, which contains the expressions.
* A parser, external to the pattern, generates the tree structure for evaluating the expressions.

# 4. What makes a good Architecture?

An architecture can only be evaluated in the context of specific goals. A good architecture can be classified into process and structural recommendations.

**Process rules of thumb**

* The architecture should be the product of a single architect or a small team with an identified leader.
* The architect should have the functional requirements and quality attributes prioritized.
* The architecture should be well documented.
* The architecture should be reviewed with the stakeholders.
* The architecture should be evaluated for quality attributes.
* The architecture should lend to incremental implementation (via the creation of a "skeletal" system)
* The architecture should result in a specific set of resource contention areas. The resolution of which is clearly specified, circulated and maintained.

**Structural rules of thumb**

* Well defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concern.
* Each module should have a well-defined interface to allow the development teams to work independently.
* Quality attributes should be achieved using well-known architecture tactics specific to each attribute
* If an architecture depends upon a commercial product, it should be structured such that changing to a different product is inexpensive.
* Creating / Consumption of data should be separated in different modules.
* Parallel-Processing modules: Well defined processes or tasks that do not necessarily mirror the module decomposition
* Task/Process assignment to processor should be changeable
* The architecture should feature a small number of simple interaction patterns

# 5. Why is Software Architecture important?

* SW Architecture can be used as a basis for mutual understanding, negotiation, consensus, and communication among stakeholders.
* Software architecture manifests the earliest design decisions, these decisions are the most difficult to get it correct and the hardest things to change later.
* The Architecture defines constraints on implementation.
* Architectures are both prescriptive and descriptive
* Architecture dictates organizational structure, work breakdown structure,work assignments to teams,plans, schedules, budgets and communication channels among teams.
* Architecture allows to predict system quality attributes without waiting until the system is developed or deployed
* Architecture makes it easier to reason about and manage change.
* The Architecture enables more accurate cost and schedule estimates
* Software architecture can serve as the basis for reuse of
  + - * requirements
      * development-support artifacts (templates, tools, etc.)
      * code / components
      * experience

# 6. Monolithic Architecture

**Definition :**

The **Monolithic** application describes a single-tiered **software** application in which different components are combined into a single program from a single platform.

Components can be:

* Authorization — responsible for authorizing a user
* Presentation — responsible for handling HTTP requests.
* Business logic — the application’s business logic.
* Database layer — data access objects responsible for accessing the database.
* Application integration — integration with other services.
* Notification module — responsible for sending email notifications whenever needed.

**Benefits:**

* Simple to develop
* Simple to test.
* Simple to deploy
* Simple to scale horizontally by running multiple copies behind a load balancer.

**Drawbacks:**

* Maintenance — If the application is too large and complex to understand entirely, it is challenging to make changes fast and correctly.
* The size of the application can slow down the start-up time.
* You must redeploy the entire application on each update.
* Monolithic applications can also be challenging to scale when different modules have conflicting resource requirements.
* Reliability — Bug in any module can potentially bring down the entire process.
* Monolithic applications have difficulty adopting new and advanced technologies. Since changes in languages or frameworks affect an entire application.

# 7. Distributed Architecture

**Definition:**

In distributed architecture, components are presented on different platforms and several components can cooperate with one another over a communication network in order to achieve a specific objective or goal.

* In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.
* A distributed system can be demonstrated by the client-server architecture which forms the base for multi-tier architectures.
* Middleware is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network.
* It sits in the middle of the system and manages or supports the different components of a distributed system.

### Advantages

* Resource sharing − Sharing of hardware and software resources.
* Openness − Flexibility of using hardware and software of different vendors.
* Concurrency − Concurrent processing to enhance performance.
* Scalability − Increased throughput by adding new resources.
* Fault tolerance − The ability to continue in operation after a fault has occurred.

