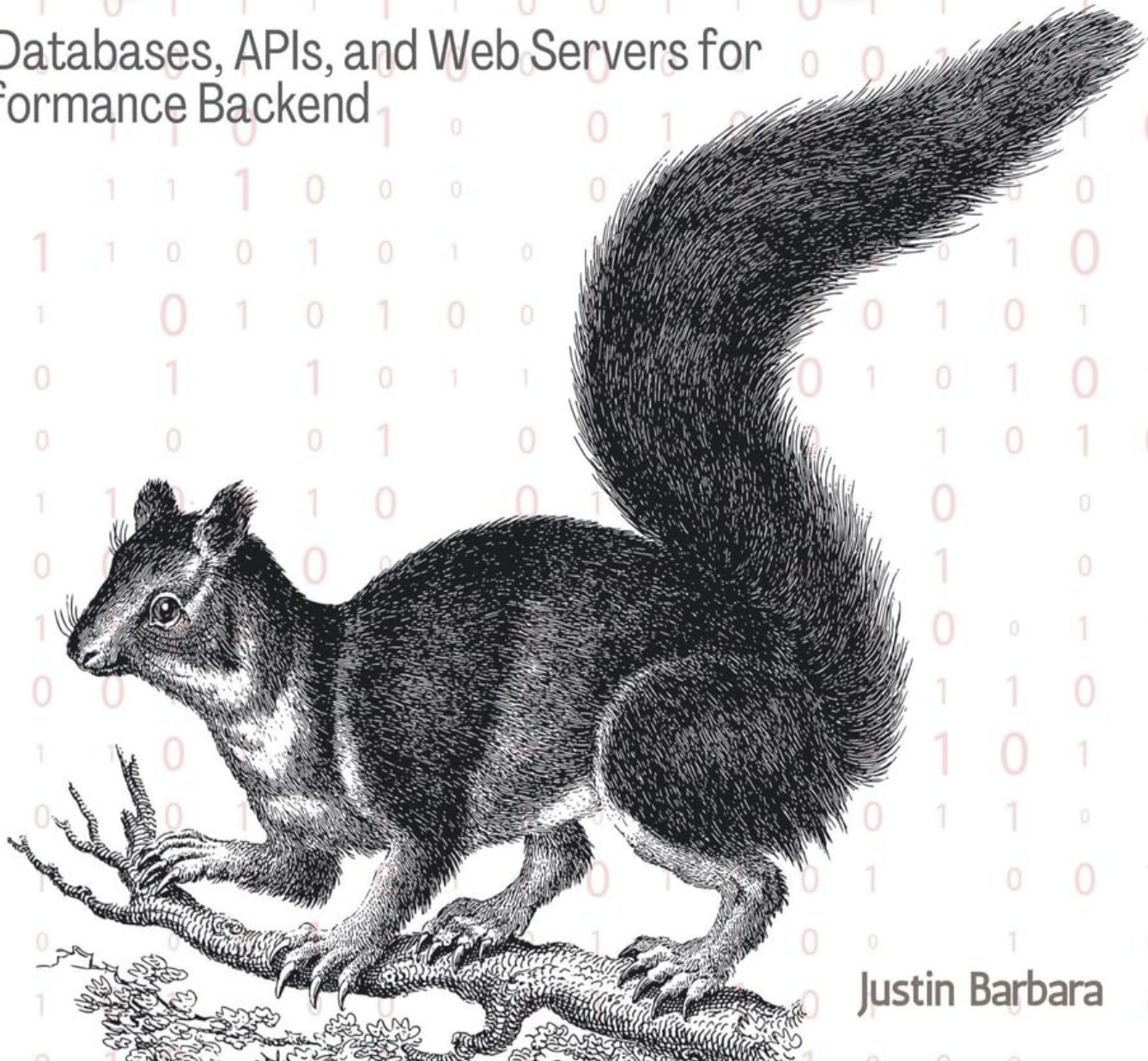
GitforGits™
ASIAN PUBLISHING HOUSE

Practical C++ Backend Programming

Crafting Databases, APIs, and Web Servers for High-Performance Backend



Justin Barbara

Preface

"Practical C++ Backend Programming" is a comprehensive walkthrough that provides readers with the necessary tools and knowledge to become proficient C++ backend developers. With a strong focus on real-world application and practical implementation, this book takes readers on a journey through the multifaceted landscape of backend development, making it an essential resource for any aspiring or current backend developer.

Starting with the basics, the book introduces C++, providing a solid foundation in the language, its structure, and core conceptswith regards to backend programming. From there, readers dive into the more complex elements of backend development. Through our engaging sections, use-cases and sample examples, readers are introduced to advanced topics such as concurrent programming, exploring threading and multiprocessing to handle intensive computational tasks, thus laying the groundwork for scalable applications.

This book offers an in-depth look into APIs, specifically gRPC, along with caching strategies, and database management, using MongoDB as a NoSQL database of choice. All the while, readers will learn to implement these technologies in a practical context, building a blog application from scratch, thereby bridging the gap between theory and practical implementation. An entire section is dedicated to securing applications, wherein the book teaches authentication, authorization, and database security, and demonstrates how to implement these measures in the blog application.

Another utmost important part of this book is to cover testing strategies, teaching the reader how to employ Google Test (gtest) to create robust and fail-proof backend solutions. Finally, the journey culminates in a step-by-step walkthrough to deploying applications on AWS, ensuring the reader is equipped with the necessary skills to take their applications live.

In this book you will learn how to:

- Detailed overview of C++ programming, catering to both beginners and experienced coders.
- Practical exploration of concurrent programming for scalable and efficient application design.
- Comprehensive understanding of API usage, specifically using the gRPC framework.
- Deep dive into MongoDB for effective NoSQL database management and operations.
- Thorough walkthrough to implementing caching strategies for performance optimization.
- Strategic use of Nginx for handling web server needs including load balancing.
- Hands-on guide to implementing security measures for databases, APIs, and web servers.
- Instruction on employing Google Test for robust application testing and debugging.
- Step-by-step guidance for deploying applications on AWS, ensuring real-world readiness.
- Practical application of concepts via building and refining a blog application.

Prologue

Today's software engineers need a diverse set of abilities to keep up with the fast-paced evolution of technology. Knowledge of databases, application programming interfaces (APIs), web servers, and other related technologies is also required. Using the robust and flexible C++ programming language, this book is meant to serve as a comprehensive guide to backend development. This book teaches readers the fundamentals of C++ application development through a combination of theoretical discussion, real-world examples, and hands-on exercises.

The first part of the book is devoted to explaining C++ and why so many programmers find it to be the best language for the job. This book provides a comprehensive overview of C++, covering everything from the language's fundamentals to its cutting-edge features like concurrency. C++ Standard Template Library (STL) is a robust toolkit for C++ programmers, and it is explored in detail. The second half of the book delves into algorithms and databases, the meat and potatoes of back-end development. It explains how various algorithms can be used to simplify difficult problems, enhance functionality, and streamline the code. The book delves into how to put the CRUD operations into practice using MongoDB, a well-known NoSQL database. Third, the book dives into the ins and outs of building and utilizing APIs, with a focus on the gRPC framework. The gRPC framework is a trustworthy and productive tool for creating distributed programs and services. It helps programmers create APIs that can scale to a large number of requests without sacrificing performance. This section also delves into the topic of protecting APIs and web servers.

The testing and debugging process takes up the bulk of the book's fourth section. Using the Google Test framework, it explains in detail how to create test cases and how to troubleshoot problems that may arise. The code is always in a deployable state thanks to the book's explanation of how to set up a continuous testing environment. The deployment procedure is discussed in the book's final chapter. It explains in detail how to use AWS, the most widely used cloud platform, for application deployment. This section also discusses the various factors, such as security, scalability, and performance, that must be taken into account during deployment.

Throughout the book, you'll build a blog application to practice the skills you learn. This method guarantees that readers aren't just absorbing the theory but also using it in practice. What sets this book apart from others is its emphasis on actual application. By the book's conclusion, readers will have a solid grounding in C++ back-end development and will be able to apply what they've learned to create applications that are efficient, robust, and secure.

PRACTICAL C++ BACKEND PROGRAMMING

Crafting Databases, APIs, and Web Servers for High-Performance Backend

Justin Barbara

Published by: GitforGits
Publisher: Sonal Dhandre
www.gitforgits.com
support@gitforgits.com

Printed in India

First Printing: July 2023

ISBN: 978-8196288389

Cover Design by: Kitten Publishing

Content

- Preface
- Prologue

Chapter 1: Introduction to Backend Development

- **Evolution of Web Ecosystem**
  - The Static Web (Web 1.0)
  - The Dynamic Web (Web 2.0)
  - The Social Web
  - The Mobile Web
  - The Programmable Web
- **The Server Side: A Deeper Dive**
  - Server Architecture
  - Backend Languages and Frameworks
  - Databases and Data Management
  - Middleware and Business Logic
  - APIs and Microservices
  - Security and Optimization
- **Exploring Backend Development**
  - High-Performance Systems
  - Systems Programming
  - Database Systems
  - Microservices Architecture
  - APIs and gRPC
  - Backend Security
  - Data Processing and Machine Learning
- **C++ Capabilities Overview**
  - C++20 Features
  - C++23 Features
- **Setting up C++ Backend Environment**
  - Installing a C++ Compiler
  - Installing an IDE/Text Editor
  - Installing Essential C++ Libraries
  - Verify the Setup
- **Summary**

Chapter 2: C++ Refresher and Essentials

- **Potential of C++ for Backend Programming**
- **C++ Syntax**
  - Variables and Data Types
  - Control Structures
  - Functions
  - Classes and Objects
  - STL Containers
  - Error Handling
- **C++ Semantics**
  - Value Semantics
  - Reference Semantics
  - Object-Oriented Programming (OOP)
  - Resource Acquisition is Initialization (RAII)
  - Templates and Generic Programming
  - Concurrency
  - Exception Handling
  - Memory Management
  - Move Semantics
  - Lambda Expressions
  - Smart Pointers
  - Type Inference
  - Standard Template Library (STL)
- **C++ Data Structures**
  - Arrays
  - Strings
  - Lists (std::list)
  - Vectors (std::vector)
- **OOP in C++**
  - Classes and Objects
  - Inheritance
  - Polymorphism
- **Standard Template Library**
  - Containers
  - Algorithms
  - Function Objects (Functors)

Error Handling
Throwing Exceptions
Catching Exceptions
Custom Exception Types
Multithreading
Create New Thread
Data Sharing and Synchronization
Application in Backend Development
Summary
Chapter 3: Deep Dive into Algorithms
Algorithms Overview
Algorithm Design
Problem Definition
Identifying Inputs and Outputs
Defining the Process
Algorithm Pseudocode
Optimization
Testing
Sorting Algorithms
Bubble Sort
Insertion Sort
Quick Sort
Merge Sort
Searching Algorithms
Linear Search
Binary Search
Graph Algorithms
Depth-First Search (DFS)
Breadth-First Search (BFS)
Dijkstra's Algorithm
Hash Algorithms
Simple Hash Function
Collision Resolution
Recursive Algorithms
Overview
Sample Program: Fibonacci Sequence

Iterative Algorithms
Overview
Sample Program: Fibonacci Sequence (Iterative)
Dynamic Programming Algorithms
Sample Program: Fibonacci Sequence (Dynamic Programming)
Sample Program: Coin Change Problem
Summary
Chapter 4: Mastering Version Control - Git and GitHub
Version Control and Repo Hosting: Overview
Exploring Git
Install Git
Configure Git
Initialize Git Repository
Tracking Changes
Checking Status
Basic Git Commands and Workflow
git clone
git status
git add
git commit
git push
git pull
git branch
git checkout
git merge
Branching and Merging
Branching
Merging
Setting up GitHub
Sign up and Create New Repository
Link your Local Repository to GitHub
Push your Local Code to GitHub
Collaborate
Fetch and Merge Changes
Repositories, Forking and Pull Requests
Repositories

Forking
Pull Requests
Using Git for Tagging, Stashing, and Reverting Changes
Tagging
Stashing
Reverting Changes
Summary
Chapter 5: Managing Database Operations with MongoDB
Role of Database in Backend Development
Explore MongoDB
Install and Configure MongoDB
Install MongoDB
Install MongoDB C++ Driver
Sample Program to Connect MongoDB
Database Schema for Full Stack Application
Performing CRUD Operations
Create
Read
Update
Delete
Performing Complex Queries: Aggregation and Indexing
Executing Aggregation
Creating Indexes
Summary
Chapter 6: Crafting REST APIs with gRPC
API Dynamics in Backend
Introduction to gRPC API Framework
Features of gRPC
Setting up gRPC for C++
Implement gRPC
Define the Service
Generate the Service Code
Implement the Service
Building CRUD API
Update .proto File
Generate Service Code

- Implement the Service
- **Troubleshooting gRPC for API Implementation**
  - Compilation Errors with .proto Files
  - Communication Errors between Client and Server
  - Errors related to Service Definitions
  - Errors in gRPC Calls
  - Performance Issues
- **Summary**

Chapter 7: Dealing with Client-Side and Server-Side Caching

- **Caching for C++ Backend Development**
- **Server-side Caching Strategies**
  - In-Memory Caching
  - Database Caching
  - Content Delivery Network (CDN) Caching
  - HTTP Caching
- **Process of Database Caching**
  - Query Execution
  - Query Result Storage
  - Cache Retrieval
  - Cache Invalidation
- **Implementing Database Caching**
- **Advanced Cache Strategies**
  - Caching Individual Blog Posts
  - Cache aside / Lazy Loading
  - Write-through Cache
- **Using gRPC for Cache Performance**
  - Define Service
  - Data Fetch
  - Establishing gRPC Server
- **Cache Eviction Strategies**
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)
  - Random Replacement (RR)
- **LRU Implementation**
- **LFU Implementation**
- **RR Implementation**

Troubleshooting Caching Errors
Cache Invalidation
Cache Stampede
Nodes Inconsistency
Summary
Chapter 8: Managing Web Servers with Nginx
Web Servers Overview
Exploring Nginx
Features
Components
Working Mechanism of Nginx
Install and Configure Nginx for C++
Update Your System
Install Nginx
Start Nginx
Adjust Firewall
Verify Installation
Configuring Nginx
Integrating Nginx with C++ Backend
Reverse Proxy
Benefits of Reverse Proxy
Setting up Reverse Proxy using Nginx
Handle HTTPS Traffic
Managing HTTPS Traffic
Handling High Traffic
Load Balancers
Overview
Setting up Load Balancers
SSL Configuration
Using Nginx to Configure SSL
Managing Static and Dynamic Assets
Serving Static Content
Serving Dynamic Content
Summary
Chapter 9: Testing Your C++ Backend
Why Testing Matters?

C++ Testing Frameworks: Doctest and Google Test
Doctest
Google Test (gtest)
Install Google Test
Clone the Google Test Repository
Build Google Test Libraries
Install Google Test
Link Google Test to your Project
Perform Unit Testing for Data Loss
Include Necessary Libraries & Project Header Files
Write the Test
Running the Test
Perform Integration Testing for MongoDB and gRPC
Set up Test Environment
Implement Tests
Run Tests
Continuous Testing and Test Automation
Sample Program to Setup Continuous Testing
Troubleshooting and Solutions
Google Test Compilation Issues
Tests Failing Unexpectedly
Flaky Tests
Troublesome with Setting up Continuous Testing
GitHub Actions unable to Locate Files/Directories
Trouble with GitHub Actions Secrets
Summary
Chapter 10: Securing Your C++ Backend
Backend Security Overview
Database Security
Secure MongoDB Database
Enable Access Control
Enforce Authentication
Use TLS/SSL
Role-based Access Control
Encrypted Storage Engine
Regular Updates

- Network Exposure
- **User Authentication and Authorization**
  - SSL/TLS Encryption
  - Token-Based Authentication
  - Network Security
- **Adding Security to Web Servers**
  - HTTPS
  - HTTP Security Headers
  - Rate Limiting
  - Hide Nginx Version Number
  - Use Web Application Firewall (WAF)
- **Summary**

Chapter 11: Deploying Your Application

- **Deployment Overview**
  - Deployment Strategies
- **Deployment Preparation**
- **Sample Program: Deploying Blog Application on AWS**
  - Preparing Application
  - Creating Elastic Beanstalk Application
  - Creating Environment
  - Configuring Environment
  - Deploying Application
  - Setting up Continuous Deployment
- **Deployment Verification**
  - AWS Elastic Beanstalk Dashboard
  - View Application Version
  - Accessing the Application
  - Checking Logs
  - Monitoring
- **Summary**
- **Index**
- **Epilogue**

GitforGits

Prerequisites

This book is appropriate for readers with some background in C++ and nothing about back-end development. It's great for those just getting their feet wet in back-end development, as well as seasoned pros looking to hone their craft and learn something new. Whether you're a student, a professional, or a hobbyist, this book will teach you everything you need to know to master the art of C++ back-end development.

Codes Usage

Are you in need of some helpful code examples to assist you in your programming and documentation? Look no further! Our book offers a wealth of supplemental material, including code examples and exercises.

Not only is this book here to aid you in getting your job done, but you have our permission to use the example code in your programs and documentation. However, please note that if you are reproducing a significant portion of the code, we do require you to contact us for permission.

But don't worry, using several chunks of code from this book in your program or answering a question by citing our book and quoting example code does not require permission. But if you do choose to give credit, an attribution typically includes the title, author, publisher, and ISBN. For example, "Practical C++ Backend Programming by Justin Barbara".

If you are unsure whether your intended use of the code examples falls under fair use or the permissions outlined above, please do not hesitate to reach out to us at support@gitforgits.com .

We are happy to assist and clarify any concerns.

Acknowledgement

I owe a tremendous debt of gratitude to GitforGits, for their unflagging enthusiasm and wise counsel throughout the entire process of writing this book. Their knowledge and careful editing helped make sure the piece was useful for people of all reading levels and comprehension skills. In addition, I'd like to thank everyone involved in the publishing process for their efforts in making this book a reality. Their efforts, from copyediting to advertising, made the project what it is today.

Finally, I'd like to express my gratitude to everyone who has shown me unconditional love and encouragement throughout my life. Their support was crucial to the completion of this book. I appreciate your help with this endeavour and your continued interest in my career.

CHAPTER 1: INTRODUCTION TO BACKEND DEVELOPMENT

The internet and World Wide Web have undergone tremendous growth and change since their beginnings in the late 1980s and early 1990s. What began as a network primarily used by academic institutions for the purpose of sharing research has evolved into a pervasive global system that provides experiences and services to billions of people. In its infancy, the World Wide Web consisted primarily of text-based pages that were linked to one another using hypertext. However, as technological development progressed, so did the nature of the web. Interactivity was completely transformed by the introduction of dynamic content that was generated by a server, and the addition of multimedia formats such as images, video, and audio significantly improved user experiences. Businesses and services were able to spring up on the internet as a result of the proliferation of online transactions and web applications.

The distinction between frontend and backend development came about as a direct result of this evolution. The user interface (UI), styling, and markup that are sent to browsers became the primary focus of the frontend, also known as the client-side. The backend, also known as the server side, was responsible for handling the fundamental logic, data storage, and APIs that powered web experiences.

The client-server architecture is now the fundamental building block of the web ecosystem. The client sends the server requests, which are then processed by the server before the client receives appropriate responses. These responses may contain content that is either static or dynamic, data that was retrieved from databases, or other resources that are consumable through APIs. In a little more than 30 years, the World Wide Web has transformed from a network of static documents into a flourishing medium of experiences that is driven by advanced protocols, architectures, and technologies. The internet has an effect on almost every facet of contemporary life, making it possible for innovation to occur in a variety of fields. And there is no doubt that it will continue to develop at a frenetic pace, bringing with it fresh opportunities for how we can connect, share, and interact with one another.

Evolution of Web Ecosystem

The evolution of the web ecosystem is a fascinating journey that can be divided into several distinct phases. Each phase represents a significant shift in how the web was used and the technologies that powered it.

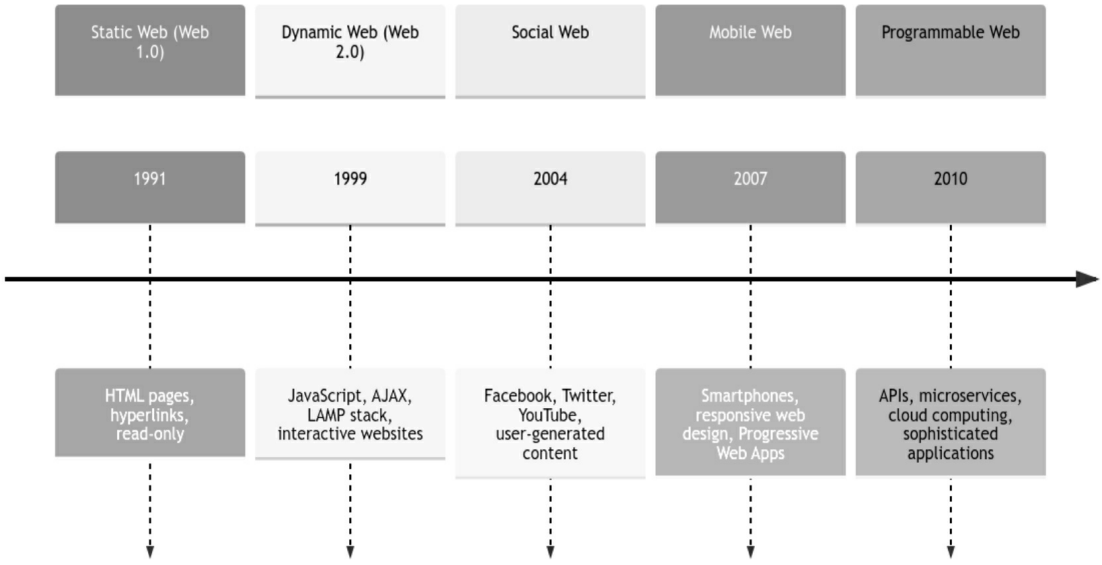


Fig 1.1 Timeline - Evolution of Web

The Static Web (Web 1.0)

The initial phase of the web was entirely static and read-only. This phase, often referred to as Web 1.0, consisted mainly of HTML pages connected via hyperlinks. These web pages were hosted on servers and viewed through web browsers. Interaction was minimal, with users only able to navigate through links. The content was created and published by specific individuals or organizations, and users were merely consumers of this content. There was no way for users to contribute content or interact with the web pages. This phase of the web was primarily about information dissemination, with the web serving as a vast, digital library.

The Dynamic Web (Web 2.0)

Web 2.0 marked a significant change with the introduction of dynamic, interactive websites. Thanks to technologies like JavaScript, AJAX, and the LAMP stack (Linux, Apache, MySQL, and PHP/Perl/Python), websites could now generate pages on the fly based on user interactions or database changes. This meant that the web pages were no longer static but could change in response to user actions. The web started to become a platform for applications, not just static content. This phase also saw the rise of user-generated content, with users now able to contribute content to the web. Blogs, wikis, and forums became popular during this phase.

