5 essential patterns of software architecture

https://www.redhat.com/en/blog/5-essential-patterns-software-architecture#microservices

As you browse redhat.com, we'll recommend resources you may like. For now, try these.

Share

The world is becoming increasingly dependent on software for almost every human activity. From mobile applications we use to connect with others to healthcare applications and deep learning models, from finance technology systems to smart buildings that leverage tech to automate many activities, software systems have permeated and simplified many aspects of human life. For these software systems to provide our desired solutions, they must be built on the right architecture to produce optimal results.

[Check out Red Hat's Portfolio Architecture Center for a wide variety of reference architectures you can use.]

Just like the architecture of a building, software architecture describes the design and collection of components into systems that make up the building blocks of software. Software architecture explains the structural composition of the software program and the interactions between the elements. The principle that defines the software organization schema for these software systems is called an architectural pattern.

The architectural pattern captures the design structures of various systems and elements of software so that they can be reused. During the process of writing software code, developers encounter similar problems multiple times within a project, within the company, and within their careers. One way to address this is to create design patterns that give engineers a reusable way to solve these problems, allowing software engineers to achieve the same output structurally for a given project.

From an engineer's perspective, software architecture patterns are important because they drive efficiency and productivity. Developers can join an existing project at any point with limited

onboarding since they already understand the architecture pattern used in the project. New features can also be added to the project without any difficulty, and common application problems can be solved easily.

From the client's perspective, architecture patterns optimize development costs, speed up the project's timeline, and allow the engineer to deliver a high-quality product. Cost estimates are based on an understanding of the architecture systems, so product managers have a more accurate project cost, allowing for early planning and budgeting. Further, a clearly defined architecture means the system has been validated and scoped. This helps engineers focus on the essentials of the product and also allows managers to plan adequately for project completion.

[Learn more about validated patterns.]

There are many different types of software architecture patterns, and this article explores five of them and how they are integral to software development.

The model-view-controller (MVC) pattern divides an application into three components: A model, a view, and a controller.

The model, which is the central component of the pattern, contains the application data and core functionality. It is the dynamic data structure of the software application, and it controls the data and logic of the application. However, it does not contain the logic that describes how the data is presented to a user.

The view displays application data and interacts with the user. It can access data in the model but cannot understand the data, nor does it understand how the data can be manipulated.

The controller handles the input from the user and mediates between the model and the view. It listens to external inputs from the view or from a user and creates appropriate outputs. The controller interacts with the model by calling a method on it to generate appropriate responses.

These three components interact via some form of notification, such as an event or a callback. These notifications contain state information, such as state changes, which are communicated to update these components. For instance, an external event from the user may be transmitted to the controller to update the view. The MVC pattern, therefore, decouples software components and

allows the codes to be reused easily.

Major programming languages such as JavaScript, Python, Java, and Swift have MVC frameworks to develop web and mobile applications. Web application frameworks such as Ruby on Rails and Django (for Python) are based on this architecture.

One advantage of the MVC pattern is that multiple engineers can work on all three components simultaneously without conflict. Further, the MVC allows logical grouping of related outputs to generate numerous views from the model. However, one drawback is that navigating the framework could be complex as it introduces several layers of abstraction.

Useful references:

[Download an architect's guide to multicloud infrastructure.]

The microservices pattern involves creating multiple applications?or microservices?that can work interdependently. Although each microservice can be developed and deployed independently, its functionality is interwoven with other microservices.

A key concept in the microservices pattern is the separate deployment of units. This creates a streamlined delivery pipeline that allows for easy deployment of microservices and increases application scalability. Another key feature of this pattern is that it is a distributed architecture, meaning that the structure's components can be fully decoupled and accessed through remote access protocols such as REST, SOAP, or GraphQL. This distributed nature of the pattern allows for its high scalability properties.

[Looking for information on system automation? Get started with The Automated Enterprise, a free book from Red Hat.]

The microservices architecture uses several design patterns: Aggregator pattern, API gateway design pattern, chain of responsibility pattern, branch pattern, and asynchronous messaging design pattern. Each approach provides a method to manipulate data to produce services.

Oreilly.com

A major advantage of microservices architecture is the independent deployment of each microservice. Engineers can write and maintain each microservice independent of the others,

potentially increasing its functionality and scalability. Further, because each microservice is small, it is easier to rewrite and update. Microservices architecture is best for web applications and websites with small components. It is also useful for corporate datacenters that have well-defined boundaries. Some challenges for microservices come up around complexity, particularly in the network layer. Furthermore, decoupling services to work completely independent of each other requires significant architectural expertise.

Useful references:

In the client-server architecture patterns, there are two main components: The client, which is the service requester, and the server, which is the service provider. Although both client and server may be located within the same system, they often communicate over a network on separate hardware.

The client component initiates certain interactions with the server to generate the services needed. While the client components have ports that describe the needed services, the servers have ports that describe the services they provide. Both components are linked by request/reply connectors. A classic example of this architecture pattern is the World Wide Web. The client-server pattern is also used for online applications such as file sharing and email.