### Disadvantages

* Complexity − They are more complex than centralized systems.
* Security − More susceptible to external attack.
* Manageability − More effort required for system management.
* Unpredictability − Unpredictable responses depending on the system organization and network load

# 8. Synchronous and asynchronous communication

**Synchronous communication**

**Definition :**

In synchronous communication the caller waits until a response is available. It is communication that happens in real time - two or more parties are exchanging information at the same moment with one another.

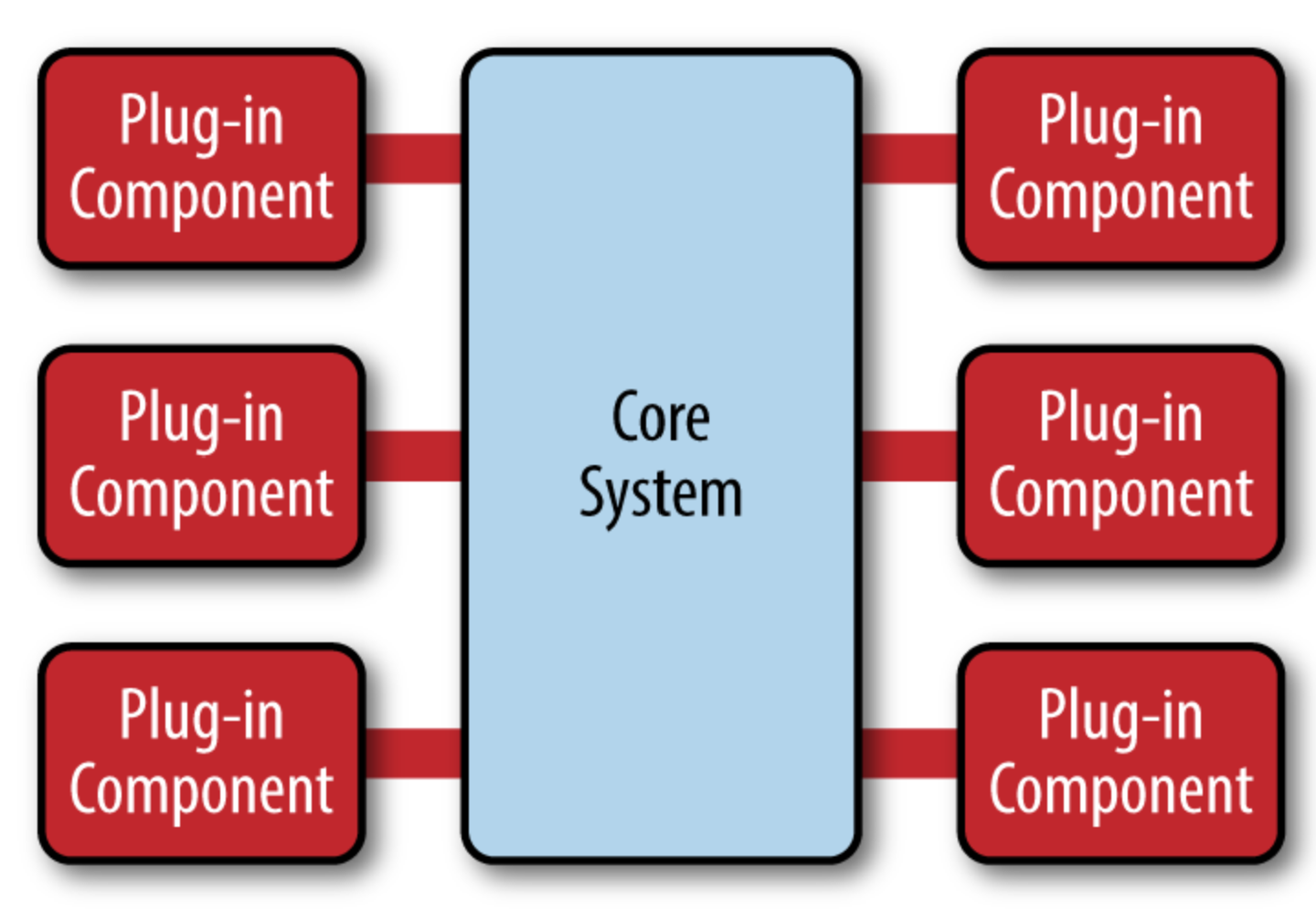
**Asynchronous communication**

**Definition:**

In this communication, the calling service does not wait for a response from the called service.