The Social Web

The emergence of social media platforms like Facebook, Twitter, and YouTube pushed the

boundaries of what was possible on the web. These platforms not only allowed user-generated content but also fostered interactions between users in ways previously unimaginable. Users could now connect with each other, share content, and engage in discussions on the web. This phase saw a significant shift towards community-driven, participative content creation and sharing. The web became a social space, and the concept of social networking took off.

The Mobile Web

The explosion of smartphones and mobile devices ushered in the era of the mobile web. Websites had to adapt to be viewed on smaller screens and different devices, leading to the rise of responsive web design. This meant that web content and applications had to be designed to provide an optimal viewing and interaction experience across a wide range of devices, from desktop computer monitors to mobile phones. The line between web apps and native apps started to blur with the advent of technologies like Progressive Web Apps (PWA), which allowed web apps to be installed on a user's device and work offline.

The Programmable Web

Today, we are in the era of the programmable web, marked by APIs, microservices, and cloud computing. The web is no longer just about browsing websites. It is a platform for building sophisticated applications that can run in any environment and on any device. APIs allow different software applications to communicate with each other and exchange data, enabling the creation of complex, integrated applications. Microservices architecture allows applications to be broken down into smaller, independent services that can be developed, deployed, and scaled independently. Cloud computing provides the infrastructure to host these applications and services, offering scalability, flexibility, and cost-efficiency.

The evolution of the web has been a journey from static, read-only pages to dynamic, interactive applications. It has transformed from a medium for information dissemination to a platform for social interaction, mobile computing, and sophisticated application development. As we look to the future, we can expect the web to continue evolving and innovating, offering new possibilities and opportunities.

The Server Side: A Deeper Dive

In the web ecosystem, the server-side or backend, often just called the "server," is an essential component that fuels the functionality of web applications. It can be thought of as the brain of a web application, taking care of processing requests, business logic, data management, security, and much more.

Server Architecture

At a fundamental level, a server is a computer or system that manages access to centralized resources or services in a network. While the client, typically a web browser, runs on the user's device and provides the user interface for interaction, the server processes these interactions behind the scenes.

The server's primary role is to wait for requests from clients, process these requests, and then return the appropriate responses. To achieve this, the server typically hosts a web server software, such as Apache, Nginx, or IIS, that listens for HTTP(S) requests. When a request comes in, this software directs the request to the relevant application running on the server.

Backend Languages and Frameworks

The backend of a web application, which is hosted on a server and runs in the background, is an essential component that manages responsibilities such as interactions with databases, server configuration, and logic processing. There are many different programming languages that can be used to write backend applications. PHP, Python, Ruby, Java, JavaScript (Node.js), and C++ are some of the programming languages that are utilized most frequently today. The selection of a particular language is frequently subject to the impact of a number of different factors. These include the particular problem domain that the application is attempting to solve, the expertise of the team developing the application, the requirements for the application's performance, and the technology stack that is currently in place. The selection of a language to use in the process of development is a strategic decision because every language has particular advantages and can be applied to a wide variety of jobs and responsibilities.

In addition to the programming languages that are available, there are a large number of frameworks that can be used to assist in the development of the backend. These frameworks, such as Django for Python, Rails for Ruby, and Express for Node.js, provide pieces of code that have already been written to handle common backend responsibilities. They provide a structured method of building applications and come with a number of features that are already built-in, which reduces the amount of time and effort that is required to develop something from the ground up. These frameworks streamline the development process, making it faster and easier to build backend applications that are reliable, efficient, and secure. In addition to this, they ensure that appropriate programming practices are followed and make the code easier to maintain and scale.

Databases and Data Management

The backend of a web application plays a pivotal role in data management. Almost every web application necessitates the storage, retrieval, and manipulation of data, and this is where databases come into the picture. Databases can be categorized into two types: SQL-based and NoSQL-based. SQL-based databases, such as MySQL and PostgreSQL, are structured and use a predefined schema. They are excellent for handling complex queries and transactions, making them ideal for applications that require multi-row transactions like accounting systems or systems that need to manage a considerable amount of data.

On the other hand, NoSQL-based databases, such as MongoDB and Cassandra, are schema-less, meaning they allow the storage of data in multiple ways - key-value pairs, wide-column stores, graph databases, or document databases. This flexibility makes them suitable for applications that need to scale rapidly or have varied and evolving data structures.

The backend application interacts with the database to perform Create, Read, Update, and Delete (CRUD) operations, which are the four basic functions of persistent storage. These operations are executed based on the user's actions. For instance, when a user makes a post on a social media platform, the server processes this request, creates a new post in the database, and sends a response back to the client. This interaction between the client, server, and database is a fundamental part of how web applications function, enabling dynamic content generation based on user input.

Middleware and Business Logic

Middleware is a critical component in server-side programming. It serves as the bridge or 'middle' layer between the application's user interface and the data management systems. Middleware manages the communication or data exchange between the front-end and the back-end of an application. It handles various tasks such as routing, request parsing, session management, error handling, and more. Middleware functions are invoked sequentially, each passing control to the next until the appropriate handler is reached. This architecture allows developers to add, remove, or modify features easily without affecting other parts of the application. Middleware can also handle aspects like logging, setting HTTP headers, and managing cookies, providing a centralized location for these cross-cutting concerns.

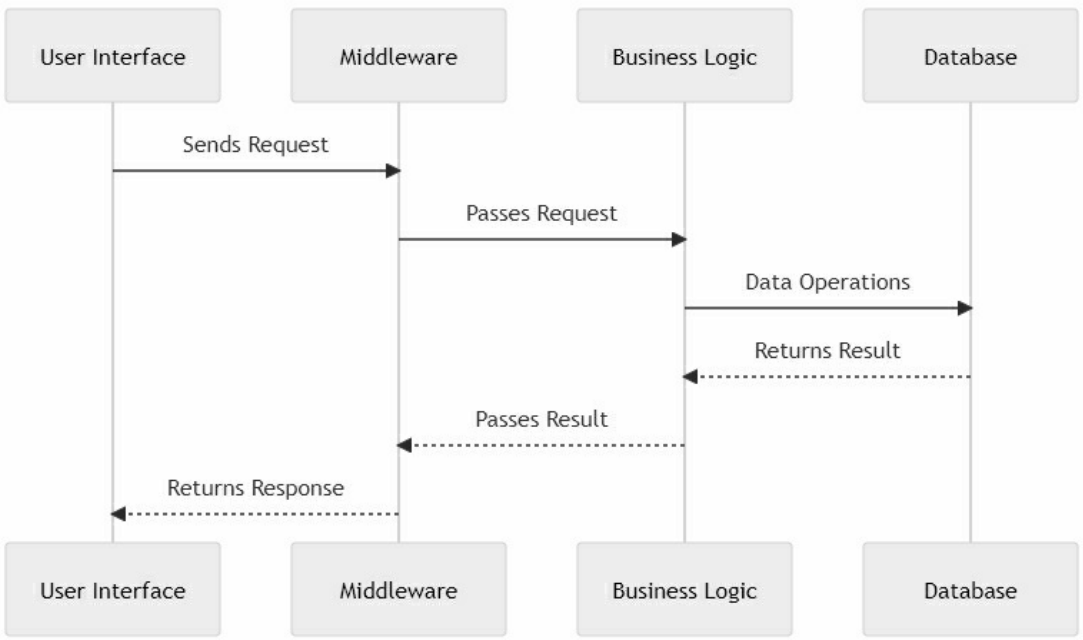


Fig 1.2 Interaction between User Interface, Middleware, Business Logic, and Database

The business logic layer is where the core functionality of an application resides. It's responsible for processing client requests, performing operations on the data, and implementing the rules and workflows specific to the application's domain. This includes tasks like data validation, authentication, and authorization, ensuring that only authorized users can perform certain actions. It also involves error handling, ensuring that the application can gracefully handle unexpected situations or invalid data. The business logic layer interacts with the data access layer to retrieve, update, or delete data in the underlying data storage. It's crucial to keep the business logic separate from the user interface and data access code, ensuring a clean separation of concerns and making the application easier to maintain and extend.

APIs and Microservices

In the realm of modern web development, the backend is often constructed as a collection of APIs, or Application Programming Interfaces, and microservices. APIs serve as the communication conduit between different software applications, enabling them to interact and exchange data. They act as a set of rules and protocols that specify how software components should interact, allowing different systems to communicate with each other regardless of their underlying architecture or technology. APIs have become the backbone of many services on the web, enabling a diverse range of applications from different vendors to interact seamlessly.

Microservices, on the other hand, represent a design approach where an application is broken down into smaller, independently deployable services. Each microservice is responsible for a specific functionality and can be developed, deployed, and scaled independently. This architectural style promotes better scalability as each service can be scaled independently based on demand. It also enhances maintainability as developers can update or fix individual microservices without impacting the entire application. This separation of concerns leads to systems that are easier to manage, more resilient, and more adaptable to changes.

Security and Optimization

Backend developers play a crucial role in maintaining the security of an application. They are tasked with ensuring that data is securely stored and that the application is safeguarded against potential attacks. This involves a multitude of tasks, including but not limited to, encryption of sensitive data, securely managing user sessions, and implementing robust authentication and authorization mechanisms. They also need to be vigilant about potential security vulnerabilities and stay updated with the latest security best practices and standards. This could involve tasks like regular code reviews, penetration testing, and adhering to secure coding principles. In essence, a backend developer acts as the first line of defense, protecting user data and maintaining the integrity of the application.

In addition to security, backend developers are also responsible for optimizing the application for speed and scalability. As the user base of an application grows, it's crucial that the application can handle the increasing load. This involves a variety of tasks such as implementing caching strategies to reduce server load, optimizing database queries for faster data retrieval, and setting up load balancers to distribute network traffic efficiently. They might also work on service partitioning and implement microservices architecture to ensure different parts of the application can scale independently. In essence, optimization tasks performed by backend developers ensure that the application remains responsive and available, providing a smooth user experience regardless of the load on the system.

In the following chapters, we will investigate each of these facets in greater depth, with a particular emphasis on how to implement them with the help of C++ and various other contemporary tools and technologies. This more in-depth understanding of the server-side will prepare you for the exciting journey of backend development, during which you will learn how to apply your C++ skills to the construction of web applications that are robust, efficient, and secure.

Exploring Backend Development

The art of backend development, as detailed above, involves multiple layers and components. These include working with servers, databases, business logic, APIs, security, and optimization - all of which require skilled programmers to operate efficiently. Although languages like Python, JavaScript (Node.js), and Ruby are often associated with backend web development, C++ has carved out a niche in areas demanding high-performance computing, real-time systems, and where fine control over memory management is required.

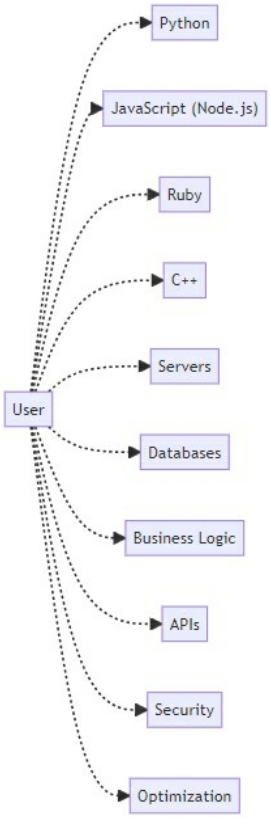


Fig 1.3 Summary of Layers in Server-side

High-Performance Systems

C++ is a compiled language known for its efficiency and control over system resources. It's faster than many other common backend languages, which makes it ideal for high-performance systems. These could be real-time bidding systems in ad tech, game servers in multiplayer online games, or financial trading systems where every millisecond counts.

Consider a real-time bidding system for an advertising platform. These systems often have to process tens or hundreds of thousands of ad auctions per second, each requiring complex computations to decide which ads to display. The low latency and high throughput provided by C++ can make a significant difference in such scenarios.

Systems Programming

C++ is widely used in systems programming, including developing web servers. Nginx, a high-performance HTTP server and reverse proxy server, is a prime example of this, written in C and C++. When building such systems, developers need to have a deep understanding of network programming, multithreading, concurrent processing, and more - areas where C++ shines.

Database Systems

The backend often requires robust interactions with databases. Many high-performance database systems, like MySQL and MongoDB, are implemented using C++. Backend developers working in C++ might find themselves optimizing these systems, writing database drivers, or even building custom database solutions tailored to their application's specific needs.

Microservices Architecture

Microservices architecture breaks an application into small, loosely coupled services. C++ can be used to write microservices where high performance is critical. Google, for example, has used C++ for several of its high-performance microservices.

Imagine a ride-sharing service like Uber, which could have separate microservices for handling ride requests, processing payments, and managing driver locations. If the service has millions of users, the microservice for managing driver locations might need to process massive amounts of data in real-time, making C++ an excellent choice for such a service.

APIs and gRPC

Modern backend development often involves building APIs, and here again, C++ has its place. Google's remote procedure call (RPC) system, gRPC, which allows for efficient communication between services, has support for C++. It uses Protocol Buffers as its interface definition language, enabling the definition of services and message types in a language-neutral way. gRPC and Protocol Buffers with C++ can create highly efficient, scalable, and fast APIs that can handle the heavy load.

Backend Security

Security is a significant aspect of backend development. C++ provides a high level of control over system resources, which can be a boon when writing secure code. Buffer overflows, a common security vulnerability, are easier to avoid in C++ than in C due to the Standard Template Library (STL) and features like std::array and std::vector. While memory management in C++ requires careful handling, this level of control allows for the creation of secure coding practices to prevent vulnerabilities.

Data Processing and Machine Learning

Backend systems often need to process large amounts of data or use machine learning for tasks like recommendations, personalization, and fraud detection. Libraries like TensorFlow for machine learning and Arrow for columnar in-memory data have first-class support for C++, making it an excellent choice for such backend tasks.

C++ Capabilities Overview

C++ continues to evolve, with regular updates adding new features and improvements to the language. Two of the most recent updates are C++20 and C++23. These introduce a range of new features that make the language more powerful, efficient, and easier to use. Some of these features are particularly beneficial for backend development.

C++20 Features

Concepts

Concepts, introduced in C++20, provide a way to express the intent of a template more clearly. They allow developers to specify constraints on template parameters, which can make templates easier to use and errors easier to understand. For instance, a developer can specify that a template parameter T must be a sortable container. The compiler will enforce this constraint, providing clear error messages when the constraint is not met. This feature can significantly improve the readability and maintainability of template-heavy code, which is common in complex backend systems.

Ranges

The ranges library, another addition in C++20, provides a new way to work with sequences of data. It promotes a more functional style of programming and can lead to code that is more readable and self-explanatory. This is particularly beneficial when processing large amounts of data in the backend, as it allows for more intuitive and concise data manipulation. Ranges can simplify complex data transformations and make the code easier to understand and maintain.

Coroutines

Coroutines simplify asynchronous programming by making asynchronous code appear as if it were synchronous. This is a game-changer in backend development, where handling multiple tasks simultaneously, such as processing multiple client requests, is common. Coroutines can make the code more readable and less error-prone by eliminating the need for callbacks or promises, which are traditionally used in asynchronous programming. This can lead to cleaner, more maintainable code that is easier to reason about.

Modules

Modules, another feature introduced in C++20, can help reduce compilation times and make it easier to manage dependencies. This can be particularly beneficial in large backend codebases, where compilation time can be a significant issue. Modules allow for better encapsulation and separation of code, making the codebase easier to navigate and understand. They also improve build times and can help prevent issues related to name collisions and macros.

Format Library

C++20 introduced std::format, a new way to format strings. It's safer and more feature-rich than printf, and more type-safe than iostreams. This is particularly beneficial for generating responses or logs in a backend context. The std::format function provides a wide range of formatting capabilities and is designed to be type-safe, extensible, and intuitive to use. This can help prevent

common errors associated with string formatting and manipulation, leading to safer and more robust backend applications.

C++20 introduces a number of new features that, when combined, have the potential to significantly boost the productivity, safety, and readability of backend development. While concepts and ranges have the potential to make the code easier to understand and maintain, coroutines have the potential to make the management of asynchronous tasks more straightforward. Modules can improve compilation times and code organization, and the newly released format library offers a safer and more flexible way to manage string formatting. Both of these improvements can be made through the use of modules. When utilized correctly, these features have the potential to result in more dependable and effective backend systems.

C++23 Features

Networking Library

One of the most anticipated features of C++23 is the proposed addition of a networking library to the standard library. This library, which is based on Boost.Asio, is expected to provide a cross-platform method to handle sockets, timers, and networking protocols. The inclusion of this library in the standard library can greatly simplify the creation of web servers and other networking tasks in backend development. It would provide developers with a standardized, high-level interface for networking, reducing the need for third-party libraries and making code more portable and maintainable.

Executors

Executors are another significant feature proposed for C++23. They represent an abstraction of where and how tasks are run. Executors can be used to control the execution of tasks, such as running tasks in a thread pool, a GPU, or a distributed system. This feature could be particularly beneficial for handling large loads in backend systems. By allowing developers to control where and how tasks are executed, executors can help optimize resource usage and improve performance in multi-threaded and distributed applications.

Coroutine Improvements

C++23 also aims to improve coroutines based on the feedback and experience from C++20. Coroutines are a powerful tool for asynchronous programming, allowing developers to write asynchronous code in a synchronous style. This can make asynchronous programming in backend development much easier and more intuitive. The proposed improvements in C++23 are expected to make coroutines even more powerful and flexible, further enhancing their usefulness in backend development.

Improved Algorithms

There's a proposal to add more parallel and vectorized versions of existing algorithms in C++23. These improved algorithms can significantly enhance performance when processing large amounts of data in the backend. By taking advantage of modern hardware capabilities, these parallel and vectorized algorithms can process data more quickly and efficiently, making them ideal for high-performance backend systems that need to handle large volumes of data.

Reflection

Reflection, which is the ability for a program to observe and modify its own structure and behavior, might be added in C++23. This feature could have many uses in backend development, from serializing and deserializing data to generating APIs automatically. Reflection can make code more flexible and adaptable, allowing developers to write more generic and reusable code. It can also simplify many tasks in backend development, such as validating input data, generating documentation, or implementing serialization and deserialization for data storage and transmission.

C++ can become a more powerful and easier to use language for backend development thanks to new features introduced in C++20 and C++23. They demonstrate that C++ is continually improving and progressing in order to fulfill the requirements of contemporary software development. In the chapters that follow, we will investigate how these features can be utilized within the framework of backend development.

Setting up C++ Backend Environment

Setting up a C++ development environment for backend programming involves installing a C++ compiler, an Integrated Development Environment (IDE) or text editor, and several libraries that will help you with various aspects of backend development. The give below is a step-by-step walkthrough:

Installing a C++ Compiler

To write and run C++ code, you'll need a C++ compiler. If you are using:

Windows

Install MinGW. It provides a complete Open Source programming tool set which is suitable for the development of native Windows programs.

macOS

Xcode command line tools can be installed. You can do this by typing xcode-select --install in the terminal.

Linux

You'll likely already have GCC installed. If not, you can install it via your distribution's package manager.

Installing an IDE/Text Editor

Next, you'll need a text editor or an Integrated Development Environment (IDE) to write your code. Following are a few good options:

Visual Studio Code

This free, open-source editor supports C++ via an extension and has integrated support for Git. It also has a built-in terminal and support for extensions to add more features.

CLion

A cross-platform IDE for C++. It's not free, but it comes with many features that make C++ development easier.

Sublime Text

A popular text editor with support for many programming languages including C++. You can add more features through plugins.

Installing Essential C++ Libraries

Backend development with C++ can be a complex task, but fortunately, there are numerous libraries available that can simplify the process and enhance your productivity. Given below are some essential libraries that you should consider installing for backend development with C++:

Boost.Asio or Standalone Asio

These are powerful libraries for network and low-level I/O programming, which are often used for servers and networking applications. Boost.Asio is part of the larger Boost library collection, while Standalone Asio can be used independently of Boost. Both libraries provide a wealth of features for asynchronous I/O operations, making them ideal for high-performance networking applications.

CPR or libcurl

These libraries are instrumental in making HTTP requests. CPR is a modern, easy-to-use library that provides a high-level API for making requests and handling responses. On the other hand, libcurl is a bit lower-level but offers more flexibility and control. Both libraries support a wide range of protocols and features, including HTTPS, cookies, and authentication.

Nlohmann::json

This is a modern C++ library for handling JSON data. It offers a straightforward, intuitive API and is very efficient, making it an excellent choice for applications that need to process JSON data.

Fmt

This is a formatting library for C++. It offers a more type-safe and faster alternative to printf and iostreams. The library provides a set of format specifiers for various data types, making it easy to create formatted strings and output.

Spdlog

This is a fast, easy-to-use logging library for C++. It supports many sinks (destinations for log messages), including rotating log files, console logging, syslog, and more. The library also provides a variety of log levels and formatting options, making it a versatile tool for application logging.

Catch2

This is a modern, C++-native, header-only, test framework for unit-tests, TDD (Test-Driven Development), and BDD (Behavior-Driven Development). It's easy to use and requires minimal setup, making it a great choice for adding tests to your C++ projects.

Google's Protocol Buffers and gRPC

These are powerful tools for building APIs and microservices. Protocol Buffers is a language-neutral, platform-neutral, extensible mechanism for serializing structured data. gRPC is a high-performance, open-source universal RPC framework that uses Protocol Buffers as its interface definition language. Together, they allow you to define services and message types in .proto files, generate service stubs and client libraries, and then use these in your application.

To install these libraries, you can download and build them from their GitHub repositories, or you can use a package manager:

- On Windows, you can use vcpkg.
- On macOS, you can use Homebrew.

- On Linux, you can use your distribution's package manager, like apt for Ubuntu.

Verify the Setup

Once you've installed your compiler, IDE, and libraries, it's a good idea to verify that everything works:

- Try compiling and running a simple C++ program, like a "Hello, World!".
- Try using one of the libraries you've installed. For instance, you could write a small program that makes a HTTP request with CPR or libcurl and prints the response.

Once you've confirmed everything is working, you're ready to start backend programming with C++! In the subsequent chapters, we shall explore how to use the libraries we've installed to build robust and efficient backend systems.

Summary

In this chapter, we started our journey into the world of backend development with C++ by providing a general overview of the web ecosystem and how it has developed over time. This was the first step on our journey. We simplified the seemingly complicated process of server-side development by highlighting the significance of a number of components, such as servers, databases, APIs, caching, and business logic, and the crucial part that each of these components plays in easing the process by which users interact with the web. We also brought attention to the growing demand for performance-intensive and real-time systems, which is where C++ really shines due to the fact that it is both efficient and capable of producing low latency.