Topcoder.com

A simple example is online banking services. When a bank customer accesses online banking services using a web browser, the client initiates a request to the bank's web server. In this case, the web browser is the client, accessing the bank's web server for data using the customer's login details. The application server interprets this data using the bank's business logic and then provides the appropriate output to the web server.

One major advantage of this architecture pattern is the central computing of data; all files are stored in a central location for this network. Therefore, the data, as well as the network peripherals, are centrally controlled. A disadvantage, however, is that the server is expensive to purchase and manage.

The client-server model is related to the peer-to-peer architecture pattern and is often described as a subcategory of this pattern. The latter uses a decentralized system in which peers communicate

with each other directly.

Useful links:

This was often referred to as the master/slave architecture pattern, but because it is not a useful metaphor, some engineers and software companies have adopted replacement terms such as primary/secondary, primary/replica, parent/helper, master/replica or the controller/responder pattern. Most notably, the IEEE has adopted this as a better term for network technology.

Like the client-server architecture pattern, this pattern consists of two components: The controller and the responders. The controller component distributes the input or work among identical responder components and generates a composite result from the results generated from each responder.

In simple terms, the controller is the actual data keeper while the responder replicates data stored in the controller database. Writing data is done only in the controller database, which is read by the responder database. The controller determines the communication priorities of the responder and exerts control over them. This is unlike the peer-to-peer architecture pattern, in which computers communicate as equals and share responsibilities.

[Free cheat sheet: IT job interview tips.]

An example of the controller-responder model includes duplication done using cassette tape or compact disc recorders.

A key advantage of this pattern is that analytic applications can be read from the responder component without changing the data content of the controller component. Also, the responders can be taken offline and synced back without any time loss. However, when a controller fails, all of the data could be lost, and the application may have to be restarted. In these situations, a responder may be promoted to controller, but not without some data and technical deficits.

Useful links:

The layered architecture pattern is the most common among developers. It is useful for programs that comprise several groups of subtasks, each of which is at a different level of abstraction. Each of these subtasks is represented by a layer in the software?a unit of modules that produces a cohesive

set of services?and each layer provides services to the next higher layer in a unidirectional pattern.

Each layer has a specific role within the application that is connected to the roles of other layers. For instance, a presentation layer, also called the UL layer, would handle all the UI and browser communication logic while a business logic layer would execute certain business requests.

[Discover ways enterprise architects can map and implement modern IT strategy with a hybrid cloud strategy.]

Other types of layers include the application layer and the data access or persistence layer, which then accesses the database layer. This allows the database layer to be separated and enables you to switch from an Oracle server to a SQL server without impacting the other layers?it allows for lower transition costs.

These layers interact in a unidirectional pattern, such that when a user initiates an input, such as clicking a button, the presentation layer sends messages to the lower layer, the application layer, which, in turn, calls the business layer, then the data access layer that accesses the database. So calls in a layered pattern flow downwards, from a higher layer to a lower one.

dzone.com

Layered architecture patterns are found in many e-commerce web applications and desktop applications. It is also useful for applications that need to be built quickly and for enterprise applications that need to adopt traditional IT processes. Furthermore, a layered pattern is ideal for applications that require strict standards of testability.

A key advantage of this pattern is that it allows for an easy way to write a well-organized application. Since it is a popular architecture pattern, developers already have an understanding of how it is used. However, it does have two major drawbacks: Complexity and the cost of adding more layers. These layers may eventually be hard to split up.

Useful references:

Several other architecture patterns, including pipe-filter pattern, blackboard pattern, broker pattern, and event-bus pattern, are also useful in different aspects of software developments. The concept is the same for all: Defining the basic characteristics of your application, enhancing the functionality of

the product, and enhancing efficiency and productivity of the app-building process.

However, using the wrong architecture pattern could delay your project and may even lead to software failure. Therefore, the key is to have a good understanding of architecture patterns and which applications they are most suitable for so that you can choose the one that fits your software requirements.

Anand Butani is a software engineer and product manager at Top Flight Apps (TFA). With over 10 years of experience, he helps scope and build the infrastructure of healthcare web and mobile applications at TFA. He helps companies find the right way to apply machine learning to their applications to yield value and increase the bottom line. After having worked with clients such as Twitter, Clarifai, Hilton, Pfizer, GSK, Johnson & Johnson, Ironwood Pharmaceuticals, and Medpace, he understands the way to communicate technical knowledge effectively to various stakeholders and departments. You can find him sharing his knowledge here at the Red Hat Enable Architect blog, Top Flight Apps Ideas blog, or LinkedIn.

The latest on IT automation for tech, teams, and environments

Updates on the platforms that free customers to run AI workloads anywhere

Explore how we build a more flexible future with hybrid cloud

The latest on how we reduce risks across environments and technologies

Updates on the platforms that simplify operations at the edge

The latest on the world?s leading enterprise Linux platform

Inside our solutions to the toughest application challenges

Entertaining stories from the makers and leaders in enterprise tech

We?re the world?s leading provider of enterprise open source solutions?including Linux, cloud, container, and Kubernetes. We deliver hardened solutions that make it easier for enterprises to work across platforms and environments, from the core datacenter to the network edge.