| **Synchronous** | **Asynchronous** |
| --- | --- |
| Communication in real time | Is well timed |
| Creates interruption in work day | Eliminates interruption |
| Ex : Online chat sessions,live customer support | Ex: Emails, pre-recorded videos |
| The other party is waiting for the replies | Neither expecting or waiting for the information |
| Too much of real time communication leads to burnout and depletes individual efficiency | Saves from unnecessary distractions and gives time to streamline activities on personal end |
| Makes the system unresponsive in case of long running task | Doesn't make the system unresponsive |

# 9. Microkernel Architecture

****

**Definition:**

* The microkernel architecture pattern consists of two types of architecture components: a ***core*** *system* and ***plug-in*** *modules*.
* Application logic is divided between independent plug-in modules and the basic core system, providing extensibility, flexibility, and isolation of application features and custom processing logic
* The core system of the microkernel architecture pattern traditionally contains only the minimal functionality required to make the system operational.
* The plug-in modules are stand-alone, independent components that contain specialized processing, additional features, and custom code that is meant to enhance or extend the core system to produce additional business capabilities.
* The core system needs to know about which plug-in modules are available and how to get to them.
* This registry contains information about each plug-in module, including things like its name, data contract, and remote access protocol details.
* Plug-in modules can be connected to the core system through a variety of ways, including messaging, web services, or even direct point-to-point binding.

# 10. Serverless Architecture

**Definition:**

Serverless architecture describes a way for companies to build and run applications but not have to manage infrastructure.

It provides a way to remove architecture responsibilities from your workload, including provisioning, scaling, and maintenance.

Scaling can be automatic, and you only pay for what you use.

**Advantages:**

* The leading advantage is that your developers can focus their attention on product development. They no longer have to account for managing and operating servers.

### Serverless is event-based

Serverless uses an event-based system versus stream-based.

With event-based architecture, each subpart of the application is independent.

If there is a failure, it just impacts that event, not the entire log.

### Faster deployments, greater flexibility, and accelerated innovation

You can rapidly deploy apps in hours because there’s no infrastructure construction to weigh you down.

With faster deployments also comes ease in scalability.

.

### Reducing architecture costs

You are no longer responsible for the huge investments required for internal architecture administration.

**Limitations:**

### Long-running application inefficiencies

Running workloads, which are long-running, could be more costly on serverless. Using a dedicated server is often more efficient.

### Third party dependency

Serverless architecture requires you to be reliant on your provider.

You don’t have full control, and changes may impact you without notice.

### Cold starts

A “cold start” occurs when a platform must initiate internal resources.

It may take some time for your serverless architecture to handle that first function request.

# 11. Microservices Architecture

**Definition:**

Microservices architecture refers to an architectural style for developing applications.

A microservices architecture is a type of application architecture where the application is developed as a collection of services.

It provides the framework to develop, deploy, and maintain microservices architecture diagrams and services independently.

Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility. To serve a single user request, a microservices-based application can call on many internal microservices to compose its response.

**Advantages:**

* Microservices are self-contained, independent deployment modules.
* The cost of scaling is comparatively less than the monolithic architecture.
* Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
* It is possible to change or upgrade each service individually rather than upgrading in the entire application.
* Microservices allows us to develop an application which is organic in nature.
* It enables event streaming technology to enable easy integration in comparison to heavyweight interposed communication.
* Microservices follow the single responsibility principle.
* The demanding service can be deployed on multiple servers to enhance performance.
* Less dependency and easy to test.
* Dynamic scaling.
* Faster release cycle.