After that, we dove deeper into the specific applications of C++ that can be found in the backend ecosystem. C++ has proven to be an extremely useful tool for a wide range of applications, including the development of high-performance systems, systems programming, and database systems; the creation of microservices and APIs by utilizing gRPC; the improvement of backend security; and the processing of massive amounts of data. The provided real-world examples shed light on how high-performance microservices are developed using C++ at global tech behemoths like Google.

In the end, we looked at how the most recent C++ standards, C++20 and C++23 could improve backend programming by examining the value that is added by their features. The concepts, ranges, coroutines, modules, and the format library introduced in C++20, as well as the networking library, executors, improvements to coroutine algorithms, and reflection introduced in C++23, provide programmers with a language that is more powerful, more efficient, and more user-friendly. Then, in order to get you ready for actual backend programming with C++, we walked you through the process of configuring your C++ environment, which included the installation of necessary C++ compilers, integrated development environments (IDEs), and libraries. Finally, that the initial stage of the thrilling adventure that is the development of a C++ backend has been finished, the stage is set for more in-depth exploration in the subsequent chapters.

CHAPTER 2: C++ REFRESHER AND ESSENTIALS

Potential of C++ for Backend Programming

As we move forward into the second part of this adventure, we are going to focus our attention on the power that C++ possesses for backend programming and the creation of dependable and trustworthy backend software applications. C++ is not typically thought of as a language used in web development; however, due to its speed, scalability, and flexibility, it is an extremely powerful tool that can be used to create the foundation of web applications.

C++ has a long-standing reputation for being an efficient and effective programming language. It provides granular control over system resources, which enables software developers to enhance the performance of their applications in terms of both memory and processing speed. This can be especially helpful in the backend development process, which requires high-performance servers that can handle thousands of requests per second while maintaining a low level of latency. In addition, C++ allows for both procedural and object-oriented programming, and in more recent years, it has even begun to support metaprogramming and functional programming techniques. This versatility enables you to use the appropriate tool for the task at hand, which in turn results in code that is cleaner, more understandable, and easier to maintain.

Another important consideration is how well C++ scales up to larger systems. You will need a language that is capable of scaling in order to keep up with the increasing demands placed on a server. You are able to create applications that are highly effective when using multiple threads and C++, which allows you to make the most of today's multicore processors. C++ is an excellent choice for high-load systems because of this, in addition to the low-level capabilities it offers, which allow you to fine-tune your application to make it run as efficiently as possible on any given piece of hardware.

C++'s extensive ecosystem of libraries and frameworks is another essential component of the programming language. This includes libraries for networking, concurrency, database access, and a variety of other functions, all of which have the potential to significantly accelerate the development process. It also comes with powerful tools for debugging and profiling, which can assist you in locating and addressing performance bottlenecks, thereby ensuring that your backend operates as smoothly as is humanly possible.

The compatibility and interoperability of C++ is one of the programming language's strongest selling points, particularly when it comes to backend development. C++ has the ability to communicate with virtually every other programming language and can run on almost any operating system, including Windows, Linux, and macOS. Because of this, a backend that is written in C++ has the capability of serving a web frontend, a mobile app, a desktop app, or even other servers. Additionally, the interoperability of C++ makes it possible to integrate components written in languages other than C++. This is especially helpful when utilising pre-existing libraries or integrating with various other types of software.

The more recent versions of C++, specifically C++20 and the features that are being considered for C++23, have added new capabilities that make the programming language easier to use. Coroutines, for example, make it much simpler to write non-blocking code, which is an essential component of high-performance servers. Asynchronous programming can also be made much simpler by their addition. The power of templates has also been increased thanks to the

introduction of concepts and the ranges library, which has resulted in code that is more expressive and safer.

Last but not least, the practical applications of C++ make its potential abundantly clear. C++ is used in a wide variety of high-traffic and high-performance systems, such as the search backend of Google, Facebook's real-time services, and a significant number of Amazon's and Microsoft's core services. These powerhouses put their faith in C++ because of its reputation for delivering speed, dependability, and effective management of resources.

C++ Syntax

Let us revisit some fundamental C++ syntax and how they may be applied in a backend setting:

Variables and Data Types

In a backend setting, variables of various data types would commonly be used to store and manipulate data that is retrieved from or to be written to a database. For example, an int might be used to store the number of users, a std::string to store a user's name, and a bool to check if a user is online.

```
int num_users = 500;
std::string user_name = "Alice";
bool is_user_online = false;
```

Control Structures

Control structures are critical in backend development, as they allow for decision-making in the code based on different conditions. For instance, an if-else statement could be used to verify a user's age before allowing them to register for a service.

```
if (user_age >= 18) {
   std::cout << "Registration successful!";
} else {
    std::cout << "Sorry, you must be at least 18 years old to register.";
}
```

Functions

Functions often represent actions that can be performed on the data. They can handle various tasks, such as data validation, calculations, and database operations. Functions also help in maintaining clean and organized code.

```
void update_user_age(std::string user, int new_age) {
    // Code to update a user's age in the database
}
int calculate_average_age() {
```

```
    // Code to calculate the average age of all users
}
```

Classes and Objects

Classes can be used to represent entities in your backend application, such as a User, Product, or Order. Each class will have properties and methods that define what data the entity holds and what operations can be performed on the data.

```
class User {
public:
  std::string name;
  int age;
  User(std::string user_name, int user_age) {
      name = user_name;
      age = user_age;
  }
  void update_age(int new_age) {
      age = new_age;
  }
};
```

STL Containers

The Standard Template Library (STL) in C++ provides a set of common classes for handling data. These include containers like std::vector, std::list, std::map, which can be very helpful in backend programming when dealing with collections of data.

```
std::vector<User> users;  // A list of users
std::map<int, User> user_id_to_user;  // A mapping from user IDs to User objects
```

Error Handling

In a backend setting, there are many potential sources of errors, such as network issues, database errors, or invalid data. Exception handling in C++ provides a way to react to these exceptional circumstances and take appropriate action, preventing the application from crashing and providing meaningful error messages.

```
try {
   User user = get_user_from_database(user_id);
} catch (const std::exception& e) {
    std::cerr << "Caught exception: " << e.what() << std::endl;
    // Handle the error (e.g., return an error response to the client)
}
```

These are just the basics of C++ syntax and their potential applications in backend programming. They are the fundamental building blocks of writing C++ code for backend applications. As we move forward, we shall explore more advanced topics, including multithreading, networking, and working with databases, where the power of C++ will truly shine.

C++ Semantics

C++ semantics, or the set of rules that determine the behavior of the C++ programming constructs, play a crucial role in backend programming. Let us look into some key semantic features of C++ and how they apply in the context of backend programming.

Value Semantics

C++ is known for its strong value semantics. In essence, when an object is assigned to a new variable or passed to a function, a copy of the object is made by default. This can be particularly beneficial in a backend context, as it allows for easy and safe manipulation of data without affecting the original source. For instance, if you have an integer 'a' and you assign 'a' to a new integer 'b', 'b' becomes a copy of 'a'. Any changes made to 'b' will not affect 'a', ensuring data integrity.

```
int a = 5;
int b = a;  // b is a copy of a, not a reference
b += 5;  // changing b doesn't affect a
```

Reference Semantics

Despite its value semantics, C++ also supports reference semantics using reference (&) and pointer (*) types. This can be useful in backend programming when you want to manipulate large data structures without incurring the cost of copying or when you want changes to a variable to be reflected in multiple parts of your application. For example, if you create a reference to 'a' called 'ref_to_a', any changes made to 'ref_to_a' will also change 'a'.

```
int a = 5;
int& ref_to_a = a;  // ref_to_a is a reference to a
ref_to_a += 5;  // changing ref_to_a also changes a
```

Object-Oriented Programming (OOP)

C++ offers class-based OOP with features like inheritance, encapsulation, and polymorphism. In the context of backend programming, OOP allows you to model complex systems in an intuitive way, encapsulating data and functionality together in classes. For instance, you could create a 'User' class that encapsulates user-related data and functionality, such as sending a message.

```
class User {
public:
  std::string name;
  // ...
  void sendMessage(const std::string& message);
};
```

Resource Acquisition is Initialization (RAII)

It's idiomatic to acquire resources (like memory, file handles, or network connections) in constructors and release them in destructors. This is critical in backend development, where resource management is key to maintaining performance and preventing leaks or contention. For example, you could create a 'DatabaseConnection' class that acquires a connection in its constructor and releases it in its destructor.

```
class DatabaseConnection {
public:
  DatabaseConnection(const std::string& dsn) {
      // acquire the connection
  }
  ~DatabaseConnection() {
      // release the connection
  }
};
```

Templates and Generic Programming

C++ provides powerful mechanisms for generic programming, primarily through templates. This can be leveraged in backend development to write flexible, reusable code. For instance, you could write a 'max' function template that works with any type that supports comparison.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Concurrency

Modern C++ provides extensive support for multithreading and concurrency, which are crucial in high-performance backend systems. For instance, you can create a new thread to perform some work, and then wait for it to finish using the 'join' method.

```
std::thread worker(do_some_work);
// ...
worker.join();
```

Exception Handling

Exceptions in C++ provide a robust way of dealing with error conditions, which are commonplace in backend systems, especially when dealing with unpredictable network conditions or external services. For instance, you can use a 'try-catch' block to handle potential errors in a risky operation.

```
try {
   // risky operation
} catch (const std::exception& e) {
   // handle error
}
```

Memory Management

With explicit control over memory allocation and deallocation, C++ can efficiently handle

resources, which is crucial in a backend setting where performance and resource utilization are paramount. For instance, you can create a dynamic array using the 'new' keyword, and then delete it using the 'delete' keyword when you're done.

```
int* dynamic_array = new int[100];
// ...
delete[] dynamic_array;
```

In essence, the rich semantics of C++ provide great control over system resources and offer a powerful toolbox to create efficient and performance-critical backend systems.

Move Semantics

Introduced in C++11, move semantics allow the resources of a temporary object to be moved into another object without costly deep-copy operations. This can be particularly useful in backend development, where we often deal with large amounts of data, and efficiency is paramount.

```
std::vector<int> vec1 = {1, 2, 3};
std::vector<int> vec2 = std::move(vec1);  // Moves data from vec1 to vec2
```

Lambda Expressions

C++ supports anonymous functions or lambdas. They are very useful for writing callbacks and functions that can be passed as arguments. This is particularly useful in asynchronous programming in backend development.

```
auto add = [](int a, int b) -> int { return a + b; };
int sum = add(5, 7);
```

Smart Pointers

Smart pointers are a feature of C++ that provides automatic memory management, ensuring no memory leaks occur. They are particularly useful in backend programming where dynamic resource allocation often happens, and manual deallocation could be error-prone.

```
std::shared_ptr<int> ptr(new int(10)); // When ptr goes out of scope, memory will be automatically deallocated
```

Type Inference

The auto keyword in C++ allows automatic type inference. It can be handy in backend programming when dealing with complex data types, reducing verbosity, and making the code cleaner.

```
auto num = 5;  // num is int
auto str = "Hello, World!";  // str is const char*
```

Standard Template Library (STL)

The STL is a library within C++ that provides several generic classes and functions, which includes but is not limited to, a powerful set of algorithms, containers like vectors and maps, and iterators. In backend programming, STL can be used to solve complex data handling problems efficiently and easily.

```
std::vector<int> nums = {4, 2, 5, 1, 3};
std::sort(nums.begin(), nums.end());  // sorts the vector
```

In essence, the rich semantics of C++ provide great control over system resources and offer a powerful toolbox to create efficient and performance-critical backend systems. Whether it's managing memory, handling exceptions, or leveraging the power of OOP, understanding C++ semantics is key to effective backend development.

C++ Data Structures

Data structures are essential tools in any developer's arsenal, forming the backbone of many programming tasks. In this section, we shall cover several fundamental data structures in C++ — arrays, strings, lists, and vectors — and their relevance in backend development.

Arrays

An array is a sequence of elements of the same type placed in contiguous memory locations. They can be used to store a fixed-size sequential collection of elements.

```
int arr[5] = {1, 2, 3, 4, 5};
```

Arrays are extremely efficient as the size is known at compile-time, which can be beneficial for backend tasks where performance is critical and the size of data is known beforehand. However, arrays have fixed size, and they can't grow or shrink dynamically.

Strings

Strings in C++ are implemented as an array of characters ending with a special character \0. The C++ Standard Library provides a string class type (std::string) that supports a wide range of string manipulations.

```
std::string greeting = "Hello, World!";
```

In backend programming, strings are omnipresent — they are used to represent text data, manipulate and store information from databases, handle user input, generate HTML responses, etc.

Lists (std::list)

The std::list is a container provided by the Standard Library that consists of a doubly-linked list. Lists allow efficient insertions and deletions at any location within the list.

```
std::list<int> mylist = {7, 5, 16, 8};
```

In the context of backend development, lists can be useful when you need to perform frequent insertions and deletions from the list, and order of elements matters. However, accessing elements in a list is not as quick as in an array or vector, since you can't access elements directly by their index.

Vectors (std::vector)

A std::vector is a dynamic array, and it's one of the most commonly used data structure in C++. It allows easy access to individual elements by their position index and provides relatively efficient addition and removal of elements from its end.

```
std::vector<int> myvector = {1, 2, 3, 4, 5};
```

Vectors are incredibly versatile and useful in backend development. They can be used to store and manipulate collections of data, and thanks to dynamic resizing, they're suitable when you don't know the size of the data at compile time. They're useful in various backend tasks, such as storing and processing user data, holding database query results, or buffering data from a network stream.

OOP in C++

Object-Oriented Programming (OOP) is a programming paradigm that is centered around the concept of "objects." These objects are instances of classes, which can be thought of as user-defined data types. The main pillars of OOP are encapsulation, inheritance, and polymorphism, all of which find extensive application in backend programming.

Classes and Objects

Classes provide a blueprint for creating objects (a particular data structure), containing variables and functions into one single unit. In the context of backend programming, classes can be used to represent entities or concepts.

For example, in a blog application backend, a class BlogPost could represent a blog post:

```
class BlogPost {
public:
    BlogPost(const std::string& title, const std::string& author)
        : title(title), author(author) {}
    std::string getTitle() const { return title; }
    std::string getAuthor() const { return author; }
private:
   std::string title;
   std::string author;
   // More properties...
};
```

An object is an instance of a class. Creating an object is like declaring a variable of a particular type:

```
BlogPost post("My First Post", "John Doe");
```

Inheritance

Inheritance allows classes to inherit commonly used state and behavior from other classes. In C++, a class can inherit from one parent class, leading to a hierarchy of classes.

Suppose we have a User class in our backend and we want to create a AdminUser class that has

all the properties of User, plus some additional capabilities. We can achieve this through inheritance:

```
class User {
public:
    User(const std::string& username) : username(username) {}
    std::string getUsername() const { return username; }
   // Other User methods...
private:
   std::string username;
};
class AdminUser : public User {
public:
   AdminUser(const std::string& username)
       : User(username) {}
   void performAdminTask() { /*...*/ }
};
```

Polymorphism

Polymorphism allows us to perform a single action in different ways. It provides a way to structure and partition your code in a flexible and reusable way. Polymorphism can be achieved in C++ through function overloading, operator overloading, and interfaces (abstract classes).

For example, we could have a Database class in our backend, and subclasses for different types of databases like MongoDB, PostgreSQL etc. Each subclass can implement the same set of methods (like connect, query, etc.), but the implementation will be different for each. This way, we can write our backend code to work with a Database object, and it can actually be an instance of any subclass:

```
class Database {
public:
   virtual void connect() = 0;
```

```
    virtual void query(const std::string& sql) = 0;
   // Other Database methods...
};
class MongoDB : public Database {
public:
    void connect() override { /* MongoDB-specific connection code */ }
    void query(const std::string& sql) override { /* MongoDB-specific query code */ }
};
class PostgreSQL : public Database {
public:
    void connect() override { /* PostgreSQL-specific connection code */ }
    void query(const std::string& sql) override { /* PostgreSQL-specific query code */ }
};
```

OOP principles are crucial to designing and organizing good backend code. They allow us to think about our code in terms of real-world entities and their interactions, making the code more understandable and manageable. By structuring our code around objects and leveraging the principles of encapsulation, inheritance, and polymorphism, we can create robust and maintainable backend systems.

Standard Template Library

The Standard Template Library (STL) is a powerful feature of C++ that provides a collection of templates for common data structures and algorithms. It offers a rich set of components which can greatly accelerate backend development. In this section, we will look at some of these components, specifically containers, algorithms, and function objects, which can be instrumental in structuring and manipulating data.

Containers

A container is a holder object that stores a collection of other objects. They are implemented as class templates. The most commonly used containers in backend development include std::vector, std::map, std::set, and std::unordered_map.

Std::vector

It's a dynamic array, as we've already discussed. Useful when you need a resizeable array-like container.

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

std::map and std::unordered_map

These are associative containers that store elements in a key-value pair. std::map stores keys in a sorted manner, while std::unordered_map stores keys using a hash table, which often leads to faster lookup times. They can be useful for storing and accessing data by keys, similar to how you'd use a database.

```
std::map<std::string, int> map = {{"apple", 5}, {"orange", 10}};

std::unordered_map<std::string, int> unordered_map = {{"apple", 5}, {"orange", 10}};
```

std::set and std::unordered_set

These containers allow storage of unique elements. std::set keeps the elements in a sorted order, while std::unordered_set uses a hash table for storage. They can be useful when you need to quickly check if an element exists in a collection.

```
std::set<int> set = {1, 2, 3, 4, 5};

std::unordered_set<int> unordered_set = {1, 2, 3, 4, 5};
```

Algorithms

The STL provides a set of algorithms that can be used on STL containers. These algorithms

cover a range of functionalities including searching, sorting, modifying, and more. A couple of examples:

std::sort()

This algorithm sorts the elements in a range. This is often used in backend development to sort data before sending it to a client or processing it further.

```
std::vector<int> vec = {5, 3, 4, 1, 2};
std::sort(vec.begin(), vec.end());
```

std::find()

This algorithm is used to find an element in a container. It's useful when you need to check if a certain value is in a container.

```
std::vector<int> vec = {5, 3, 4, 1, 2};
auto it = std::find(vec.begin(), vec.end(), 3);
```

Function Objects (Functors)

A function object, or a functor, is an object that can be called as if it is a function. You can create a functor by defining the operator () in a class. Functors can be used as arguments to STL algorithms, offering a way to customize the behavior of an algorithm.

For example, you can create a functor to compare two BlogPost objects, and use this to sort a std::vector of BlogPost objects:

```
class BlogPost {
public:
    BlogPost(const std::string& title) : title(title) {}
    std::string getTitle() const { return title; }
private:
   std::string title;
};
class CompareBlogPosts {
public:
    bool operator()(const BlogPost& a, const BlogPost& b) const {
```

```
    return a.getTitle() < b.getTitle();
  }
};
std::vector<BlogPost> posts = {BlogPost("Post 2"), BlogPost("Post 1")};
std::sort(posts.begin(), posts.end(), CompareBlogPosts());
```

This design pattern allows you to write code that is easily reusable and adaptable to a wide range of scenarios, which is often needed in backend development. The STL is an invaluable part of C++ that every backend developer should be familiar with.

Error Handling

In C++, error handling is often accomplished through the use of exceptions. An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an error condition is detected, an exception can be thrown, which then needs to be caught and handled in some way.

Throwing Exceptions

You can throw an exception using the throw keyword followed by an exception object. This object can be of any type, but it's common to use the exception types provided by the C++ Standard Library (std::exception and its subclasses), or to define your own exception types.

Given below is an example in the context of backend programming. Suppose you have a function that retrieves a BlogPost object from a database by its ID:

```
BlogPost getBlogPost(int id) {
    // Fetch the BlogPost from the database...
    if (/* failed to fetch the BlogPost */) {
        throw std::runtime_error("Failed to fetch the blog post from the database");
    }
    // Return the fetched BlogPost...
}
```

Catching Exceptions

When an exception is thrown, the control flow of the program jumps to the nearest catch block that can handle the type of exception thrown. If no such catch block is found, the program terminates.

To catch exceptions, you use a try/catch block. You put the code that may throw an exception inside the try block, and the code to handle the exception in the catch block.

Given below is an example:

```
try {
    BlogPost post = getBlogPost(1);
    // Process the fetched BlogPost...
```

```
} catch (const std::exception& e) {
    // Log the error message and handle the error condition...
    std::cerr << "An error occurred: " << e.what() << std::endl;
}
```

In the catch block, we catch exceptions by reference to avoid slicing (i.e., losing data from derived exceptions). We catch std::exception, which is the base class for all exceptions in the Standard Library, so this will catch all standard exceptions.

The what() member function of std::exception returns a message describing the error, which we log to the standard error stream (std::cerr).

Custom Exception Types

In some cases, you might want to define your own exception types. This can be done by defining a class that inherits from std::exception (or one of its subclasses), and overriding the what() member function to provide a custom error message.

For example:

```
class DatabaseException : public std::runtime_error {
public:
   DatabaseException(const std::string& message)
       : std::runtime_error(message) {}
};
BlogPost getBlogPost(int id) {
    // Fetch the BlogPost from the database...
    if (/* failed to fetch the BlogPost */) {
       throw DatabaseException("Failed to fetch the blog post from the database");
   }
   // Return the fetched BlogPost...
}
```

Multithreading

Multithreading is a common technique in backend development for improving the performance and responsiveness of applications by allowing multiple tasks to be performed concurrently. C++ provides robust support for multithreading through its threading library.

Create New Thread

The basic unit of execution in C++ is a thread. Each thread runs independently of others, and each has its own execution path and its own stack memory. Creating a new thread is simple. You just need to instantiate a std::thread object and provide a function for the new thread to execute.

Given below is a simple example:

```
#include <thread>
void doWork() {
    // Code for the new thread to execute...
}
int main() {
   std::thread worker(doWork);
   // ... other code ...
    // Wait for the worker thread to finish before exiting the main thread.
   worker.join();
}
```

In the above sample program, a new thread is launched that runs the doWork function concurrently with the rest of the code in main. The join function is used to ensure that the main thread waits for the worker thread to finish before it exits.

Data Sharing and Synchronization

In a multithreaded program, multiple threads may need to share data. However, because threads run independently, there's a risk of data races, where two threads try to access and modify the same data concurrently, leading to undefined behavior.

To avoid this, C++ provides various synchronization primitives, such as std::mutex and std::lock_guard. Given below is an example of how they can be used:

```
#include <thread>
#include <mutex>
std::mutex mtx;
int sharedData = 0;
void incrementData() {
  std::lock_guard<std::mutex> lock(mtx);
  ++sharedData;
}
int main() {
  std::thread t1(incrementData);
  std::thread t2(incrementData);
  t1.join();
  t2.join();
   // At this point, sharedData is guaranteed to be 2.
}
```

In the above sample program, the incrementData function is executed by two threads concurrently. However, because it locks the mtx mutex before incrementing sharedData, only one thread can increment sharedData at a time, avoiding a data race.

Application in Backend Development

Multithreading in backend development enhances performance by allowing concurrent handling of multiple requests. Each incoming request can be assigned to a new thread, ensuring uninterrupted processing. This parallel execution optimizes resource utilization, improves response times, and increases the overall throughput of the system.

Given below is a simplified example:

```
#include <thread>
#include <vector>
void handleRequest(Request req) {
```

```
  // Process the request...
}
int main() {
  std::vector<std::thread> workers;
  while (true) {
      Request req = getIncomingRequest();
      workers.push_back(std::thread(handleRequest, req));
  }
   // In a real application, you'd need to join the threads at some point.
}
```

In the above sample program, for each incoming request, a new thread is launched that runs the handleRequest function. This allows the server to handle multiple requests concurrently.

Summary

In this chapter, we have taken a more in-depth look at the ways in which the programming language C++ can be utilized effectively for backend development. In the beginning of the chapter, we discussed the possibility of using C++ for backend programming. We also reviewed some of the reasons why C++ is a good option for the development of the backend of software applications, including its high level of efficiency, its high level of robustness, and its flexibility.

Our travels took us on an enlightening excursion through the syntax and semantics of C++, illustrating how these aspects of the language can be applied within the framework of backend development. We went over several data structures such as arrays, strings, lists, and vectors, and we also dove deeper into the object-oriented programming concepts of classes, objects, inheritance, and polymorphism. Arrays, strings, lists, and vectors are examples of data structures. The power of the Standard Template Library (STL) was then revealed to us, and our attention was focused on the various templates and components contained within it that are essential for backend development.

Near the end of the chapter, we turned our attention to the error handling and multithreading capabilities that are available in C++. We were able to learn how to write code that was more robust and free of errors by investigating and comprehending exceptions as well as their use for handling errors. The investigation of multithreading led us to discover a method that, when applied, enables the concurrent execution of tasks with the intention of enhancing the efficiency and receptivity of backend applications. In general, the chapter was very helpful in that it provided us with a comprehensive understanding of how to effectively use C++ for backend programming. This laid a solid foundation for further investigation as well as practical application of the material.

CHAPTER 3: DEEP DIVE INTO ALGORITHMS

Algorithms Overview

Backend systems are significantly influenced in terms of their overall functionality and level of efficiency by the algorithms that power them. They are well-defined sets of instructions that provide a solution to a particular problem or undertaking. When it comes to the backend programming of a website, every single task, including the management of a database, the processing of requests, and the operations performed on data, is driven by an algorithm.

The development of an algorithm has a significant impact on both its efficiency and its dependability once it is implemented. The response time is improved, and the load on the server is decreased, when efficient algorithms are used. On the other hand, an algorithm that is poorly designed can result in an inefficient use of resources, which can lead to slower response times and even the failure of the system. As a result, the ability to craft algorithms that are well-structured and efficient is an essential skill for backend developers.

Consider the process of looking for specific data within a large database, which is a common task in backend development. A linear search algorithm would be the most straightforward method to take, as it would involve simply going through the process of iterating over the entire database. Nevertheless, if the database contains millions of records, then using this strategy is obviously inefficient. In this context, more complex algorithms, such as binary search or hash-based search, have the potential to significantly improve performance. They cleverly minimize the number of operations that are required, which results in a reduction in the load placed on the server and an improvement in response time. The organization of data is a further essential component of backend programming. Efficient sorting algorithms are required for a variety of tasks, including sorting user records by name, sorting transactions by date, and sorting products by price. In comparison to algorithms that are more straightforward, such as BubbleSort and InsertionSort, more complex sorting procedures, such as QuickSort, MergeSort, or HeapSort, are able to effectively manage large datasets, providing sorted data in a shorter amount of time and making use of fewer resources.

In addition, algorithms assist in the resolution of difficult problems that arise in backend programming. Take, for instance, the issue of load balancing, which refers to the process of distributing network traffic across multiple servers in order to prevent any one server from becoming overloaded. In this situation, algorithms can be of assistance by defining how the load should be distributed. This can be done using a straightforward round-robin approach, a more complicated least-connections approach, or even a predictive approach that is based on machine learning. In concurrent processing, in which multiple tasks must be carried out in synchrony with one another, algorithms are used to manage the sequencing, synchronization, and carrying out of these tasks. They make certain that tasks are completed in the correct order, that resources are utilized efficiently, and that tasks do not interfere with each other, which enables the backend system to operate smoothly and efficiently.

It is impossible to overstate the significance of algorithms in the process of developing backend software. They are the skeletal structure of backend systems that are effective, dependable, and scalable. They simplify difficult problems into workable tasks and clear the way for developers to build backend solutions that are both robust and optimized. As we continue to explore deeper

into the complexities of backend development in C++, gaining an understanding of algorithms and finding ways to leverage their power will be a constant companion for us.

Algorithm Design

Designing an algorithm is a systematic process. It starts with understanding the problem at hand, identifying the inputs and outputs, and planning the steps required to transform the input into the desired output. It involves careful consideration of edge cases, potential pitfalls, and trade-offs between time and space efficiency.

Problem Definition

The first step in designing an algorithm is understanding the problem that needs to be solved. This includes identifying the inputs, outputs, and any constraints or requirements that need to be fulfilled. A clear problem definition is essential for designing a successful algorithm.

For instance, let us consider a backend problem - developing a recommendation algorithm for a book-selling platform. The algorithm's role would be to analyze a user's purchase history and recommend books that they might be interested in.

Identifying Inputs and Outputs

The next step is to identify the inputs and outputs of the algorithm. In the case of the book recommendation system, the input would be the user's purchase history, and the output would be a list of recommended books. Identifying the inputs and outputs is crucial because it helps define what the algorithm needs to do.

Defining the Process

Next, we define the steps or the process that would convert the input to the desired output. This could be a simple or complex process based on the problem. For our recommendation algorithm, we might decide to recommend books that fall into the same genre as the majority of the user's past purchases. We could further refine the recommendations by considering factors like the authors of previously purchased books and the average ratings of these books.

Algorithm Pseudocode

After defining the process, the next step is to write pseudocode or create a flowchart that outlines the steps of the algorithm. This helps to visualize the process and makes it easier to identify potential issues or areas for optimization. The pseudocode for the book recommendation system might look something like this:

Given below is an example of how the pseudocode for the recommendation algorithm might look:

1. Initialize an empty list of recommendations.
2. Get the list of books purchased by the user.
3. Determine the most common genre in the purchased books.
4. Filter the platform's book database to only include books of that genre.
5. Rank the filtered books based on their average ratings.

6. Add the top-rated books to the recommendations list.
7. Return the recommendations list.

Optimization

Finally, consider how the algorithm can be optimized. This might involve reducing redundant steps, using more efficient data structures, or leveraging parallel processing.

For the book recommendation algorithm, optimization might involve caching the user's purchase history so that it doesn't have to be retrieved from the database each time a recommendation is made, or it could involve using a priority queue to efficiently select the top-rated books.

Testing

The final step in designing an algorithm is to test it thoroughly. This involves running the algorithm with a variety of inputs to ensure that it produces the expected outputs. You should test the algorithm with normal inputs, edge cases, and even invalid inputs to ensure that it can handle all possible scenarios. For the book recommendation system, you might test the algorithm with users who have a clear favorite genre, users who have purchased books from many different genres, and users who have not purchased any books yet.

To sum up, designing an algorithm is a systematic and iterative process that involves understanding the problem, defining the process, writing pseudocode, optimizing the algorithm, and thoroughly testing it. A well-designed algorithm can make the difference between an efficient, reliable backend system and one that is slow and error-prone. Therefore, spending time on algorithm design is a worthy investment for any backend developer.

Sorting Algorithms

Sorting is one of the most fundamental operations in computer science, and understanding different sorting algorithms is essential for backend developers. Let us look into Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort, and see how they function.

Bubble Sort

Bubble Sort is a simple algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

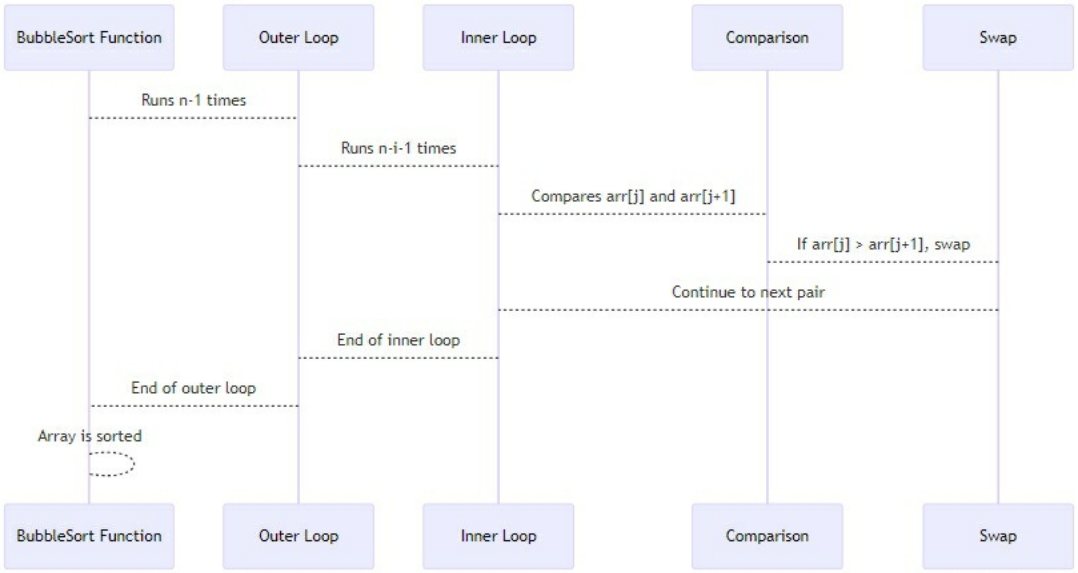


Fig 3.1 Bubble Sort Algorithm

Following is the sample program of using bubble sort algorithm:

```
void bubbleSort(int arr[], int n) {
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
```

```
        }
    }
  }
}
```

The given below walkthrough is a step-by-step interpretation of how the above code snippet works:

- The function bubbleSort takes two arguments: an array arr[] of integers and an integer n representing the size of the array.
- The outer loop runs n-1 times, where n is the size of the array. This is because in each pass through the array, the largest element is "bubbled" to its correct position at the end of the array. So, after n-1 passes, the smallest element will also be in its correct position, and the array will be sorted.
- The inner loop runs n-i-1 times. This is because with each pass of the outer loop, the largest element among the first (n-i) elements is placed at the end of this section, and so it does not need to be considered in the next pass.
- Inside the inner loop, the code checks if the current element arr[j] is greater than the next element arr[j+1]. If it is, it means these elements are in the wrong order (since we're aiming for ascending order), and they need to be swapped.
- The swapping is done using a temporary variable temp. The value of arr[j] is stored in temp, then arr[j] is set to the value of arr[j+1], and finally arr[j+1] is set to the value stored in temp.

This process continues until the entire array is sorted in ascending order. The reason this algorithm is called "Bubble Sort" is because in each iteration the largest unsorted element "bubbles up" to its correct position.

Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as QuickSort, HeapSort, or MergeSort.

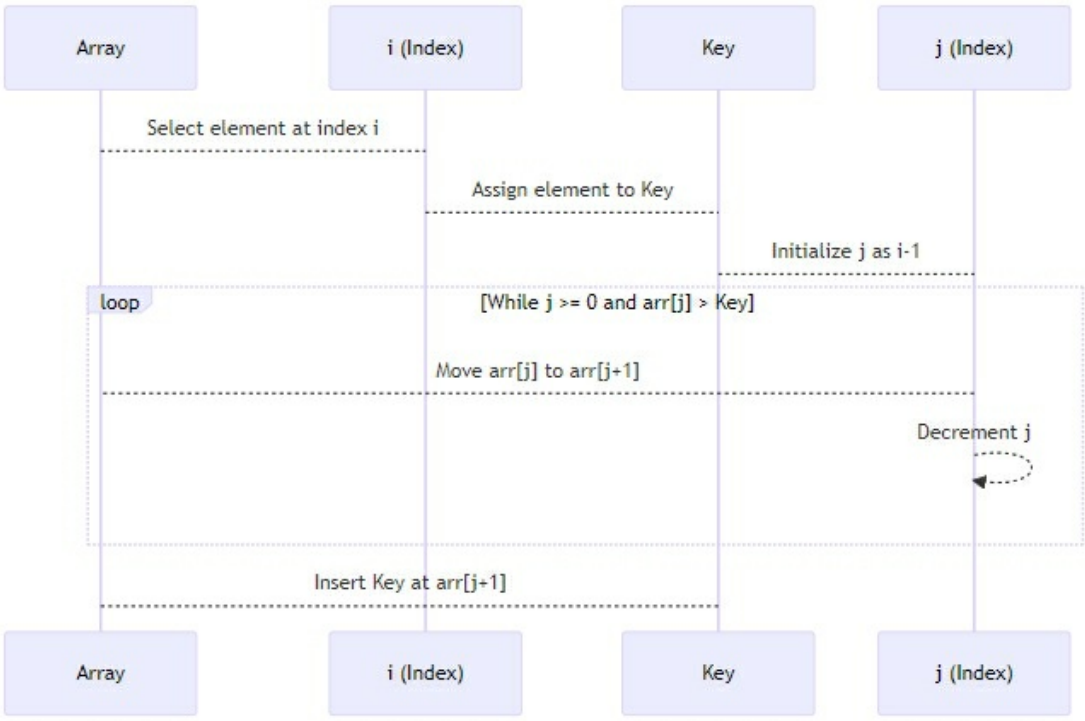


Fig 3.2 Insertion Sort Algorithm

Following is the sample program of using insertion sorting algorithm:

```
void insertionSort(int arr[], int n) {
   for(int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;
       // Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position
      while(j >= 0 && arr[j] > key) {
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1] = key;
   }
```

```
}
```

Following is a step-by-step interpretation of how it works:

- The function insertionSort takes two parameters: an array of integers arr[] and the size of the array n.
- The outer loop runs from the second element to the last element of the array. The variable i is the index of the element to be inserted into the sorted sequence arr[0..i-1].
- The key variable holds the value of the current element to be compared with elements to its left.
- The inner loop while(j >= 0 && arr[j] > key) shifts each element of the array arr[0..i-1], that is greater than key, to one position ahead of its current position.
- The shifting process continues until the correct position for key is found, or it reaches the beginning of the array. The key is then placed in its correct position in the sorted sequence.
- The outer loop then moves to the next element and the process is repeated until the entire array is sorted.

This algorithm is called Insertion Sort because it works by inserting each item into its proper place to form the sorted list. It's an in-place, stable sorting algorithm that is efficient for smaller datasets, but less efficient for larger datasets. Though not suitable for large data sets, the Insertion Sort provides several advantages: simple implementation, efficient for (quite) small data sets, adaptive (i.e., efficient for data sets that are already substantially sorted), and stable (i.e., does not change the relative order of elements with equal keys).

Quick Sort

Quick Sort is a highly efficient sorting algorithm and is based on the divide-and-conquer technique. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

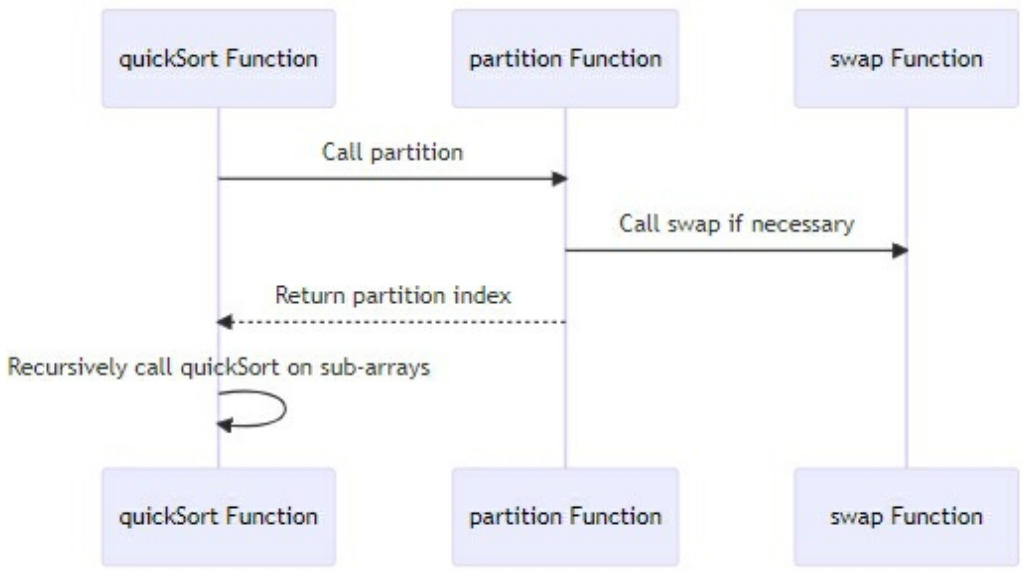


Fig 3.3 Quick Sort Algorithm

Given below is the code sample of Quick Sort algorithm:

```
void quickSort(int arr[], int low, int high) {
  if (low < high) {
     // pi is partitioning index, arr[pi] is now at right place
     int pi = partition(arr, low, high);
     // Separately sort elements before partition and after partition
     quickSort(arr, low, pi - 1);
     quickSort(arr, pi + 1, high);
  }
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // pivot
   int i = (low - 1); // Index of smaller element
   for (int j = low; j <= high - 1; j++) {
     // If current element is smaller than the pivot
```

```
    if (arr[j] < pivot) {
        i++; // increment index of smaller element
        swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[high]);
  return (i + 1);
}
```

The given below walkthrough is a step-by-step interpretation of the code:

- The quickSort function is the main function that implements QuickSort. It takes three parameters: an array arr[], and two integers low and high, which represent the starting and ending indices of the portion of the array to be sorted.
- The quickSort function first checks if low is less than high. This is to ensure that there are at least two elements to sort. If low is not less than high, the function simply returns, as there is nothing to sort.
- If low is less than high, the function proceeds to partition the array. The partitioning is done by the partition function, which takes the same parameters as quickSort and returns an integer pi, the partitioning index. The partition function ensures that the element at arr[pi] is in its correct place in the sorted array.
- After partitioning, the quickSort function recursively sorts the elements before pi and after pi.
- The partition function works by choosing a pivot element from the array (in this case, the element at arr[high]) and partitioning the other elements into two sections: those less than the pivot and those greater than the pivot. It keeps track of the index of the smaller elements with i.
- The partition function iterates over the array with a for loop. If the current element arr[j] is less than the pivot, it increments i and swaps arr[i] with arr[j].
- After all elements have been processed, the partition function swaps the pivot element with the element at arr[i + 1], effectively placing the pivot in its correct position in the sorted array. It then returns i + 1, the index of the pivot.
- The swap function is a standard function to swap the values of two variables. It's not shown in the provided code, but it's a common utility function.

Quick Sort is not a stable sort, meaning the relative order of equal sort items is not preserved. Although its worst-case time complexity is O(n^2), QuickSort is one of the fastest algorithms for sorting arrays, and is widely used.

Merge Sort

Merge Sort is also based on the divide-and-conquer technique. It works by dividing the unsorted list into N sublists, each containing one element (a list of one element is considered sorted), and then repeatedly merging sublists to produce new sorted sublists until there is only one sublist remaining.

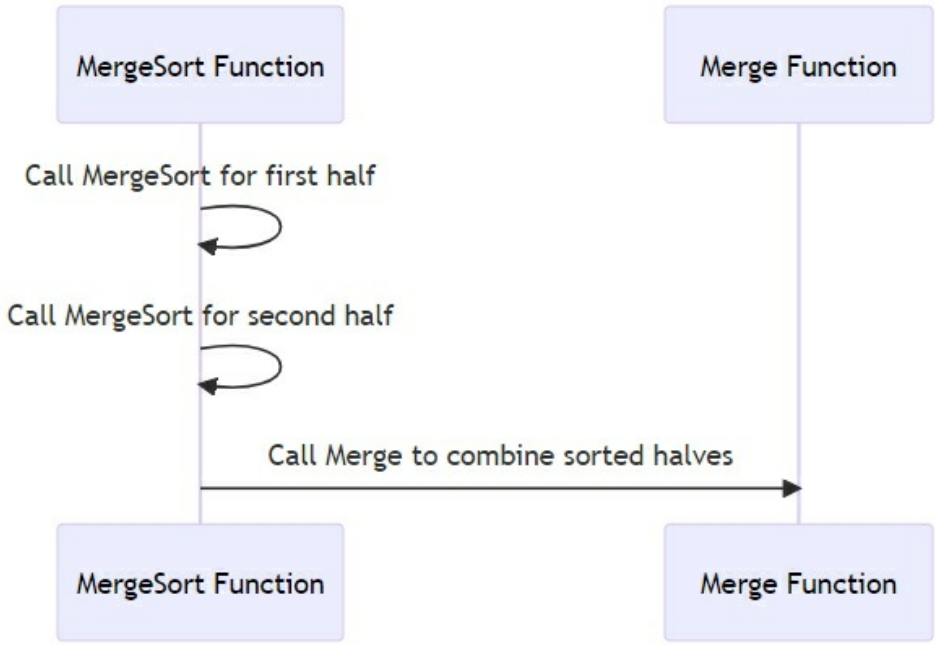


Fig 3.3 Merge Sort Algorithm

Given below is the code snippet of Merge Sort algorithm:

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
     // Same as (l+r)/2, but avoids overflow for large l and h
     int m = l+(r-l)/2;
     // Sort first and second halves
     mergeSort(arr, l, m);
     mergeSort(arr, m+1, r);
     merge(arr, l, m, r);
```

```
  }
}
void merge(int arr[], int l, int m, int r) {
  int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
  /* create temp arrays */
  int L[n1], R[n2];
   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];
   /* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
          arr[k] = L[i];
          i++;
      } else {
          arr[k] = R[j];
          j++;
```

```
        }
        k++;
    }
    /* Copy the remaining elements of L[], if there are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Merge Sort is a stable sort, and is more efficient than Bubble Sort and Insertion Sort, with a time complexity of O(n log n). However, it has a space complexity of O(n) due to the use of additional space for the temporary arrays.

Understanding these sorting algorithms and their trade-offs will help you make appropriate choices in your backend development tasks, ensuring efficiency and performance.

Searching Algorithms

Searching algorithms are fundamental to computer science, used in a multitude of applications, from simple data retrieval to complex problem-solving. Among the various searching algorithms, two stand out due to their simplicity and efficiency: Linear Search and Binary Search.

Linear Search

Linear Search, as the name suggests, involves searching a list in a linear or sequential manner. It is the most basic type of search algorithm and works by iterating over all elements in a list, one by one, until the desired element is found or all elements have been checked. Despite its simplicity, Linear Search is incredibly versatile, capable of handling both sorted and unsorted lists, and even lists with duplicate elements.