**Disadvantages:**

* Microservices have all the associated complexities of the distributed system.
* There is a higher chance of failure during communication between different services.
* Difficult to manage a large number of services.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

# 12. Monolithic Vs Microservices

|  | Monolithic | Microservice |
| --- | --- | --- |
| Architecture | Built as a single logical executable | Built as a suite of services, each running separately and communicating with lightweight mechanisms |
| Modularity | Based on language features | Based on business capabilities |
| Agility | Changes to the system involves building and deploying a new version of the entire application | Changes can be applied to each service independently |
| Scaling | Entire application scaled horizontally behind a load balancer | Each service scale independently whenever needed |
| Implementation | Typically written in one language | Each service is implemented in the language that best fits the need |
| Maintainability | Large code base intimidating to the new developer | Smaller code base is easier to manage and understand |
| Transaction | ACID | Base |

# 13.C4 Model

### Person

A person represents one of the human users of your software system (e.g. actors, roles, personas, etc).

### Container

### In the C4 model, a container represents an application or a data store. A container is something that needs to be running in order for the overall software system to work. In real terms, a container is something like:

* **Server-side web application**: A Java EE web application running on Apache Tomcat, an ASP.NET MVC application running on Microsoft IIS, a Ruby on Rails application running on WEBrick, a Node.js application, etc.
* **Client-side web application**: A JavaScript application running in a web browser using Angular, Backbone.JS, jQuery, etc.
* **Client-side desktop application**: A Windows desktop application written using WPF, an OS X desktop application written using Objective-C, a cross-platform desktop application written using JavaFX, etc.

### Component

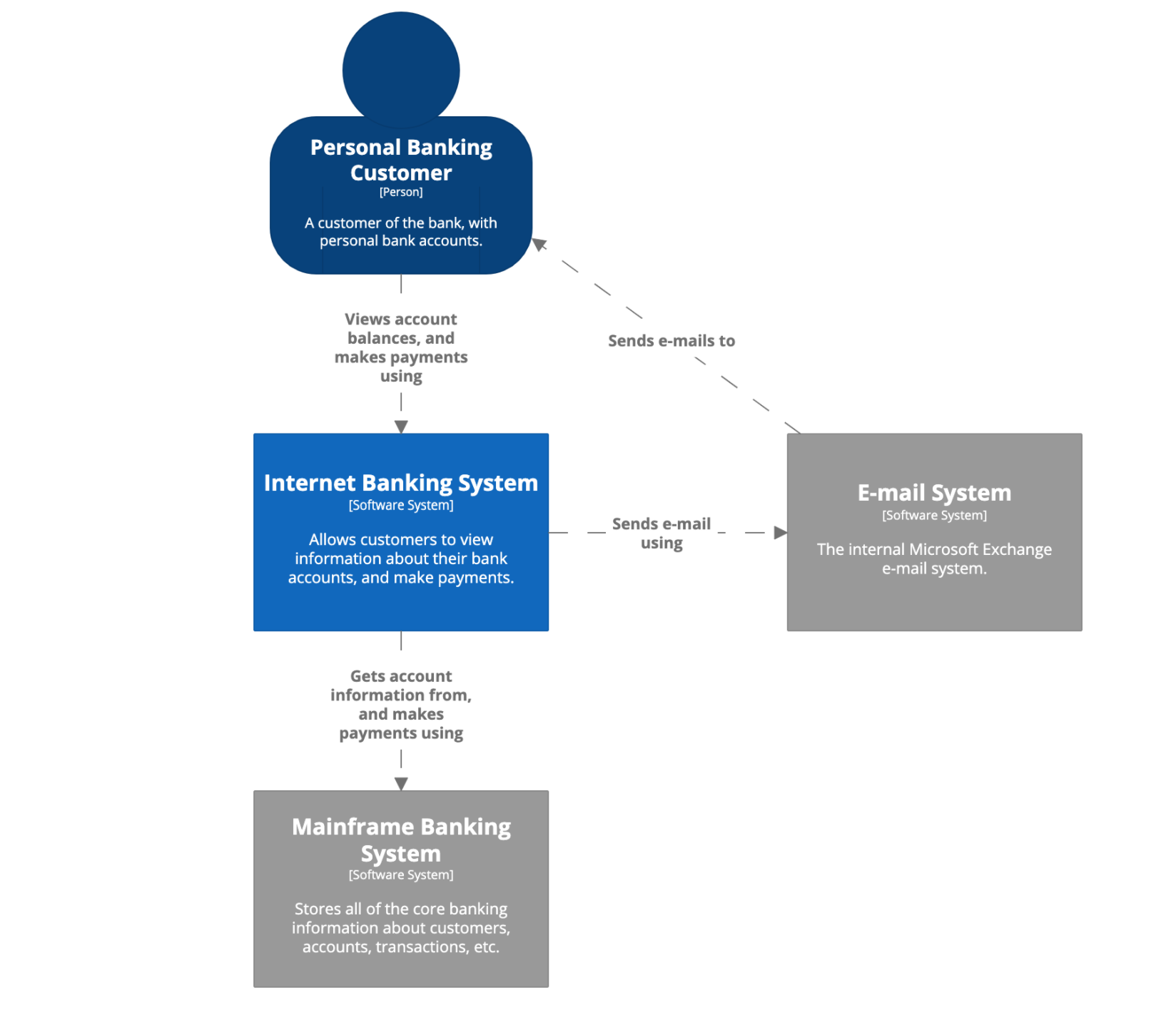
A component is a grouping of related functionality encapsulated behind a well-defined interface.

**Code**

In the C4 model, code-level diagrams are similar to class diagrams or entity-relation diagrams, which represent very specific implementations.

**Level 1 : System Context Diagram**

A system context diagram shows the software system you are building and how it fits into the world in terms of the people who use it and the other software systems it interacts with.



Personal customers of the bank use the Internet banking system to view information about their bank accounts and to make payments.

The Internet banking system uses the bank's existing mainframe banking system to do this, and uses the bank's existing e-mail system to send email to customers.

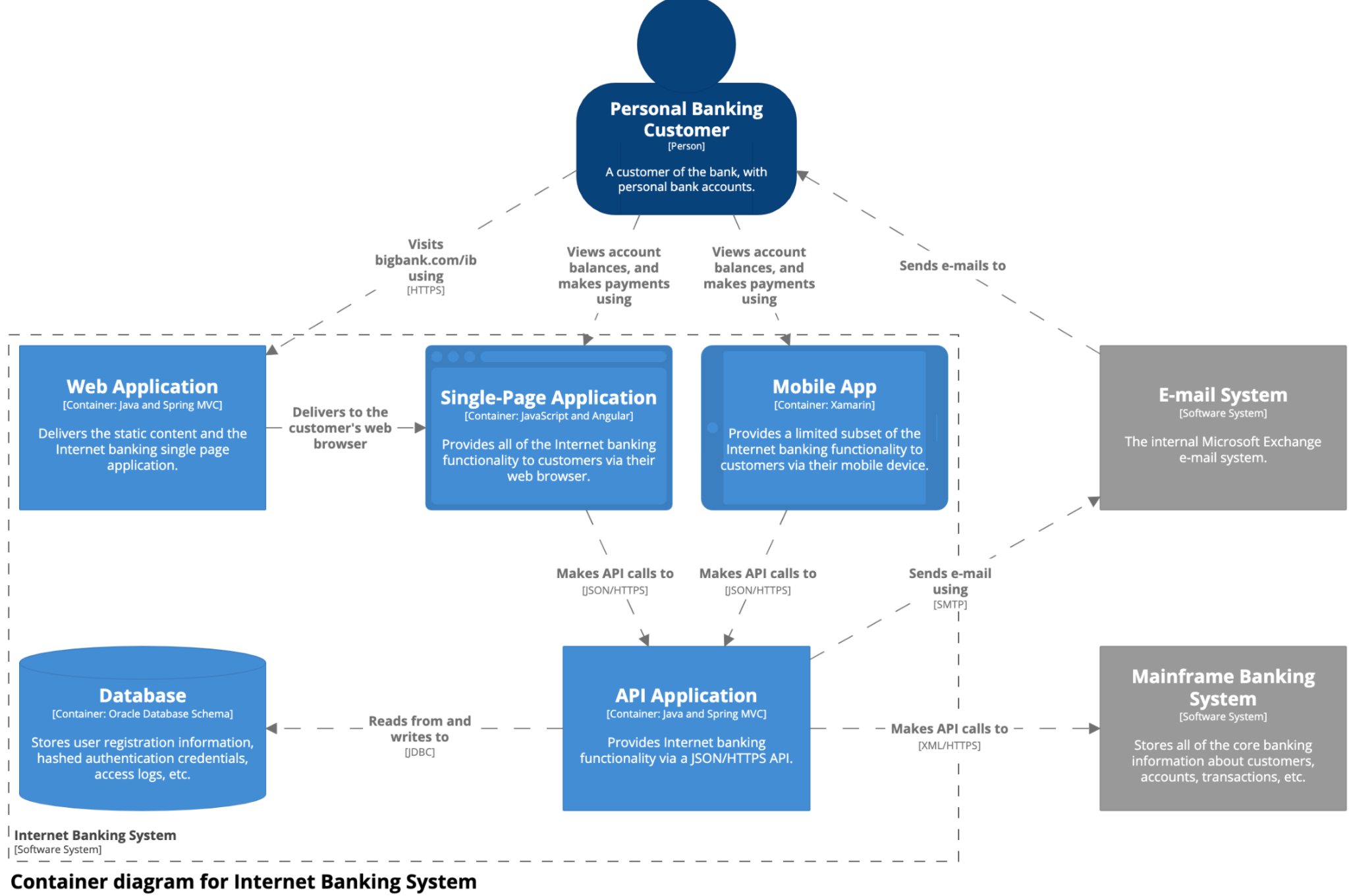
Colour coding in the diagram indicates which software systems already exist (the grey boxes) and those to be built (blue).

## Level 2: Container diagram

A container diagram zooms into the software system, and shows the containers (applications, data stores, microservices, etc.) that make up that software system.

Technology decisions are also a key part of this diagram.

Below is a sample container diagram for the Internet banking system. It shows that the Internet banking system (the dashed box) is made up of five containers: a server-side web application, a client-side single-page application, a mobile app, a server-side API application, and a database.



The web application is a Java/Spring MVC web application that simply serves static content (HTML, CSS, and JavaScript), including the content that makes up the single-page application.

The single-page application is an Angular application that runs in the customer's web browser, providing all of the Internet banking features.

Alternatively, customers can use the cross-platform Xamarin mobile app to access a subset of the Internet banking functionality.

Both the single-page application and mobile app use a JSON/HTTPS API, which another Java/Spring MVC application running on the server side provides.