The process of Linear Search can be summarized as follows: Starting from the first element of the list, compare each element with the target element. If the current element matches the target, the search ends, and the position of the element is returned. If the target element is not found after checking all elements, the search concludes that the element is not present in the list.

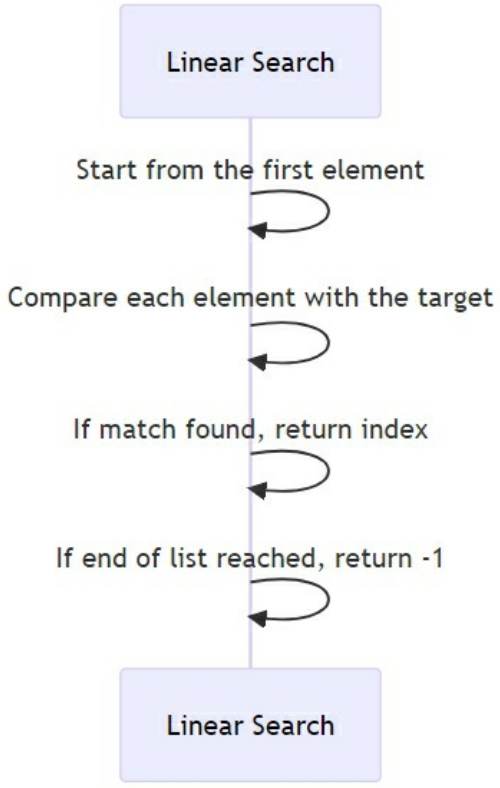


Fig 3.4 Linear Search Algorithm

Following is the code sample of Linear Search algorithm:

```
int linearSearch(int arr[], int n, int x) {
```

```
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i; // Element found, return index        }
    }
    return -1; // Element not found
}
```

In the context of backend development, linear search can be useful for small data sets or for unsorted data. However, as data sets grow in size, linear search can become increasingly inefficient due to its worst-case and average time complexity of O(n).

Binary Search

Binary Search is a highly efficient searching algorithm that significantly reduces the time complexity, especially for large, sorted data sets. Unlike linear search, which scans each element in the list sequentially, binary search employs a divide-and-conquer strategy, effectively reducing the search space by half after each comparison.

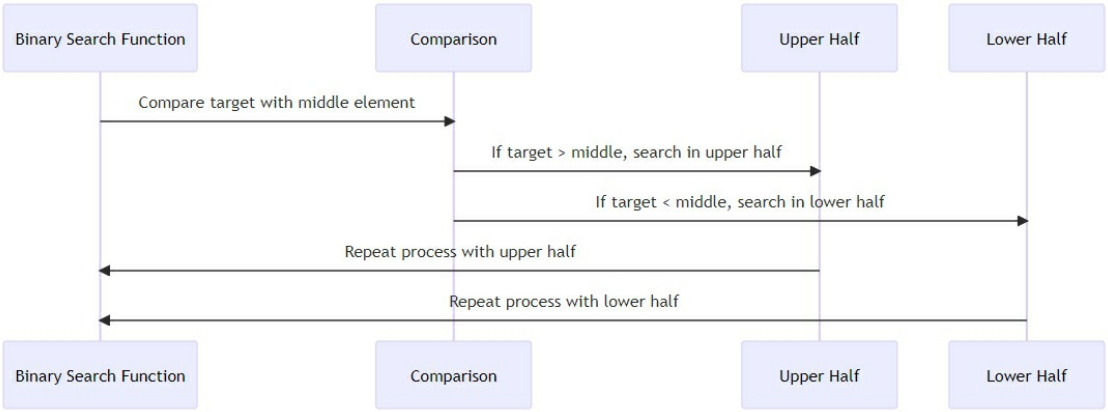


Fig 3.5 Binary Search Algorithm

The binary search algorithm operates by comparing the target value to the middle element of a sorted list. The key idea behind binary search is that if the target value is equal to the middle element, then the search is successful, and the position of the middle element is returned. If the target value is less than the middle element, it implies that the target can only lie in the lower (left) half of the list. Conversely, if the target value is greater than the middle element, the target can only be in the upper (right) half of the list. In each subsequent step, the algorithm repeats the process on the narrowed half, either to the left or right, depending on the result of the comparison, until the target is found or all possibilities have been eliminated. This process of dividing the list continues until the search space is reduced to a single element.

Following is the code sample of Binary Search algorithm:

```
int binarySearch(int arr[], int l, int r, int x) {
  if (r >= l) {
     int mid = l + (r - l) / 2;
     // If the element is present at the middle
     if (arr[mid] == x)
        return mid;
      // If element is smaller than mid, then it can only be present in left subarray
     if (arr[mid] > x)
        return binarySearch(arr, l, mid - 1, x);
     // Else the element can only be present in right subarray
     return binarySearch(arr, mid + 1, r, x);
  }
   // We reach here when element is not present in array
  return -1;
}
```

Binary Search is significantly faster than Linear Search for larger, sorted data sets with a worst-case and average time complexity of O(log n). However, it requires that the data set is sorted beforehand, which may not always be the case.

In backend development, Binary Search can be useful in a variety of contexts where you need to quickly locate a specific item from a large, sorted data set. For example, if you need to quickly find a user from a list of users sorted by user ID, Binary Search would be an excellent choice.
Knowing how these algorithms function and their time complexity is crucial for backend development, as the efficiency of these operations can have a significant impact on the performance of your applications. The choice of algorithm will depend on the specific requirements of the task, including the size of the data set and whether or not it's sorted.

Graph Algorithms

Graph algorithms are fundamental to various operations in backend development, especially in the realms of data analytics, networking, and route/path finding. Let us discuss three important graph algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's Algorithm.

Depth-First Search (DFS)

Depth-First Search (DFS) is a fundamental algorithm in computer science for traversing or searching through a graph. As the name suggests, it dives deeper into the graph, exploring one branch as far as possible until there are no more new nodes along that path. Once it reaches the end, it backtracks and starts exploring the next available branch, repeating the process until every vertex has been visited.

DFS uses a stack data structure to remember which vertices to visit next. The stack follows the Last-In-First-Out (LIFO) principle, meaning that the most recently discovered vertex that hasn't been fully explored is the one to be explored next. This approach gives DFS its characteristic of going as deep as possible along each branch before backtracking.

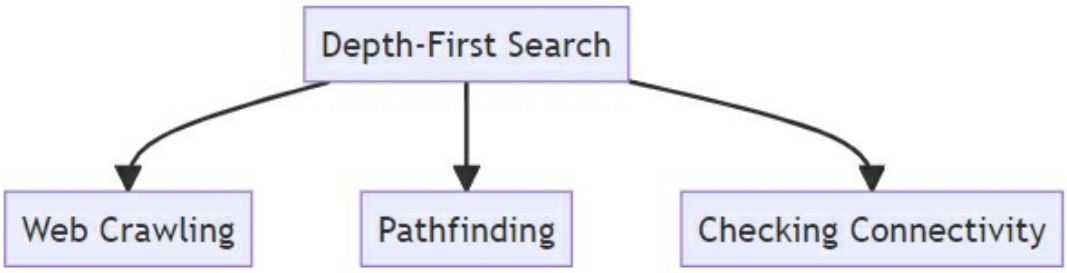


Fig 3.6 Applications of Depth-First Search Algorithm

Given below is an implementation of DFS in C++:

```
#include <list>
#include <iostream>
class Graph {
  int V;
  std::list<int> *adj;
public:
  Graph(int V);
```

```
  void addEdge(int v, int w);
  void DFS(int v);
};
Graph::Graph(int V) {
  this->V = V;
  adj = new std::list<int>[V];
}
void Graph::addEdge(int v, int w) {
  adj[v].push_back(w);
}
void Graph::DFS(int v) {
  bool *visited = new bool[V];
   for (int i = 0; i < V; i++)
      visited[i] = false;
  std::list<int> stack;
  visited[v] = true;
  stack.push_back(v);
  while (!stack.empty()) {
      v = stack.back();
      std::cout << v << " ";
      stack.pop_back();
      std::list<int>::iterator i;
      for (i = adj[v].begin(); i != adj[v].end(); ++i)
          if (!visited[*i]) {
              stack.push_back(*i);
```

```
            visited[*i] = true;
        }
    }
}
```

Despite its simplicity, DFS is a powerful tool in algorithm design and problem-solving. However, it's important to note that DFS isn't always the most efficient or suitable algorithm for every case. Its efficiency and suitability depend on the specific characteristics and requirements of the problem at hand.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores all vertices of a graph in breadth-first order, meaning it explores all of the neighbor vertices at the present depth before moving on to vertices at the next depth level. This is in contrast to Depth-First Search (DFS), which explores as far as possible along each branch before backtracking.

BFS operates by maintaining a queue of vertices to explore. It starts by enqueuing a starting vertex, then enters a loop that continues until there are no more vertices to explore. In each iteration of the loop, BFS dequeues a vertex and examines it. If the vertex is the goal, BFS stops. Otherwise, it enqueues all neighbors of the vertex that have not yet been discovered. This process continues until all vertices have been explored or the goal vertex has been found.

The queue data structure is a crucial component of BFS. It follows the First-In-First-Out (FIFO) principle, meaning that the order in which vertices are enqueued is the order in which they will be dequeued. This ensures that BFS explores vertices in breadth-first order.

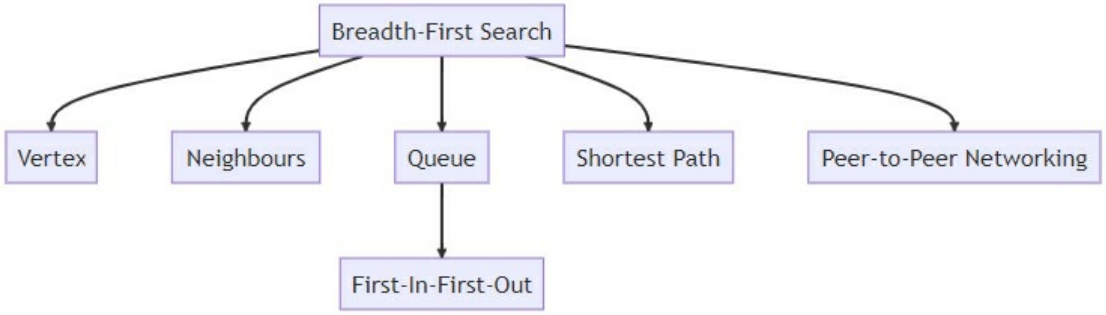


Fig 3.7 Breadth-First Search Algorithm

Given below is an implementation of BFS in C++:

```
#include <list>
#include <iostream>
```

```
class Graph {
   int V;
   std::list<int> *adj;
public:
   Graph(int V);
   void addEdge(int v, int w);
   void BFS(int s);
};
Graph::Graph(int V) {
   this->V = V;
   adj = new std::list<int>[V];
}
void Graph::addEdge(int v, int w) {
   adj[v].push_back(w);
}
void Graph::BFS(int s) {
   bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
       visited[i] = false;
   std::list<int> queue;
   visited[s] = true;
   queue.push_back(s);
   while(!queue.empty()) {
       s = queue.front();
       std::cout << s << " ";
```

```
      queue.pop_front();
      for (auto i = adj[s].begin(); i != adj[s].end(); ++i) {
         if (!visited[*i]) {
            queue.push_back(*i);
            visited[*i] = true;
         }
      }
   }
}
```

BFS is a versatile and fundamental graph traversal algorithm that is widely used in various fields, including backend development, due to its ability to find the shortest path and its efficient handling of large datasets.

Dijkstra's Algorithm

Dijkstra's Algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is a powerful tool used in graph theory. It is a greedy algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree.

The algorithm works by maintaining a set of unvisited nodes and continuously selecting the node with the smallest tentative distance from the start node, then visiting all its unvisited neighbors. The tentative distance to a node is updated only if a shorter path to this node is found. The process continues until all nodes have been visited, at which point the algorithm is guaranteed to have found the shortest possible paths from the start node to all other nodes.

Dijkstra's algorithm is widely used in network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). These protocols are used to determine the best route for data packets to take in order to reach a particular destination.

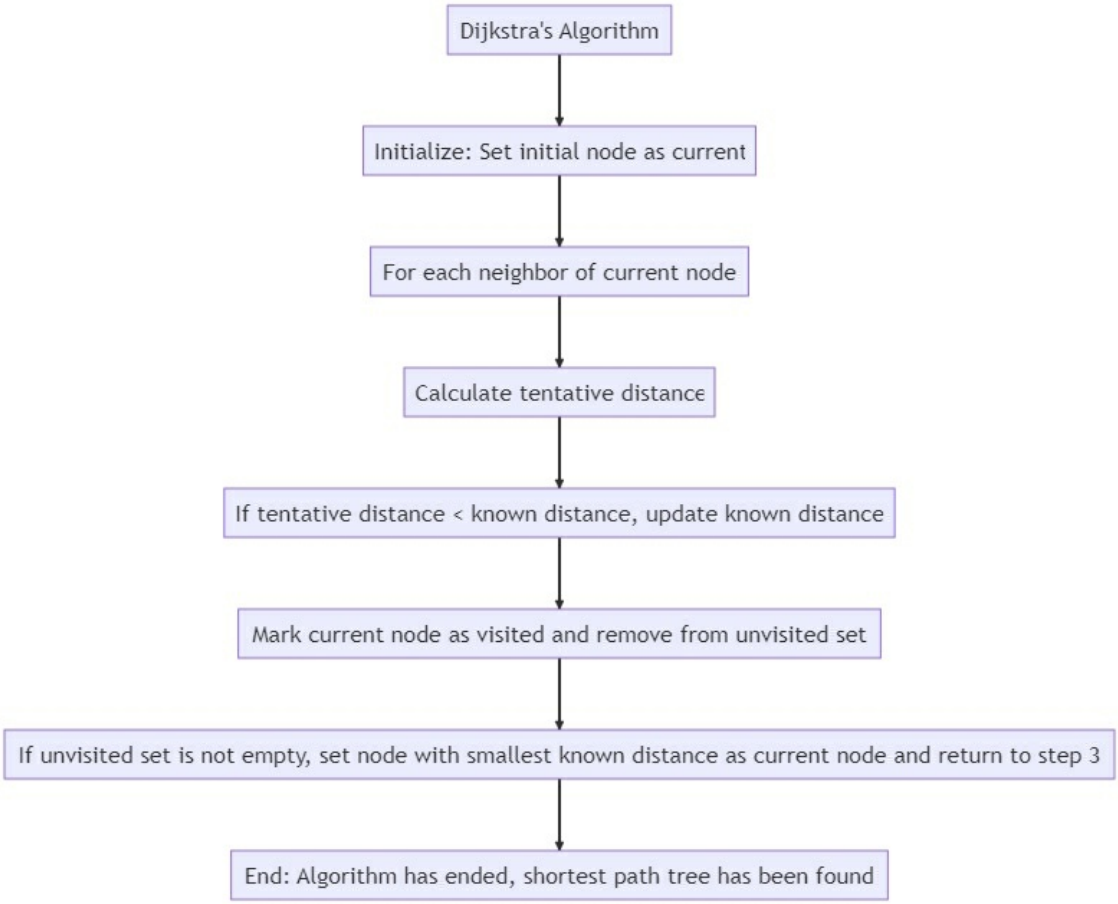


Fig 3.8 Dijkstra Algorithm

Given below is an implementation of Dijkstra Algorithm in C++:

```
#include <iostream>
#include <limits>
#include <vector>
using namespace std;
// Function that implements Dijkstra's algorithm
void dijkstra(vector<vector<int> > graph, int src) {
  int V = graph.size();
   vector<int> dist(V, INT_MAX); // Initialize distance values
  dist[src] = 0;
```

```
  vector<bool> processed(V, false);
   for (int count = 0; count < V-1; count++) {
      // Pick the minimum distance vertex from the set of vertices not yet
processed
      int min = INT_MAX, min_index;
      for (int v = 0; v < V; v++)
         if (processed[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;
      int u = min_index;
      processed[u] = true;
      for (int v = 0; v < V; v++)
         if (!processed[v] && graph[u][v] && dist[u] != INT_MAX
            && dist[u]+graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
   }
   cout << "Vertex Distance from Source\n";
   for (int i = 0; i < V; i++)
      cout << i << " " << dist[i] << "\n";
}
```

Moreover, Dijkstra's algorithm serves as a subroutine in many other graph algorithms to solve complex problems such as the Travelling Salesman Problem, making it a fundamental tool in computer science.

Hash Algorithms

Hashing is a technique used to convert a range of key values into a range of index values. Hashing makes data retrieval extremely efficient, being nearly constant time complexity, O(1). In this case, we shall focus on the most basic hash function, collision resolution techniques, and some applications in backend development.

Simple Hash Function

A hash function maps a big number or string to a small integer that can be used as an index in the hash table. The idea of hashing is to distribute entries (key-value pairs) across an array of buckets.

A simple hash function could be to divide the key by the length of the array (N), and take the remainder as the hash value (key % N).

```
unsigned int simpleHashFunction(string key, int N) {
  unsigned int hashValue = 0;
   for (char c : key) {
     hashValue += c;
  }
  return hashValue % N;
}
```

In the example above, we are simply converting each character in the key to its ASCII value, adding them up, and then taking the modulo of N (table size) to get the hash value.

Collision Resolution

Collisions occur when two keys hash to the same index. There are several techniques to resolve collisions:

- Separate Chaining: Each bucket contains a list of all entries that hash to the same index.
- Open Addressing: When a collision occurs, we look for another free bucket among the array slots by following a sequence until we find an empty bucket.

```
// Using separate chaining
class MyHash {
  int BUCKET;
```

```
  list<int> *table;
public:
  MyHash(int b) {
     BUCKET = b;
     table = new list<int>[BUCKET];
  }
  void insert(int key) {
     int i = hash(key);
     table[i].push_back(key);
  }
  bool search(int key) {
     int i = hash(key);
     for (auto x : table[i])
       if (x == key)
          return true;
     return false;
  }
  void remove(int key) {
     int i = hash(key);
     table[i].remove(key);
  }
private:
  int hash(int key) {
     return key % BUCKET;
  }
```

```
};
```

Hash functions are used in various applications such as in database indexing, caches, password storage, and more. For example, when storing passwords, a hash function is used to map a password to a unique hashed value, which is stored in the database. This means that even if the database is compromised, the attacker will not have access to the actual passwords, only the hashed values.

Hash tables are used in caching applications where quick access to data is required. The key contains the query and the value contains the result of the query. When a query is made, the cache is checked to see if the result is already available (cache hit) to avoid re-computation or database retrieval. Understanding and implementing hash functions and collision resolution techniques can greatly optimize the performance and security of your backend application.

Recursive Algorithms

Overview

Algorithms are the beating heart of any application, including backend programming. Recursive and iterative algorithms are ways to solve problems that need repetitive computations. We will go through their concepts and then explore some examples.

Recursion involves a function calling itself to solve a smaller version of the original problem. The approach can be highly effective for certain tasks, such as traversing tree-like data structures. A recursive algorithm will typically have a base case to stop the recursion and a recursive case that breaks down the large problem into smaller subproblems.

Sample Program: Fibonacci Sequence

The Fibonacci sequence is a classic example of a recursive algorithm in programming. The sequence starts with 0 and 1, and every number thereafter is the sum of the previous two.

```
int fibonacci_recursive(int n) {
  if (n <= 1)
     return n;
  else
     return(fibonacci_recursive(n-1) + fibonacci_recursive(n-2));
}
```

In the code above, the base cases are defined for n=0 and n=1. For other values, the function calls itself with n-1 and n-2, thereby breaking down the problem into smaller subproblems.

Iterative Algorithms

Overview

Iteration involves looping through a set of instructions until a specific condition is met. Compared to recursion, iteration tends to be more efficient in terms of memory and performance as it does not involve the overhead of repeated function calls.

Sample Program: Fibonacci Sequence (Iterative)

The Fibonacci sequence can also be calculated iteratively.

```
int fibonacci_iterative(int n) {
  if (n <= 1)
     return n;
  else {
     int fib = 1;
     int prevFib = 0;
     for(int i = 2; i <= n; ++i) {
        int temp = fib;
        fib += prevFib;
        prevFib = temp;
     }
     return fib;
  }
}
```

In the iterative version of the Fibonacci sequence, we use a loop to calculate the Fibonacci number at the nth position. It's more efficient as we are not recalculating the same Fibonacci numbers as we do in the recursive approach.

In backend development, whether to use recursion or iteration will depend on the specific problem at hand. Recursion is often more intuitive and easier to implement, especially for problems involving tree-like data structures. However, it can be more memory-intensive. Iteration, on the other hand, is typically more efficient but might be less straightforward to

implement for certain types of problems. Understanding both recursive and iterative algorithms is crucial for efficient backend programming.

Dynamic Programming Algorithms

Dynamic Programming, also known by its common abbreviation DP, is a powerful computational technique that is used in the process of problem-solving by dividing the primary problem into a number of simpler and more manageable subproblems. This strategy is especially useful for solving optimization problems, which require one to select the most advantageous course of action from among a number of potential alternatives.

The idea of "Memoization" is the cornerstone of Dynamic Programming and serves as the methodology's guiding principle. The process of storing the solutions to each subproblem in a methodical manner in order to ensure that each problem is only ever solved once is referred to by the term "solution caching." Instead of recalculating the solution whenever a particular subproblem needs to be solved again, the answer that was previously stored is used instead. This results in a significant reduction in the amount of computational time and resources required, which makes DP an effective method for solving complex problems.

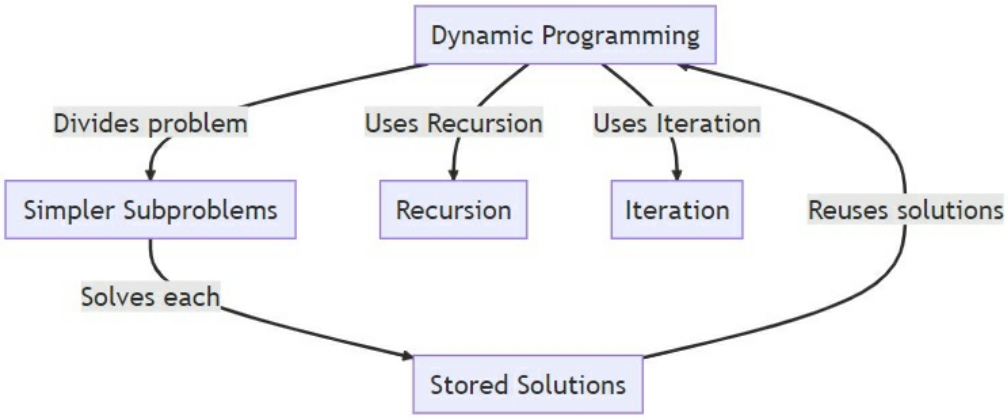


Fig 3.9 Approaches of Dynamic Programming

Recursion and iteration are the two primary approaches that can be utilized when putting into practice dynamic programming. In recursion, a problem is solved by first breaking it down into smaller subproblems, which in turn are solved in a manner that is analogous to the original solution. Iteration is a method for solving problems step by step, typically beginning with the problem's simplest subproblem and working one's way up to the primary issue, while saving one's progress at each stage. The effective archiving and recycling of used solutions is essential to the DP process. By saving the solutions to each individual subproblem, we ensure that we won't have to start from square one in the event that the main problem rears its head again. Instead, we can make direct use of the solution that was computed in the past. The principle of reusing solutions not only cuts down on the amount of time needed for computation, but it also makes the process of problem-solving more effective and manageable.