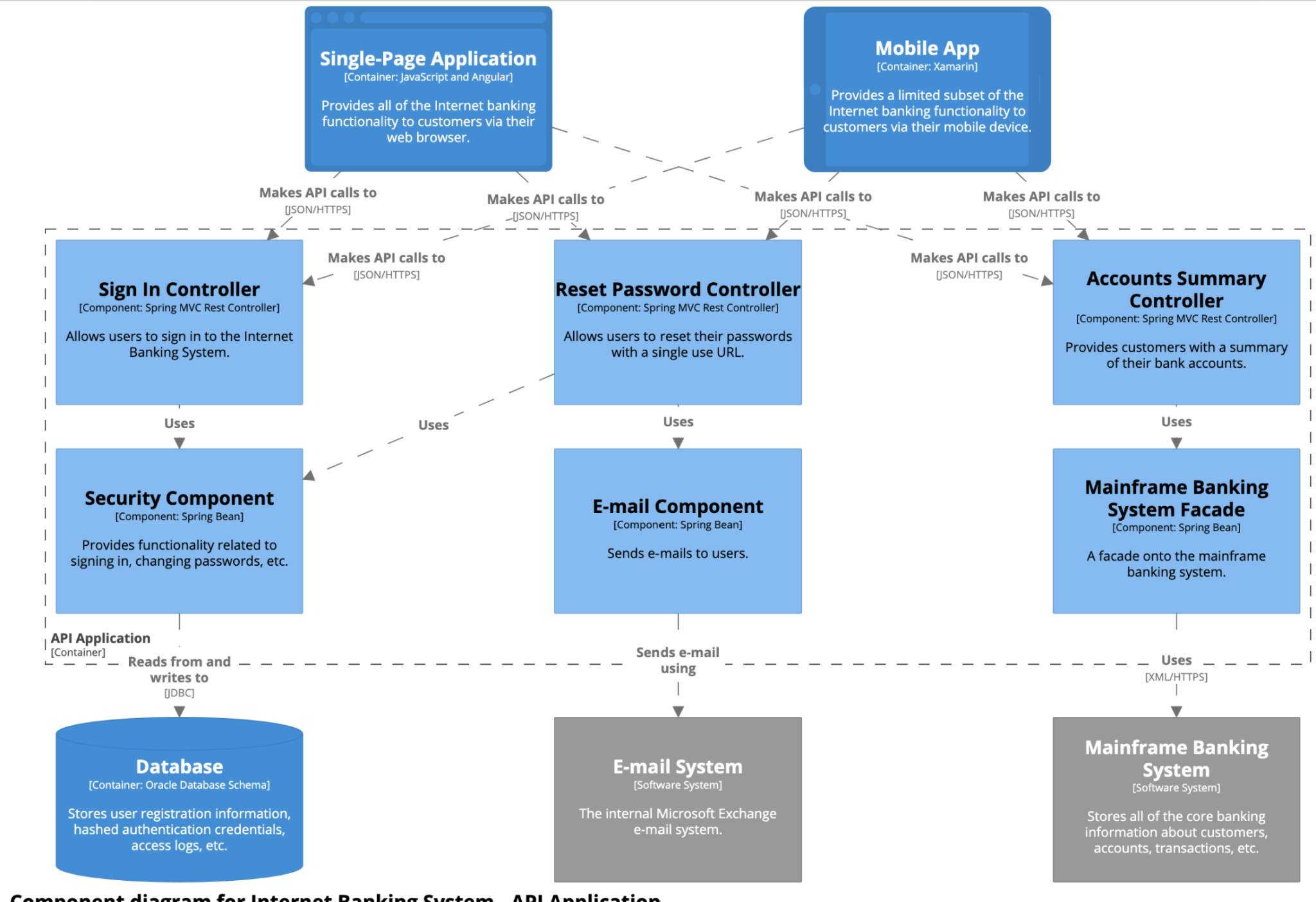
The API application gets user information from the database (a relational-database schema).

The API application also communicates with the existing mainframe banking system, using a proprietary XML/HTTPS interface, to get information about bank accounts or make transactions. The API application also uses the existing e-mail system if it needs to send e-mail to customers.

## Level 3: Component diagram

A component diagram, zooms into an individual container to show the components inside it.