Let us explore a practical example to illustrate how Dynamic Programming can be utilized to optimize common algorithmic problems.

Sample Program: Fibonacci Sequence (Dynamic

Programming)

The Fibonacci sequence is a series of numbers where a number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence goes: 0, 1, 1, 2, 3, 5, 8, 13, 21, and so forth.

Following is a sample program that calculates the Fibonacci sequence using Dynamic Programming:

```
int fibonacci_DP(int n) {
    vector<int> fib(n + 1, 0); // Initialize a vector to store Fibonacci numbers
    fib[0] = 0; // The first number in the Fibonacci sequence is 0
    fib[1] = 1; // The second number in the Fibonacci sequence is 1
    // Calculate each Fibonacci number starting from 2
    for(int i = 2; i <= n; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2]; // Each number is the sum of the two preceding ones
    }
    return fib[n]; // Return the nth Fibonacci number
}
```

In this code, we initialize a vector fib with a size of n + 1 to store the Fibonacci numbers up to n. The vector is filled with zeros initially. We then manually set the first two numbers of the Fibonacci sequence, fib[0] and fib[1], to 0 and 1 respectively. The core of the program lies in the for loop, where we calculate each Fibonacci number starting from the third one (i = 2). For each i from 2 to n, we calculate fib[i] as the sum of the two preceding numbers, fib[i - 1] and fib[i - 2]. This way, we ensure that each Fibonacci number is calculated only once, leveraging the principle of Dynamic Programming.

Finally, we return the nth Fibonacci number, fib[n], as the result of the function. This approach significantly reduces the computational time and resources compared to traditional recursive methods, demonstrating the power and efficiency of Dynamic Programming.

Sample Program: Coin Change Problem

A more complex problem solvable by Dynamic Programming is the coin change problem. This problem is a classic example of an optimization problem where the goal is to determine the minimum number of coins required to make a specific amount of change.

```
int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, amount + 1);
   dp[0] = 0;
    for (int i = 1; i <= amount; i++) {
       for (int j = 0; j < coins.size(); j++) {
           if (coins[j] <= i) {
               dp[i] = min(dp[i], dp[i - coins[j]] + 1);
           }
       }
   }
    return dp[amount] > amount ? -1 : dp[amount];
}
```

The provided code snippet is a function named coinChange that implements a solution to this problem. This function accepts two parameters: a vector of integers representing the available coin denominations (coins), and an integer representing the target amount of change (amount).

The function begins by initializing a vector dp of size amount + 1, with each element set to amount + 1. This vector will store the minimum number of coins needed to make each possible amount of change up to amount. The value at index 0 is set to 0, as no coins are needed to make 0 change.

The function then enters a nested loop. The outer loop iterates over each possible amount of change from 1 to amount. The inner loop iterates over each coin denomination. For each coin that is less than or equal to the current amount, the function updates the corresponding dp value to the minimum of its current value or the value at dp[i - coins[j]] + 1, which represents the minimum number of coins needed to make the current amount of change using the current coin denomination.

Finally, the function checks if the value at dp[amount] is greater than amount, which would indicate that it's not possible to make the exact change with the given coin denominations. If this is the case, the function returns -1. Otherwise, it returns the value at dp[amount], which represents the minimum number of coins needed to make the exact change.

The application of dynamic programming is not restricted to merely the resolution of mathematical issues. Additionally, it has a significant impact on the programming of the backend. Dynamic programming offers an approach that is both effective and scalable, and it can

be utilized for a variety of purposes, including the optimization of database queries, the management of resources, and the resolution of complex scheduling issues. Dynamic programming has the potential to significantly improve the performance and scalability of backend systems by reducing complex problems to smaller, more manageable subproblems and reusing previously developed solutions.

Summary

In this chapter, we explored the importance and practical application of various algorithms in the context of backend programming. We started by understanding the role of algorithms in solving computational problems. They are vital for efficient data processing, resource allocation, and task management in backend systems. Algorithms help in optimizing the backend for better performance, scalability, and reliability, especially in high-traffic applications.

We went on to explore the specifics of designing algorithms, demonstrating the design process through sorting and searching algorithms such as Bubble Sort, Quick Sort, Merge Sort, and Binary Search. We also explored into more complex areas like Graph Algorithms and Hash Algorithms, which are key to solving complex backend problems involving data storage, retrieval, and routing. Each algorithm was detailed with its operating principle and relevant examples to help you grasp their implementation in the backend setting.

Lastly, we discussed recursive and iterative algorithms, and the relatively advanced concept of Dynamic Programming. Recursion and iteration are fundamental problem-solving techniques, while Dynamic Programming is a higher-level approach that optimizes problem-solving by breaking down large problems into simpler, manageable subproblems. We illustrated these concepts using examples such as Fibonacci sequence and the Coin Change problem. These techniques and methodologies are critical in backend programming for tasks such as optimizing database queries and managing complex computations. The chapter highlighted the integral role of algorithms in shaping the backbone of software applications and services.

CHAPTER 4: MASTERING VERSION CONTROL - GIT AND GITHUB

Version Control and Repo Hosting: Overview

Version control systems and repository hosting services are indispensable tools in the intricate world of software development, particularly for backend programming. They not only help manage changes and keep the project organized, but they also make collaboration easier, foster transparency, and increase the overall quality of the codebase. These benefits are in addition to the fact that they help manage changes and keep the project organized.

Developers are able to navigate and manage the progression of a project by utilizing version control systems such as Git, which offer a comprehensive history of the modifications that have been made to the codebase. They make it possible for multiple developers to work on the same codebase simultaneously without overwriting each other's changes, which is a feature that is particularly useful in backend development, where teams frequently collaborate on developing complex systems. Version control also provides the ability to generate multiple versions of a project, which can be helpful for testing new features without affecting the functionality of the primary application. In the event that something goes wrong with the code, developers are able to easily roll back to an earlier version of the code.

Additionally, version control systems introduce accountability and traceability by assigning each change to a particular developer. This allows for the tracking of who made which changes. Because of this, it is simple to determine who was responsible for introducing a particular piece of code or a bug. This functionality is especially useful in backend development, which has a greater number of interconnected systems and services than frontend development does. As a result, determining the source of an issue can be more difficult.

Hosting services for repositories, such as GitHub and Bitbucket, serve as a centralized location for the storing and sharing of computer code. They do this by utilizing the power that version control systems provide and enhancing those systems with additional features such as issue tracking, pull requests, code reviews, and documentation. They make it simpler for teams to collaborate with one another, track progress, discuss changes, and provide feedback, which ultimately results in higher-quality code.

These features offer an incredible amount of value when considered in the context of backend development. Backend code is frequently difficult to understand, and even small alterations can have far-reaching effects. The repository hosting services help ensure that the code is not only functional, but also maintainable, efficient, and well-documented. This is achieved through the facilitation of code reviews and discussions surrounding changes.

In a nutshell, the most important tools for backend development are version control systems and hosting services for repositories. They make the development process more efficient, improve the ability to collaborate, raise the level of code quality, and ultimately contribute to the development of backend systems that are dependable and robust.

Exploring Git

Git is a widely used version control system that helps developers manage and track changes to their code. It is especially useful when working in a team environment, as it allows multiple people to work on a project without overwriting each other's changes. The give below is a step-by-step walkthrough to setting up Git in your C++ environment:

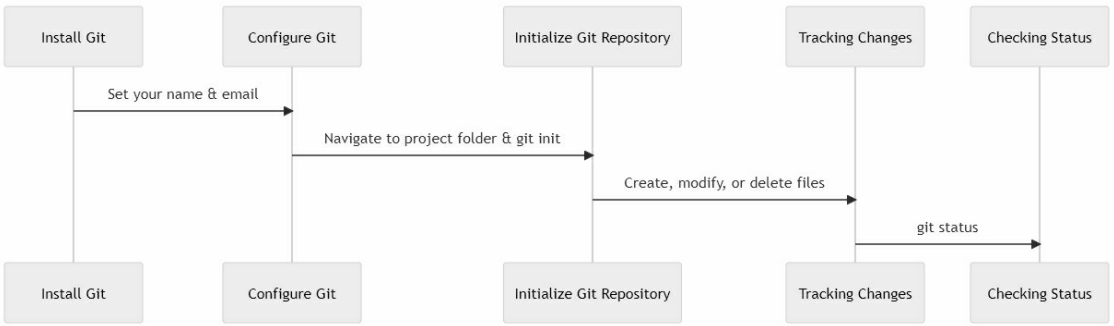


Fig 4.1 Git Configuration

Install Git

First, you need to install Git on your system. You can download Git from its official website. Choose the version that corresponds to your operating system.

- For Windows, you can download it from https://git-scm.com/download/win
- For MacOS, you can use Homebrew brew install git
- For Linux (Ubuntu), you can use the command sudo apt-get install git

Configure Git

After installing Git, you need to configure it. Open your terminal or command prompt and type in the following commands:

- Set your name: git config --global user.name "Your Name"
- Set your email: git config --global user.email "youremail@myblogapp.com"

These details will be associated with any commits that you create.

Initialize Git Repository

To start using Git, you need to initialize a Git repository in your project directory. Navigate to your C++ project folder using the terminal or command prompt and type git init. This command creates a new subdirectory named .git that contains all the necessary metadata for the new repository.

Tracking Changes

Once the repository is initialized, you can start tracking changes. When you create, modify, or

delete a file, Git can track these changes. Use the command git add . to stage changes and git commit -m "Commit message" to save them in the repository. The commit message should describe what changes you made.

Checking Status

You can check the status of your repository using git status. This command will show you any changes that have been staged, changes that haven't been staged yet, and files that aren't being tracked.

This initial setup is enough to start using Git in your C++ environment. As you work with Git, you will learn more advanced commands and concepts, such as branching, merging, and resolving conflicts. In a team or public project, you will also need to learn about pulling and pushing changes to remote repositories.

Basic Git Commands and Workflow

Continuing from the setup, let us explore into the basic commands and workflow of Git. We will look at it from the perspective of a typical backend development project. In backend development, we work with complex codebases and collaborative teams, so understanding Git's workflow is crucial. Given below are the basic commands:

git clone

When you want to work on an existing repository, you clone it. This downloads a copy of the repository to your local machine. The command is git clone <repository URL>. Replace <repository URL> with the URL of the repository you want to clone.

For instance, if you want to contribute to an open-source backend project hosted on GitHub, you'd start by cloning the repository:

```
git clone https://github.com/username/repository.git
```

This creates a directory in your current folder named repository.

git status

The git status command is used to display the state of the working directory and the staging area. It shows which changes have been staged, which haven't, and which files aren't being tracked by Git. This command helps you understand the state of your project and the changes that have been made.

git add

The git add command is used to stage changes for the next commit. This command tells Git that you want to include updates to certain files in the next commit. To add all changes in the working directory, use git add.

git commit

The git commit command is used to save your changes to the local repository. Each commit is a 'snapshot' of your work that you can revert to if needed. Including a message with your commit is good practice, as it provides context for the changes made. This can be done using git commit -m "Your message here".

git push

The git push command is used to upload local repository content to a remote repository. After committing your changes locally, you can share your commits with the rest of your team or the public by pushing your changes to the remote repository.

git pull

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. This is a convenient way to sync your local repository with the latest changes made in the remote repository.

git branch

The git branch command is used to manage branches in Git. Branches are essentially pointers to different versions of your project files. They are useful when you want to add a new feature or fix a bug, and you don't want to affect the main project. The git branch command lets you create, list, rename, and delete branches.

git checkout

The git checkout command is used to switch between different branches in Git. When you checkout to a branch, Git updates your working directory to match the version of the project files on that branch.

git merge

The git merge command is used to combine changes from one branch into another. This is typically done when you have finished developing a feature or fixing a bug on a separate branch and want to integrate those changes into the main codebase.

Following is a visual representation of a simple Git workflow:

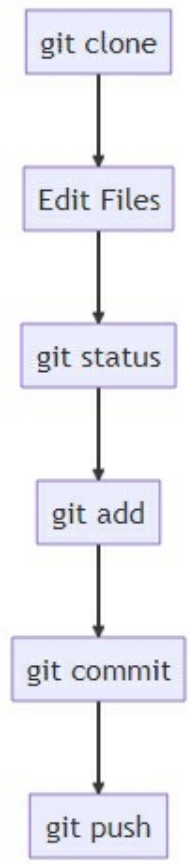


Fig 4.2 Git Workflow

Suppose you're developing a new feature for your backend, such as a new API endpoint. You'd start by cloning the repository, then create a new branch using git branch new-feature. You'd then checkout to the branch with git checkout new-feature and start building your feature. As you code and test, you might commit several times. When the feature is complete and tested, you would merge it back to the main branch with git checkout main and then git merge new-feature. Finally, you'd push your changes with git push, making your feature part of the main codebase.

The specifics can change depending on your team's workflow, but these basic Git commands form the backbone of version control for any backend development project.

Branching and Merging

Branching and merging are two of the most powerful features of Git. They allow developers to work on different features simultaneously without stepping on each other's toes, and then seamlessly combine their changes later.

Branching

A Git branch is essentially a pointer to a specific commit. It represents an independent line of development in a project, which you can use to isolate work on a particular feature or bug fix.

You can create a new branch with the git branch <branch-name> command. For instance, if you were adding a new REST endpoint to your backend, you might create a new branch for that work:

```
git branch add-endpoint
```

To switch to this new branch and start working on it, you would use the git checkout <branch-name> command:

```
git checkout add-endpoint
```

Merging

Merging is a fundamental operation in Git that allows you to bring together the work from different branches. Once you've completed your work on a separate branch, you'll often want to incorporate that work back into your main line of development, typically referred to as the main or master branch. This process of integrating the changes from one branch into another is what we call merging.

To merge a branch in Git, you first need to switch to the branch that you want to merge into. This is done using the git checkout command. For instance, if you want to merge into the main branch, you would use:

```
git checkout main
```

Once you're on the target branch, you can then merge your feature branch into it. This is done using the git merge command. For example, if your feature branch is named add-endpoint, you would use:

```
git merge add-endpoint
```

Git will then attempt to automatically combine your changes from the add-endpoint branch into the main branch. If Git can successfully reconcile the changes from both branches, the merge will be completed automatically. However, if the same line or section of code has been modified

in both branches, Git will not be able to determine which change should take precedence. This results in a merge conflict, which requires manual intervention to resolve.

The following diagram provides a simplified visual representation of the branching and merging process in Git:

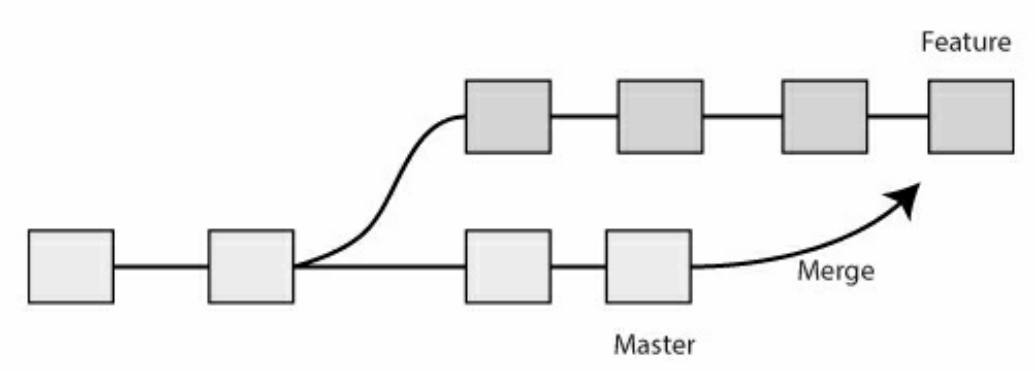


Fig 4.3 Branching & Merging

The ability to branch and merge makes Git incredibly flexible. It allows multiple developers to work on a project simultaneously, each in their own isolated environment. When they're ready, they can merge their changes back into the main codebase, even if other work has been done in the meantime.

Setting up GitHub

GitHub is a platform that allows developers to host and manage their Git repositories in the cloud. It not only provides a place to store code, but also offers various tools to facilitate collaborative development, such as issue tracking, code reviews, and automated testing.

The given below walkthrough is how you would set up and configure GitHub for your application development:

Sign up and Create New Repository

First, you need to sign up for a GitHub account if you haven't done so already. Once you're logged in, you can create a new repository by clicking the "+" icon in the top-right corner and selecting "New repository".

You'll be asked to name your repository and provide a short description. You can also choose whether to make your repository public or private and whether to initialize it with a README file.

Link your Local Repository to GitHub

Once your GitHub repository is set up, you can link it with your local repository. First, navigate to your local repository using the terminal or command line. Then, add your GitHub repository as a remote, which is a version of your project that is hosted on a server:

```
git remote add origin https://github.com/username/repository.git
```

Replace "username" with your GitHub username and "repository" with the name of your GitHub repository.

Push your Local Code to GitHub

You can now push your local code to GitHub using:

```
git push -u origin main
```

This command pushes your code to the "main" branch of the "origin" remote, which is your GitHub repository. The -u flag sets up tracking between your local and remote repositories, making future pushes and pulls easier.

Collaborate

Now that your code is on GitHub, you can easily collaborate with others. You can invite others to contribute to your project, submit changes for you to review, or even fork your repository and develop their own independent versions. You can also take advantage of GitHub's features like pull requests, issues, and actions to facilitate your development process.

Fetch and Merge Changes

If you're working with others, you'll need to regularly fetch and merge changes from the remote repository. You can fetch the latest changes with git fetch origin, and then merge any new commits from the "main" branch with git merge origin/main.

In short, GitHub and Git, when used in combination, form a powerful toolset for backend development. With these tools, you can easily manage and track changes to your codebase, collaborate with others, and maintain a well-structured development workflow.

Repositories, Forking and Pull Requests

Repositories

In GitHub, a repository (or "repo") is a project's workspace. It contains all of the project files and stores each file's revision history. Repositories can also have multiple collaborators and can be either public or private.

To create a new repository, click the plus sign in the top-right corner of your GitHub account and select "New repository". You can then specify the name, description, and visibility settings for the repository.

Once the repository is created, you can clone it to your local machine using the git clone command, which creates a local copy of your repository on your computer:

```
git clone https://github.com/username/repository.git
```

Forking

Forking is the process of making a copy of someone else's repository to your own GitHub account. This lets you freely experiment with changes without affecting the original project.
You can fork a repository by navigating to the repository's GitHub page and clicking the "Fork" button in the upper-right corner. Once the repository is forked to your account, you can clone it to your local machine and treat it just like any other GitHub repository.

Pull Requests

Pull Requests (PRs) are a feature of GitHub that facilitate collaboration. They're a way of proposing changes to a project: you make modifications to a repository, push them to a branch in your GitHub repository, and then open a pull request.

The repository's maintainers can then review your changes, discuss modifications, and eventually accept or reject your proposed changes.

Following is a basic workflow:

- Create a new branch: git checkout -b my-feature.
- Make some changes to the code and commit them: git commit -am "Add new feature".
- Push your changes to your fork on GitHub: git push origin my-feature.
- Navigate to your fork on GitHub, switch to "my-feature" branch, and click the "New Pull Request" button.
- Once the PR is opened, the project maintainers can review your changes and either merge them into the project or give feedback for you to make adjustments.

By understanding and using repositories, forking, and pull requests, you'll be well-equipped to contribute to open-source projects and collaborate with other developers on GitHub. These concepts form the basis of many modern development workflows and are essential tools for backend developers.

Using Git for Tagging, Stashing, and Reverting Changes

Tagging

Tagging in Git is a way to mark a specific point in your project's history as being important. Usually, developers use this functionality to mark release points (v1.0, v2.0 and so on). In other words, tags are references that point to specific points in Git history.

To create a new tag, you can use the git tag command. For example, to create a tag called v1.0, you can use the following command:

```
git tag v1.0
```

To list all the available tags in your repository, you can use git tag without any arguments.

To push your tags to the remote repository, you can use the git push --tags command.

Stashing

Git stash is a powerful command in Git that allows you to temporarily save changes that you have made to your working directory but do not want to commit yet. It takes your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.

Below is how you can stash your changes:

```
git stash save "Your stash message"
```

To see a list of all your stashes, you can use:

```
git stash list
```

To reapply the changes stored in a stash, you can use the git stash apply command followed by the stash name:

```
git stash apply stash@{0}
```

Reverting Changes

The git revert command can be used to undo changes to your repository's commits. It's a safe way to undo changes, as it does not alter the existing history—instead, it creates a new commit that undoes the changes made in a previous commit.

To revert a commit, you need to know its hash. You can get this hash by looking at the git log.

Once you have the hash, you can revert the commit with the following command:

```
git revert [commit_hash]
```

In practice, these commands provide flexibility in managing and modifying your project history. By understanding tagging, stashing, and reverting, you are better equipped to handle unexpected scenarios, manage releases, and maintain a clean and understandable commit history—all critical skills for backend developers. As always, be sure to thoroughly test all changes after applying these Git commands to ensure your codebase remains stable and functional.

Summary

In this chapter, we explored deep into the importance of version control and how Git, one of the most popular version control systems, forms an integral part of backend programming. We learned that version control systems like Git provide a structured approach to tracking changes, collaborating with other developers, and managing different versions of code, all of which are crucial in any large-scale, complex development project such as a backend system.

We began by setting up Git in our C++ environment, taking the first step towards effective version management. We learned the essential Git commands and understood how they're used in the workflow for tracking, staging, and committing changes. This knowledge was further enhanced by diving into branching and merging concepts, emphasizing how they promote parallel development without hampering the main code base. We learned how Git, when used effectively, helps in maintaining a clean and manageable code history.

Next, we introduced GitHub, a platform for hosting Git repositories that enhances collaborative possibilities. We examined GitHub repositories, understanding how to fork, clone, and manage these online versions of our code. We then moved onto pull requests, an essential aspect of collaborative programming, and saw how these help in proposing changes and facilitating discussions about the code. Finally, we looked at advanced Git usage, including tagging, stashing, and reverting changes, learning how these commands provide additional flexibility and control in managing our code. By the end of this chapter, we have equipped ourselves with vital skills to manage and maintain our backend code effectively using Git and GitHub.

CHAPTER 5: MANAGING DATABASE OPERATIONS WITH MONGODB

Role of Database in Backend Development

Databases play an essential part in the process of persistently storing, retrieving, and manipulating data in the world of application development. As a result, databases provide a backbone for the majority of the functions that applications carry out, if not all of them. Databases are essential for any operation, no matter how simple or complex it is; for example, registering a new user is a simple task, but analyzing user behavior patterns for a personalized recommendation system is a complex operation.

When developing the backend of a website or application, the database is frequently the most important part of the architecture. It is the location where business data, information about users, the state of applications, and other things are managed and stored. Backend applications are responsible for communicating with databases in order to carry out the four fundamental CRUD operations (Create, Read, Update, and Delete). These operations are necessary for any data-driven application.

- Create: This operation involves adding new data to the database. It can include actions such as registering a new user, creating a new product, or inserting a new blog post. These operations often involve the backend application receiving data from a client-side application, performing validations, and then issuing an INSERT command to the database.
- Read: Reading data from the database is the most common operation. Every time a user logs in, searches for a product, or navigates to a different page on a web application, the backend performs a read operation. This involves issuing a SELECT command to the database, often with certain conditions to filter the data.
- Update: Update operations modify existing data in the database. For example, when a user updates their profile, changes a setting, or an e-commerce application updates the stock of a product after a purchase. This involves the backend issuing an UPDATE command to the database.
- Delete: Finally, delete operations remove existing data from the database. This might occur when a user deletes their account, or an administrator removes a discontinued product from an e-commerce platform. A DELETE command is issued from the backend to the database to perform this operation.

In addition to these fundamental operations, databases also support more complex operations such as aggregation (the calculation of sums, averages, and other similar values), joining data from multiple tables, and transactions (the grouping together of several operations into a single unit of work). The specific requirements of the application should guide the selection of the database, which could be a conventional SQL database such as MySQL or PostgreSQL, or a NoSQL database such as MongoDB or Cassandra. Both types of databases store data in a relational format. In order to satisfy various requirements, certain applications might even make use of several distinct database types.

No matter which option is selected, it is essential for backend developers to have a solid

understanding of the fundamentals of database operation, including how to effectively design database schemas, efficiently write queries, and handle potential errors and exceptions. If you possess these skills, you will be able to ensure the robustness and efficiency of the application's data layer, which will, in turn, ensure the performance and reliability of the application as a whole.

Explore MongoDB

MongoDB is a popular NoSQL database that provides high performance, high availability, and easy scalability. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. It’s this flexibility, along with other features, that often makes it an attractive choice over traditional relational databases for many applications.

Flexible Data Model

The first major advantage of MongoDB over relational databases is its flexible schema. This flexibility allows for faster application development because developers don't have to predefine the structure of data or modify the structure if new fields are added. This also means you can store complex data structures, like nested arrays and documents, which can be challenging in a relational database.

Scalability

MongoDB's horizontal, scale-out architecture can make it a better fit for large and ever-growing datasets. Instead of scaling up by enhancing a single server (as is common in SQL databases), MongoDB scales out by adding more servers to the network. This ability to expand by adding more machines, known as sharding, is a powerful feature of MongoDB and provides an effective solution to the challenge of big data.

Speed

MongoDB is often faster than relational databases for many types of operations. This speed comes from its ability to handle large amounts of unstructured data, as well as from features like indexing, which can significantly speed up data access.

Replication and High Availability

MongoDB increases data availability with built-in replication, providing multiple copies of data across different servers. Automatic failover means that if your primary server goes down, a new primary will be up and running automatically. This feature helps ensure that your application is always up and running and that the data is constantly available.

Rich Query Language

MongoDB supports a rich and expressive object-based query language. This allows developers to build applications that can query and analyze their data in many different ways – by single keys, ranges, text search, graph processing, and geospatial queries through real-time aggregations providing powerful ways to access and analyze your data.

It is essential to keep in mind that although MongoDB offers a variety of advantages, it is not a solution that can be applied universally. It is not meant to be used as a replacement for relational databases in every circumstance. It's possible that the more organized approach provided by a relational database would be more beneficial for certain use cases and applications.

Install and Configure MongoDB

To integrate MongoDB into your C++ development environment, start by installing MongoDB on your system. Download the MongoDB Community Server from the official MongoDB website and follow the installation instructions. Next, install the MongoDB C++ driver, which allows your C++ application to interact with MongoDB. You can do this using a package manager like Homebrew or from source using GitHub. Once installed, include the necessary header files in your C++ code. Now, you can establish a connection to your MongoDB server using the MongoDB URI, and start creating, reading, updating, and deleting data in your MongoDB database from your C++ application.

The give below is a step-by-step walkthrough:

Install MongoDB

Depending on your operating system, the process for installing MongoDB can vary. You can find detailed instructions for your specific system in the MongoDB documentation. Below are steps for a Unix-based system:

Update your system's package database.

```
sudo apt-get update
```

Install MongoDB.

```
sudo apt-get install -y mongodb
```

To ensure MongoDB starts and restarts automatically with your server, you can issue:

```
sudo systemctl enable mongodb
```

Install MongoDB C++ Driver

The MongoDB C++ driver allows your C++ application to interact with MongoDB. BSON is a binary representation of JSON-like documents and is used when storing and accessing data.

Below are the steps:

First, install the required packages for building the driver:

```
sudo apt-get install build-essential libboost-filesystem-dev libboost-program-options-dev libboost-system-dev libboost-thread-dev
```

Next, clone the MongoDB C++ driver repository:

```
git clone https://github.com/mongodb/mongo-cxx-driver.git
```

Navigate into the directory and compile the driver:

```
cd mongo-cxx-driver/build

cmake -DCMAKE_BUILD_TYPE=Release -
DCMAKE_INSTALL_PREFIX=/usr/local ..

make EP_mnmlstc_core

sudo make install
```

Finally, you can use MongoDB in your C++ program.

Sample Program to Connect MongoDB

Below is a simple C++ program that connects to a MongoDB instance running on the same machine:

```
#include <bsoncxx/json.hpp>
#include <mongocxx/client.hpp>
#include <mongocxx/stdx.hpp>
#include <mongocxx/uri.hpp>
using bsoncxx::builder::stream::close_document;
using bsoncxx::builder::stream::open_document;
using bsoncxx::builder::stream::document;
int main(int, char**) {
  mongocxx::instance inst{};
  mongocxx::client conn{mongocxx::uri{}};
   auto collection = conn["testdb"]["testcollection"];
  document builder{};
   builder << "name" << "MongoDB" << "type" << "database" <<
"count" << 1
          << "versions" << bsoncxx::builder::stream::open_array
          << "v3.2" << "v3.0" << "v2.6"
          << close_array
          << "info" << open_document << "x" << 203 << "y" << 102 <<
close_document;


  collection.insert_one(builder.view());
   auto cursor = collection.find({});
```

```
  for (auto&& doc : cursor) {
    std::cout << bsoncxx::to_json(doc) << std::endl;
  }
}
```

This program connects to a MongoDB instance, inserts a document into a collection, and then retrieves and prints the document. It is important to keep in mind that in order for this code to work, MongoDB must be operational on the same machine (localhost) and on the standard port (27017). In the event that your MongoDB instance is operating in a different location, you will be required to supply the correct connection URI when creating the client.

These are the key primary steps that must be taken in order to get MongoDB operational within your C++ environment. From this point forward, you will be able to begin constructing applications that are more complex and can read, write, update, and delete data from your MongoDB database.

Database Schema for Full Stack Application

MongoDB uses BSON, a binary representation of JSON-like documents, to store data. Unlike SQL databases, MongoDB is schema-less. This means that documents in the same collection do not need to have the same set of fields, and the data type for a field can vary across documents within a collection.

That being said, it is crucial to design your data models carefully for optimal performance. In MongoDB, you should strive for a balance between embedding sub-documents and creating new collections based on your application's data access patterns.

Following is an example of a MongoDB schema for a hypothetical blog application:

```
{
  "user": {
    "username": "john_doe",
    "email": "john@myblogapp.com",
    "password": "hashed_password",
    "profiles": [
      {
        "platform": "twitter",
        "handle": "@johndoe"
      },
      {
        "platform": "instagram",
        "handle": "@johndoeig"
      }
    ]
  },
  "posts": [
    {
```

```
        "title": "My first blog post",
        "content": "This is my first blog post",
        "user": "john_doe",
        "comments": [
            {
                "user": "commenter1",
                "content": "Great post!",
                "timestamp": "2023-07-01T14:00:00Z"
            },
            {
                "user": "commenter2",
                "content": "Thanks for sharing!",
                "timestamp": "2023-07-01T15:00:00Z"
            }
        ],
        "tags": ["intro", "first", "blog"],
        "published_date": "2023-07-01T12:00:00Z"
    }
  ]
}
```

In this schema:

- Each user has a username, email, hashed password, and a profiles array containing social media handles.
- Posts are stored in an array, each containing a title, content, the username of the author, comments, tags, and the published date.
- Comments are stored in an array in each post, containing the username of the commenter, the content of the comment, and the timestamp of the comment.

This design embeds related data in a single document, which can be retrieved in one query. This works well for data that is frequently accessed together (like user profiles or blog post comments), and when there is a one-to-many relationship that doesn't result in an array that grows without bound. In case of relationships leading to unbounded growth, it might be better to use references and separate collections. This decision will be very specific to your use-case and how your application interacts with data.

Performing CRUD Operations

CRUD operations are the foundation of any application that involves persistent storage. CRUD stands for Create, Read, Update, and Delete - the four basic operations on any persistent storage.

Let us take a look at how we would perform these operations in MongoDB using the C++ MongoDB driver for our blog application example:

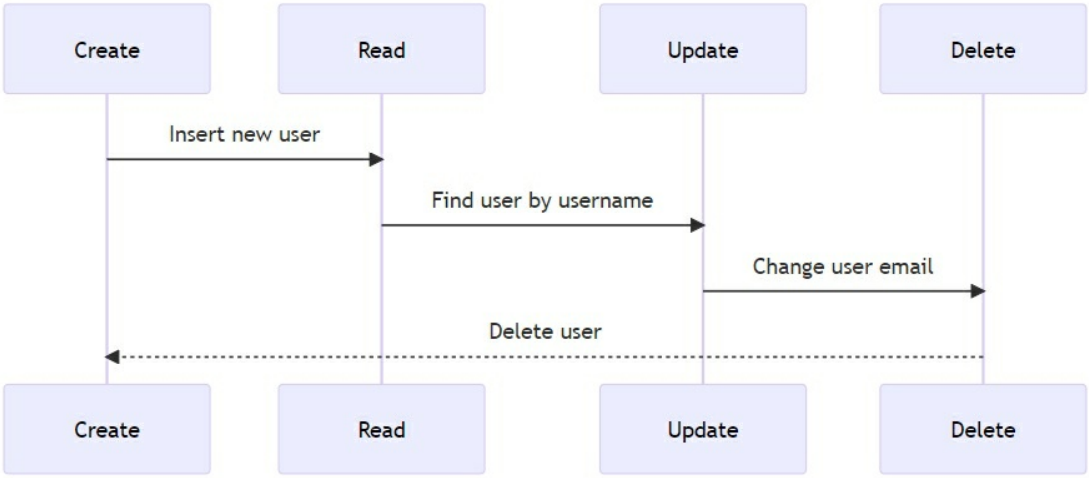


Fig 5.1 MongoDB CRUD Operations

Create

Inserting new documents in MongoDB is done using the insert_one or insert_many functions.

To create a new user:

```
mongocxx::collection user_collection = db["users"];
bsoncxx::builder::stream::document user_doc{};
user_doc << "username" << "new_user"
        << "email" << "new_user@myblogapp.com"
        << "password" << "hashed_password";
user_collection.insert_one(user_doc.view());
```

In the above snippet we are adding a new user to our 'users' collection. We construct a BSON document using the builder API and then use insert_one function to insert the document.

Read

To read documents from a MongoDB collection, we use the find function.

To find a user by username:

```
bsoncxx::builder::stream::document filter_builder;
filter_builder << "username" << "new_user";
bsoncxx::stdx::optional<bsoncxx::document::value> maybe_result =
user_collection.find_one(filter_builder.view());
if(maybe_result) {
  std::cout << bsoncxx::to_json(*maybe_result) << "\n";
}
```

In the above snippet, we are constructing a filter document with the username "new_user", then passing this filter to the find_one function to find the matching document.

Update

To update a document in MongoDB, we can use the update_one or update_many function.

To change the email of a user:

```
bsoncxx::builder::stream::document update_doc;
update_doc << "$set" << bsoncxx::builder::stream::open_document
        << "email" << "new_email@myblogapp.com"
        << bsoncxx::builder::stream::close_document;
user_collection.update_one(filter_builder.view(), update_doc.view());
```

We are creating an update document that uses the $set operator to change the email field of the matching document.

Delete

To delete documents, we can use the delete_one or delete_many function.

To delete a user:

```
user_collection.delete_one(filter_builder.view());
```

This line deletes the user with the username "new_user" from the 'users' collection.

In all the above operations, error checking is omitted for brevity but in a real-world application,

you should always check the result of these operations for any errors or exceptions. These operations are asynchronous and the C++ MongoDB driver provides several ways to deal with this asynchronicity. For complex applications, the aggregate function is often used, which allows for complex data manipulation and transformations on the server side.

Performing Complex Queries: Aggregation and Indexing

Aggregation and indexing are two powerful features of MongoDB that allow us to perform complex queries, filter and sort data, and increase the efficiency of our database operations.

Executing Aggregation

Aggregation in MongoDB is a way to perform complex data analysis on the documents in a collection. It provides a means to process data records and return computed results. Aggregation operations can group values from multiple documents together and perform a variety of operations on the grouped data to return a single result.

For instance, in a blogging application, you might want to know the number of posts each user has made. This can be achieved using MongoDB's aggregation framework. The $group stage groups input documents by a specified identifier expression and outputs a document for each unique grouping. The $sum operator is then used to increment a count for each group.

Given below is how you could do it:

```
mongocxx::collection posts_collection = db["posts"];
bsoncxx::builder::stream::document group_stage;
group_stage << "$group" << bsoncxx::builder::stream::open_document
        << "_id" << "$username"
        << "count" << bsoncxx::builder::stream::open_document << "$sum" << 1
        << bsoncxx::builder::stream::close_document
        << bsoncxx::builder::stream::close_document;
mongocxx::pipeline p{};
p.group(group_stage.view());
auto cursor = posts_collection.aggregate(p, mongocxx::options::aggregate{});
for(auto doc : cursor) {
  std::cout << bsoncxx::to_json(doc) << "\n";
```

```
}
```

In the above snippet, we're using MongoDB's aggregation framework. The $group stage groups input documents by the specified _id expression (in this case, the username field) and outputs a document for each distinct grouping. The $sum expression is used to increment the count for each group.

Creating Indexes

Indexing in MongoDB serves the same purpose as indexing in other database systems. An index is a data structure that holds a subset of the collection's data in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

Creating an index can improve the performance of search operations significantly. For example, creating an ascending index on the "username" field means MongoDB can use this index for queries that filter by the username field, speeding up these queries.

Let us create an index on the username field of our users collection as below:

```
bsoncxx::builder::stream::document index_builder;
index_builder << "username" << 1;
user_collection.create_index(index_builder.view(), {});
```

In the above snippet, we're creating an ascending index on the "username" field. This means MongoDB can use this index for queries that filter by the username field, which can significantly speed up these queries.

Indexes have some trade-offs, they can greatly speed up reads but will slow down writes because MongoDB needs to update the index every time a document is inserted or updated. It's important to find the right balance and only index fields that are frequently queried.

Summary

We took a deep dive into the world of databases in this chapter, concentrating specifically on MongoDB and the ways in which it can be integrated into the backend programming that we do in C++. To begin, we focused on gaining an understanding of the fundamental role that a database plays in the development of an application and the various operations that are carried out on the backend. A database management system, and in particular MongoDB, is an essential component of any modern full stack application. This component is required because it is able to store, retrieve, and manipulate the data that is used to drive the user interface.

We continued the process of installing and configuring MongoDB in a manner that was compatible with our C++ development environment. You have seen how MongoDB, which is a NoSQL database, offers flexibility in terms of the data structures it uses, as well as scalability and the ability to query the database. We talked about the schemas in MongoDB, focusing on the preference toward denormalization or embedding documents whenever it was possible to do so. We gained knowledge about the benefits that MongoDB schemas offer in comparison to traditional relational databases in terms of performance and scalability. A blog application was used as a sample to discuss the schema design considerations, which established a solid foundation for subsequent database interactions.

In the final step of this MongoDB implementation, we dove headfirst into the CRUD operations, which stand for create, read, update, and delete. In order to assist you in comprehending the interaction of the backend with the database, we broke down these fundamental operations and provided you with a step-by-step example as well as sample codes. After that, we discussed complicated queries that included aggregation and indexing. We gained an understanding of how to group data and carry out operations on it, as well as how indexing can significantly increase the performance of queries through the application of various practical examples. The utilization of these advanced database operations paves the way for our backend development process to incorporate new opportunities for the efficient manipulation and retrieval of data. The information that you've gained from reading this chapter provides a solid basis for the development of any backend application that involves database interactions and operations.

CHAPTER 6: CRAFTING REST APIS WITH GRPC

API Dynamics in Backend

Application Programming Interfaces (APIs) are an essential part of developing modern web applications, particularly in the context of backend development. An application programming interface, or API, is essentially a collection of rules and protocols that facilitates communication and collaboration between different software applications. It basically acts as a bridge between the various software components, making it easier for data to be transferred between them and for functionalities to be carried out. In the context of backend development, application programming interfaces (APIs) serve as the channel through which frontend and backend systems authenticate users, exchange data, and carry out a variety of other database-related tasks.

Consider the example of an online bookstore to get a better grasp on the function that APIs serve. When a user conducts a search for a book, the frontend (the user interface within the browser) sends a request to the backend server by way of an application programming interface (API). After receiving the request, the backend server will engage in a series of interactions with the database in order to retrieve the pertinent data (book details), and it will then send this information back to the frontend via the API. After that, the data is presented to the user by the frontend. APIs are an integral part of the backend development because they are used to orchestrate the entire operation and are therefore essential.

Several distinct types of application programming interfaces (APIs) have emerged over the course of time, each featuring a distinct set of benefits. Several examples include the protocols REST (Representational State Transfer), GraphQL, and gRPC. In spite of this, REST APIs have recently seen a surge in popularity and are increasingly becoming the go-to option for many developers when it comes to the construction of APIs for web applications. REST APIs carry out CRUD operations by utilizing a variety of standard HTTP methods, such as GET, POST, PUT, and DELETE, amongst others. These APIs operate over HTTP. This is consistent with the stateless nature of HTTP, which makes REST APIs easy to understand and straightforward to put into action. A client written in a different language is able to communicate with a REST API even though it was built using a different programming language (for example, C++). This feature is known as language independence.

The simplicity of REST APIs and the ease with which they can be integrated is another reason for their widespread use. They make use of the standard HTTP protocols and status codes, which are already well known to the majority of software developers. In addition, the JavaScript Object Notation (JSON) format, which is a lightweight and user-friendly data exchange format, is utilized by them. Scalability and statelessness are two additional benefits offered by REST APIs. Due to the stateless nature of REST APIs, it is necessary for each request sent from the client to the server to include all of the information required to comprehend and carry out the processing of the request. Because of this, the application does not require the server to be kept busy maintaining user sessions, which makes it much simpler to scale.

REST APIs do not, however, come without their own set of restrictions. Because they over-fetch or under-fetch the data, they can be less efficient when dealing with complex hierarchical data structures. This is because of the way they retrieve the data. In spite of these challenges, REST APIs continue to be the most popular option for developers due to the fact that they are easy to

use, simple, and have a wide range of applications. They are an essential component of any backend developer's toolkit in every sense of the word.

Introduction to gRPC API Framework

gRPC is a high-performance, open-source, and universal Remote Procedure Call (RPC) framework developed by Google. It enables developers to build distributed systems where services can call each other seamlessly as if they were local objects in the same process. The name gRPC stands for Google Remote Procedure Call, indicating its origin and primary function.

Features of gRPC

gRPC is designed to enable efficient communication between services in a microservices architecture. gRPC has several distinguishing features that set it apart from other communication protocols, making it a popular choice for developers.

Following are the distinguished features:

- Language Interoperability: gRPC supports multiple programming languages, including C++, Java, Python, Go, Ruby, and others. This allows services written in different languages to communicate with each other.
- High Performance: gRPC uses Protocol Buffers (protobuf) as its interface definition language, which leads to smaller payloads and faster serialization/deserialization compared to text-based formats like JSON used in REST APIs.
- Bi-directional Streaming and Flow Control: Unlike REST APIs that use HTTP/1.1 and only support request-response communication, gRPC uses HTTP/2 as its transport protocol. This allows gRPC to provide true bidirectional streaming between client and server, enabling both parties to read and write concurrently.
- Strongly Typed Service Contracts: With gRPC, you define your service contract using Protocol Buffers. This enforces a strong contract between services and provides many benefits such as efficient serialization, type-safety, and easy API evolution.
- Pluggable Authentication: gRPC supports a variety of authentication mechanisms, such as SSL/TLS and token-based authentication with Google, making it versatile for different security requirements.

We will look into setting up gRPC in your C++ development environment: To set up gRPC in your C++ environment, you'll need to install gRPC and Protocol Buffers libraries, configure your build system, and then compile and run a sample gRPC application.

Setting up gRPC for C++

Install Protocol Buffers Compiler (protoc)

The Protocol Buffers compiler is used to generate gRPC service code. You can download pre-compiled binaries from the Protocol Buffers GitHub repository. After downloading, extract the contents and add the bin folder to your PATH.

Install gRPC

The simplest way to compile gRPC from source is to clone the gRPC GitHub repository and then run a script from the repository to pull-in the latest version of gRPC submodules. Then you can build and install it:

```
$ git clone -b $(curl -L https://grpc.io/release)
https://github.com/grpc/grpc

$ cd grpc

$ git submodule update --init

$ make

$ sudo make install
```

Compile and Install gRPC C++ plugins

The gRPC C++ plugins are needed for the gRPC compiler to generate C++ code. You can build and install them as follows:

```
$ make grpc_cpp_plugin

$ sudo make install-grpc_cpp_plugin
```

Finally, your environment is ready to create gRPC services in C++. In the following sections and further chapters, we will be diving deeper into creating gRPC services and integrating them into your backend application.

Implement gRPC

In order to create a gRPC service, you must first define the service interface in a.proto file, then generate service code from the.proto file, and finally implement your service by making use of the code that was generated.

For the sake of demonstration, let us build a basic gRPC service for our blogging app. AddPost is the name of the RPC that will be provided by our service to customers so that they can post new blog entries.

Define the Service

First, we define our service in a .proto file. This file specifies the syntax of our protocol buffers, the service interface, and the structure of the payload messages.