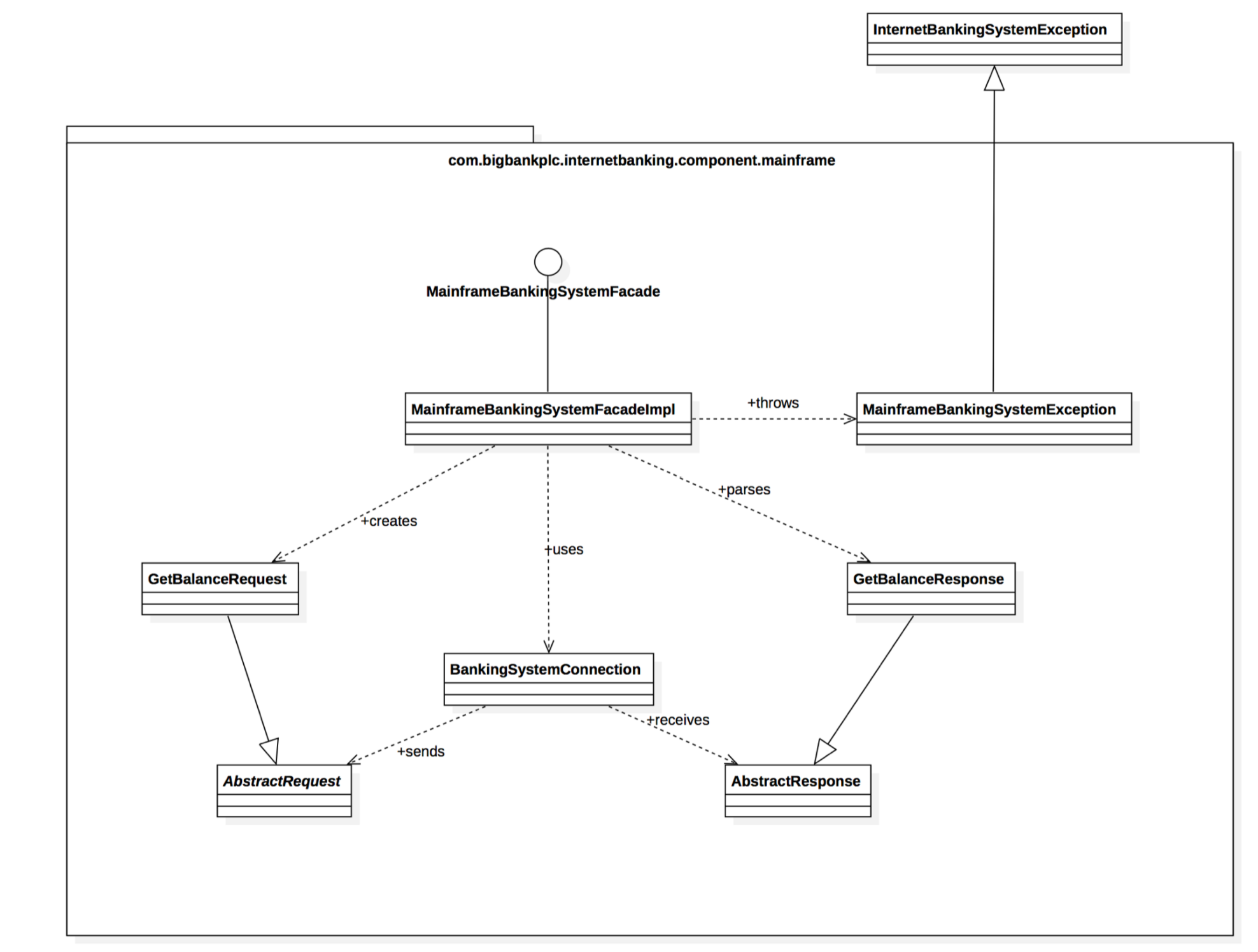
These components should map to real abstractions in your codebase.



Two Spring MVC Rest Controllers provide access points for the JSON/HTTPS API, with each controller subsequently using other components to access data from the database and mainframe banking system.

**Level 4: Code**

Finally, if you really want or need to, you can zoom into an individual component to show how that component is implemented. This is a sample (and partial) UML class diagram for the fictional Internet banking system, showing the code elements (interfaces and classes) that make up the MainframeBankingSystemFacade component.



It shows that the component is made up of a number of classes, with the implementation details directly reflecting the code.