In blog_service.proto, we write:

```
syntax = "proto3";
package blog;
// The blog post message type.
message Post {
  string title = 1;
  string author = 2;
  string content = 3;
}
// The response message containing the blog post ID.
message PostResponse {
  string id = 1;
}
// The Blog service definition.
service BlogService {
  // Sends a new blog post.
  rpc AddPost(Post) returns (PostResponse);
```

```
}
```

In the above snippet, we define a Post message that contains the title, author, and content of a blog post. We also define a PostResponse message that contains the blog post ID. Finally, we define a BlogService with one RPC, AddPost, that takes a Post message and returns a PostResponse message.

Generate the Service Code

Next, we use the protocol buffers compiler (protoc) to generate the service code from our .proto file. This command generates both the message classes (Post and PostResponse) and the service classes (BlogService).

```
protoc --proto_path=. --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` blog_service.proto
```

Implement the Service

Finally, we implement our service by subclassing the generated service class and providing an implementation for our AddPost RPC.

In blog_service_impl.cc, we write:

```
#include "blog_service.grpc.pb.h"

using grpc::ServerContext;

using grpc::Status;

class BlogServiceImpl final : public blog::BlogService::Service {

public:

  grpc::Status AddPost(ServerContext* context, const blog::Post* post, blog::PostResponse* response) override {

   // Your implementation goes here.

    // For instance, you could save the blog post to a database and then return the post ID in the response.

   return Status::OK;

 }

};
```

In this code, BlogServiceImpl is our service implementation. It subclasses blog::BlogService::Service and overrides the AddPost method. The actual implementation of this method would typically involve saving the blog post to a database and returning the post ID in the response. By generating code from a .proto file, gRPC provides a strong contract for our service and ensures that our service implementation adheres to that contract. This makes it easier for us to build reliable, high-performance backend services with C++. In subsequent chapters, we will dive deep further into how to build out this service and integrate it into a larger application.

Building CRUD API

As part of the expansion of our blogging application, we are going to improve the gRPC service that was described earlier by adding the remaining CRUD operations. These include GetPost, which can be used to retrieve a particular post; UpdatePost, which can be used to modify an existing post; and DeletePost, which can be used to get rid of a post. In order to guarantee that our service is well suited to handle the management of blog posts, each operation will be defined as its own RPC.

Update .proto File

We start by updating our blog_service.proto file to include the new operations:

```
syntax = "proto3";
package blog;
// The blog post message type.
message Post {
  string id = 1;
  string title = 2;
  string author = 3;
  string content = 4;
}
// The response message containing the blog post ID.
message PostResponse {
  string id = 1;
}
// The response message containing a full blog post.
message FullPostResponse {
  Post post = 1;
}
```

```
// The Blog service definition.
service BlogService {
  // Sends a new blog post.
  rpc AddPost(Post) returns (PostResponse);

  // Retrieves a blog post.
  rpc GetPost(PostResponse) returns (FullPostResponse);

  // Updates a blog post.
  rpc UpdatePost(Post) returns (PostResponse);

  // Deletes a blog post.
  rpc DeletePost(PostResponse) returns (PostResponse);
}
```

In this updated .proto file, GetPost, UpdatePost, and DeletePost have been added to the BlogService. Each RPC requires an input and output message, and we use PostResponse and Post as needed. We also added FullPostResponse to return a complete post from GetPost.

Generate Service Code

We then generate the service code as we did before:

```
protoc --proto_path=. --grpc_out=. --plugin=protoc-gen-grpc=`which grpc_cpp_plugin` blog_service.proto
```

Implement the Service

Finally, we implement our new RPCs in blog_service_impl.cc:

```
#include "blog_service.grpc.pb.h"
using grpc::ServerContext;
using grpc::Status;
```

```
class BlogServiceImpl final : public blog::BlogService::Service {
public:
  grpc::Status AddPost(ServerContext* context, const blog::Post* post, blog::PostResponse* response) override {
   // Your implementation goes here.
   return Status::OK;
 }
  grpc::Status GetPost(ServerContext* context, const blog::PostResponse* request, blog::FullPostResponse* response) override {
   // Your implementation goes here.
   return Status::OK;
 }
  grpc::Status UpdatePost(ServerContext* context, const blog::Post* post, blog::PostResponse* response) override {
   // Your implementation goes here.
   return Status::OK;
 }
  grpc::Status DeletePost(ServerContext* context, const blog::PostResponse* request, blog::PostResponse* response) override {
   // Your implementation goes here.
   return Status::OK;
 }
};
```

We've added stubs for GetPost, UpdatePost, and DeletePost. The actual implementation would typically interact with the database, performing the necessary operations based on the incoming request and preparing an appropriate response. The client-side code will be similar to server-side,

and a client application could be written in another language that gRPC supports, which makes gRPC really versatile.

The completion of our gRPC service for the blogging application is accomplished by implementing these RPCs. We now have a backend service that is both strongly typed and high performance, and it supports the full set of CRUD operations for blog posts.

Troubleshooting gRPC for API Implementation

gRPC is indeed a powerful tool for creating APIs, but it's not immune to challenges. When using gRPC in your C++ backend application, you might encounter issues like connection errors, slow response times, or unexpected behavior from services.

Following are some common issues you might encounter when using gRPC in your C++ backend application, along with tips on how to troubleshoot and resolve them:

Compilation Errors with .proto Files

Suppose you're writing a BlogService with an AddPost RPC method and a Post message. An error might occur if you've defined your message incorrectly. For instance, you may have used an unsupported data type or syntax in your .proto file.

```
message Post {
  int32 id = 1;
  string author_id = 2;
  string content = 3;
  string title = 4;
   datetimetime posted_at = 5; // Incorrect datatype
}
```

Solution

In this case, datetimetime is not a valid datatype. The posted_at field could be of type string, Timestamp, or a google.protobuf.Timestamp depending on how you want to manage it. The corrected Post message should look like this:

```
message Post {
  int32 id = 1;
  string author_id = 2;
  string content = 3;
  string title = 4;
   google.protobuf.Timestamp posted_at = 5; // Corrected datatype
```

```
}
```

Communication Errors between Client and Server

In the context of a blogging application, suppose you're trying to add a new post using the AddPost method, but it isn't executing correctly. You might be encountering an UNAVAILABLE error, which is a common issue when using gRPC.

Solution

This error often indicates that the client is unable to establish a connection with the server. It could be due to network issues, or it could be that the server is not running or not listening on the correct port.

To resolve this issue, first, verify that your server is running correctly and is listening on the port that the client is trying to connect to. You can do this by checking the server logs or using network tools to inspect the open ports on the server. If the server is running correctly, check for any network issues that might be preventing the client from reaching the server. This could involve checking your network configuration, firewall settings, or even your internet connection.

Errors related to Service Definitions

In another scenario, imagine that you've added a new DeletePost RPC method to your .proto file, but you forgot to regenerate the C++ code. Finally, when you're trying to implement this method in your C++ code, you're encountering errors.

Solution

This issue arises because the C++ code doesn't have the necessary classes and methods to implement the DeletePost method. Whenever you make changes to the .proto file, you need to regenerate the corresponding C++ code using the protoc command. This command generates the necessary classes and methods based on the service definitions in the .proto file.

So, to resolve this issue, run the protoc command to regenerate the C++ code. After doing this, you should be able to implement the DeletePost method without any errors.

Errors in gRPC Calls

While implementing the AddPost method, suppose you're seeing a FAILED_PRECONDITION error. This error is part of gRPC's standard set of error codes and indicates that the system was in a state where the requested operation is not allowed.

Solution

In the context of the AddPost method, this error could occur if you're passing an incorrect or invalid Post object to the method. For example, if the Post object is missing required fields or has invalid values, the server might reject the request, resulting in a FAILED_PRECONDITION error.

To resolve this issue, ensure that you're creating and passing the correct Post object to the

AddPost method. Check the values of all the fields in the Post object and while doing this, ensure they meet the requirements defined in the .proto file.

Performance Issues

Let us consider a scenario where your GetAllPosts method is running slowly when the number of posts increases. This could be due to inefficiencies in your message structures or server-side processing.

Solution

To improve the performance of the GetAllPosts method, consider using more efficient data types in your .proto file. For example, use int32 instead of string for numerical identifiers, or use enums instead of strings for fields with a limited set of values.

Also, evaluate your server-side code for potential improvements. If you're fetching all posts from the database in one go, it might be more efficient to fetch them in batches. This would reduce the amount of data that needs to be loaded into memory at once, potentially improving the performance of the GetAllPosts method.

Summary

In this chapter, we dove into the world of APIs, with an emphasis on their critical function in backend programming. We started with a high-level view and then narrowed in on the importance of application programming interfaces (APIs) in tying together different software components and enabling seamless communication. After that, we moved on to a more in-depth examination of REST APIs, during which we discussed how their stateless, client-server communication model has made them the most popular option in the landscape of full-stack development. We also discussed the shift towards other API styles, such as GraphQL and gRPC, and emphasized how important it is to understand the ever-changing web ecosystem and be able to adapt accordingly.

After that, we dove headfirst into an in-depth investigation of the gRPC API framework. Protocol Buffers (protobuf) is the language that gRPC, a high-performance open-source universal RPC framework, employs as its interface definition language. This makes it possible for.proto files to be used for the definition of services and message types. We walked you through the process of installing gRPC in the C++ environment you already have, discussed the process of creating service definitions, and put these definitions into practice in a sample application. In addition, we broke down the steps involved in developing a CRUD API for the running example of a blogging application that we provided. This demonstrates how to make the most of gRPC in order to develop backend services that are both effective and reliable.

We changed gears towards the end of the chapter and began troubleshooting common issues and errors that may occur when using gRPC for backend development. This was done so that we could conclude the chapter. This included addressing compilation errors with.proto files, resolving communication issues between the client and server, fixing errors related to service definitions, overcoming performance issues, solving issues when making gRPC calls, and more. Every issue was described in detail within the framework of the blogging application that my team and I have been working on, along with specific examples illustrating how to resolve each problem. You were equipped with the knowledge to independently troubleshoot and resolve these issues as a result of participating in this exercise, which provided you with valuable insights into the common pitfalls of backend development with gRPC.

CHAPTER 7: DEALING WITH CLIENT-SIDE AND SERVER-SIDE CACHING

Caching for C++ Backend Development

The utilization of caching is essential to the operation of high-performance web applications. It fulfills the function of a component that stores data in an intermediate state for the purpose of making subsequent requests for the same data more quickly satiated. The data that is saved in a cache is either a copy of data that has been saved in another location or the result of an earlier computation. Caching is a method that enables you to store data or files that are accessed frequently in a manner that makes it possible for them to be retrieved significantly more quickly than traditional methods of direct access.

The effectiveness of a system can be improved by the addition of a cache, which acts as an intermediary layer between the client and the server. This can be especially useful in server-side programming because it lowers the load on your server, shortens the amount of time it takes for your web application to respond, and improves the quality of the experience for the user as a whole.

When it comes to server-side programming, there are many different ways in which caching can be applied. Caching on the server can involve storing the results of expensive database queries, storing entire web pages, or even caching API responses. As an illustration, server-side caching can involve storing web pages. Basically, a candidate for caching is anything that is frequently requested, takes a significant amount of time to generate, and is also frequently accessed.

Caching is an essential component in the context of developing C++ backends, and it plays an important part in maximizing the speed and performance of applications. Responses that are fast and have a low amount of lag time are often necessary for modern C++ applications, particularly in the realm of backend web services. The power of caching can dramatically improve response times, reduce database load, and ultimately provide a smoother, faster user experience. This is true whether it is delivering a high volume of requests per second or performing complex operations.

We are going to explore the world of caching in great detail in this chapter. We are going to discuss its importance in backend development, the various kinds of caching, and the steps necessary to implement caching in an environment that uses C++. We will also discuss the implications of the decisions you make regarding caching, specifically how these decisions affect the performance and scalability of the backend services you provide.

Server-side Caching Strategies

There are several different types of caching that you can leverage depending on the requirements of your backend services. In this section, we will examine four types: In-memory caching, Database caching, CDNs, and HTTP caching.

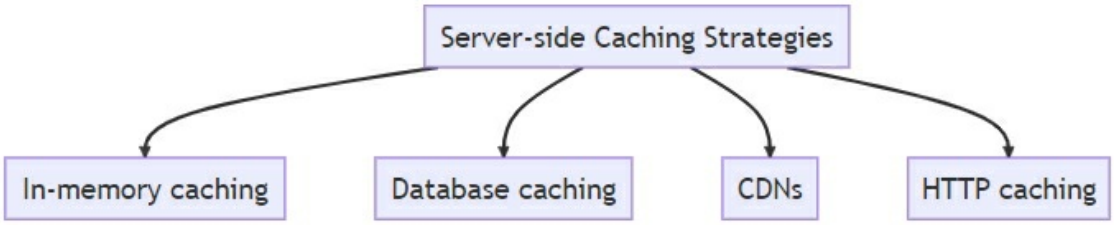


Fig 7.1 Types of Server-side Caching Strategies

In-Memory Caching

In-memory caching is a method where data is stored in the memory of a server for quick access. It is particularly beneficial for frequently read and infrequently updated data. The most well-known and widely-used in-memory caching system is Redis, though other systems like Memcached are also commonly used. In-memory caching can significantly reduce the latency of data retrieval as memory access is quicker than disk access. It's often used to cache the results of complex calculations, results of database calls, and session data.

For example, in a C++ backend, you might use in-memory caching to store user session data. Since this data is frequently accessed but doesn't change often, storing it in memory makes the application more responsive.

Database Caching

Database caching involves caching the results of queries made to the database. Since databases can often become performance bottlenecks, particularly when dealing with complex queries or large data sets, caching database results can significantly improve the performance of a backend system.

For instance, if your C++ backend is making complex SQL queries to a database to retrieve blog posts, caching the results of these queries can make the application more performant. The next time you need the same data, it can be retrieved from the cache rather than executing the same SQL query again.

Content Delivery Network (CDN) Caching

CDNs are a type of cache that stores data at the edge of a network. This type of caching is especially beneficial for static content that doesn't change frequently, such as images, CSS, and JavaScript files. By storing this content closer to the users accessing it, CDNs can significantly reduce latency and improve the performance of your application.
In the context of a C++ backend, you might use a CDN to serve static assets of a web application. For example, you might store images for a blog post on a CDN to ensure quick,

reliable access regardless of where your users are located.

HTTP Caching

HTTP caching involves storing the contents of an HTTP request to be reused for later responses. This can be implemented using various HTTP headers that control how, when, and for how long the caching occurs. It's a powerful way to cache web page resources, reducing the load on the server and improving response times.

HTTP caching could be utilized in a C++ backend to cache the responses of certain API endpoints. For example, if you have an endpoint that retrieves a list of all blog posts, you could cache this response so that subsequent requests are served faster.

Each of these different techniques for caching data offers a unique set of benefits and can be applied in a variety of settings. The specific requirements of your application, the kind of data you are working with, and the kind of performance optimization you are trying to achieve are often important considerations when selecting a caching method to use. You can significantly boost the performance of your C++ backend services, as well as the speed at which they are delivered and the quality of the user experience they provide, by making effective use of these caching techniques.

Process of Database Caching

Database caching is a process that can considerably optimize the interaction between an application and its database. In the case of NoSQL databases like MongoDB, the process of database caching follows these basic steps:

Query Execution

Initially, the application executes a database query. This query could involve fetching, updating, deleting, or creating data.

Query Result Storage

Once the result of the database query is returned, instead of immediately discarding this data after use, the application stores this data in a cache. The cache is generally implemented as a key-value store, with the query being the key and the result being the value. It's important to note that the cached data represents a snapshot of the data at a specific point in time and doesn't automatically update when the database data changes.

Cache Retrieval

The next time the application needs to execute the same query, it first checks the cache. If the cache contains an entry for this query (cache hit), the application can simply return the cached result instead of querying the database again. This significantly reduces the load on the database and improves the performance of the application as reading from a cache is typically faster than executing a database query.

Cache Invalidation

Whenever data is updated in the database that corresponds to a cached query, the corresponding cache entry becomes outdated or "stale". The application needs a strategy to handle these situations. It could either invalidate the stale cache entry (remove it from the cache), or update the cache entry with the new data. This step is crucial because if not handled properly, the application could end up using outdated data which could lead to inconsistencies.

For instance, let us consider a blog application with a MongoDB database. The application frequently retrieves the list of all blog posts to display on the homepage. Instead of executing the same expensive database query every time, the application can cache the result. When a user accesses the homepage, the application first checks the cache. If the data is present (cache hit), the application serves the cached data, reducing database load and improving response time. However, when a new blog post is added, the cached list of blog posts becomes stale. The application needs to handle this event and update or invalidate the cache accordingly. This could be achieved by setting up a database trigger or using an event-driven architecture.

Cache management, such as setting a suitable size for the cache, deciding on an eviction policy for when the cache is full (such as LRU - Least Recently Used), handling cache misses (when the requested data is not found in the cache), and setting a suitable time-to-live (TTL) for cache entries are also essential parts of implementing database caching.

Implementing Database Caching

Using MongoDB to implement database caching necessitates the addition of a new layer of abstraction between the application and the MongoDB database. We can use Redis, a high-performance in-memory data structure store that is excellent for caching, for our caching needs. Redis was designed specifically for caching.

Because it is typically the section of a blog that receives the most attention from readers, we will focus on the function of the blog application that is responsible for fetching blog posts. The following is a simplified illustration of how we could alter this function to include caching.

Suppose we have a function called fetchPosts() that retrieves all of the blog posts from the MongoDB database:

```
std::vector<Post> fetchPosts(mongocxx::collection& postCollection) {
  std::vector<Post> posts;
  auto cursor = postCollection.find({});
  for(auto&& doc : cursor) {
      Post post;
      // Assume that fillPostFromDoc() fills a Post object from a BSON document.
      fillPostFromDoc(post, doc);
      posts.push_back(post);
  }
  return posts;
}
```

As a next step, we will add caching to this function using Redis. Note that the above sample program assumes that you have a running Redis instance and you've set up a connection to it. We will use cpp_redis library to interact with Redis from C++.

```
#include <cpp_redis/cpp_redis>
// Assume that redisClient is a connected cpp_redis::client instance.
std::vector<Post> fetchPosts(mongocxx::collection& postCollection) {
```

```
  std::vector<Post> posts;
  std::string cacheKey = "posts";
  cpp_redis::reply redisReply = redisClient.get(cacheKey);
  redisClient.sync_commit();
   // Cache miss: key doesn't exist in Redis.
  if(redisReply.is_null()) {
      // Fetch posts from MongoDB.
      auto cursor = postCollection.find({});
      for(auto&& doc : cursor) {
          Post post;
          fillPostFromDoc(post, doc);
          posts.push_back(post);
      }
       // Cache the result in Redis. Assume that postsToJson() converts a vector of Post objects to a JSON string.
       // The third argument is the TTL (time-to-live) of the cache entry, after which it'll be automatically deleted.
      // Here it's set to 60 seconds.
      redisClient.set(cacheKey, postsToJson(posts), 60);
      redisClient.sync_commit();
  }
   // Cache hit: key exists in Redis.
  else {
      // Assume that jsonToPosts() converts a JSON string to a vector of Post objects.
```

```
        posts = jsonToPosts(redisReply.as_string());
    }
    return posts;
}
```

We must keep in mind if a blog post is added, updated, or deleted, we need to either update or invalidate the cache. This could be done by modifying the respective functions to also update or delete the posts key in Redis.

```
void addPost(mongocxx::collection& postCollection, const Post& post) {
    // Add post to MongoDB.
    postCollection.insert_one(postToDoc(post));  // Assume that postToDoc() converts a Post object to a BSON document.
    // Invalidate the cache.
    redisClient.del(std::vector<std::string>{"posts"});
    redisClient.sync_commit();
}
```

These are simplified examples and real-world applications would need more sophisticated cache management strategies. For instance, you might want to only cache popular blog posts or posts that are expensive to fetch, or use a more intelligent cache invalidation strategy to keep the cache as fresh as possible.

Advanced Cache Strategies

Advanced caching strategies can significantly enhance the performance of your blog application. The choice of caching strategy depends on the specific needs and constraints of your application. Lets apply one of the strategies on our blog application.

Caching Individual Blog Posts

In the previous example, we cached the entire list of blog posts. However, if individual posts are expensive to fetch, it might be beneficial to cache them separately. The key would be something like post:<id>, where <id> is the ID of the blog post.

```
// Assume that redisClient is a connected cpp_redis::client instance.
Post fetchPost(mongocxx::collection& postCollection, const std::string& postId) {
   Post post;
    std::string cacheKey = "post:" + postId;
   cpp_redis::reply redisReply = redisClient.get(cacheKey);
   redisClient.sync_commit();
   if(redisReply.is_null()) {
      // Fetch post from MongoDB.
      bsoncxx::document::view_or_value filter = document{} << "_id" << bsoncxx::oid(postId) << finalize;
      auto maybeResult = postCollection.find_one(filter);

      if(maybeResult) {
         fillPostFromDoc(post, maybeResult.value());

         // Cache the post in Redis for 60 seconds.
         redisClient.set(cacheKey, postToJson(post), 60);
         redisClient.sync_commit();
```

```
    }
    else {
        // Handle the case when the post doesn't exist.
    }
  }
  else {
    post = jsonToPost(redisReply.as_string());
  }
  return post;
}
```

Cache aside / Lazy Loading

In the examples so far, we've been using a strategy known as "cache aside" or "lazy loading". This means that we only load data into the cache when it's requested and it's not already in the cache. This is great when the cost of reading from the database is high and the data is frequently read. However, it can lead to high latency for cache misses as the request has to go all the way to the database.

Write-through Cache

The write-through strategy ensures data consistency between the cache and the database. When the data is written, it's written to the cache and the database at the same time. The cache is always up-to-date with the database and read operations have low latency as they always hit the cache.

Implementing a write-through cache requires modifying the write operations (like addPost(), updatePost(), and deletePost()) to also update the cache.

```
void updatePost(mongocxx::collection& postCollection, const Post& post) {
  // Update post in MongoDB.
   bsoncxx::document::view_or_value filter = document{} << "_id" << bsoncxx::oid(post.id) << finalize;
   bsoncxx::document::view_or_value update = document{} << "$set"
```

```
<< open_document << "title" << post.title << "body" << post.body << close_document << finalize;
  postCollection.update_one(filter, update);
  // Update cache.
   std::string cacheKey = "post:" + post.id;
  redisClient.set(cacheKey, postToJson(post));
  redisClient.sync_commit();
}
```

These are some strategies to consider when implementing caching in your application. Depending on your specific needs and the characteristics of your data, you might choose to use a different strategy or a combination of several strategies.

Using gRPC for Cache Performance

gRPC is a high-performance, open-source, universal RPC framework. It can be a handy tool in improving server-side caching for several reasons. gRPC provides a way for client applications to directly call methods on a server application as if it were a local object, making it easier for you to create distributed applications and services.

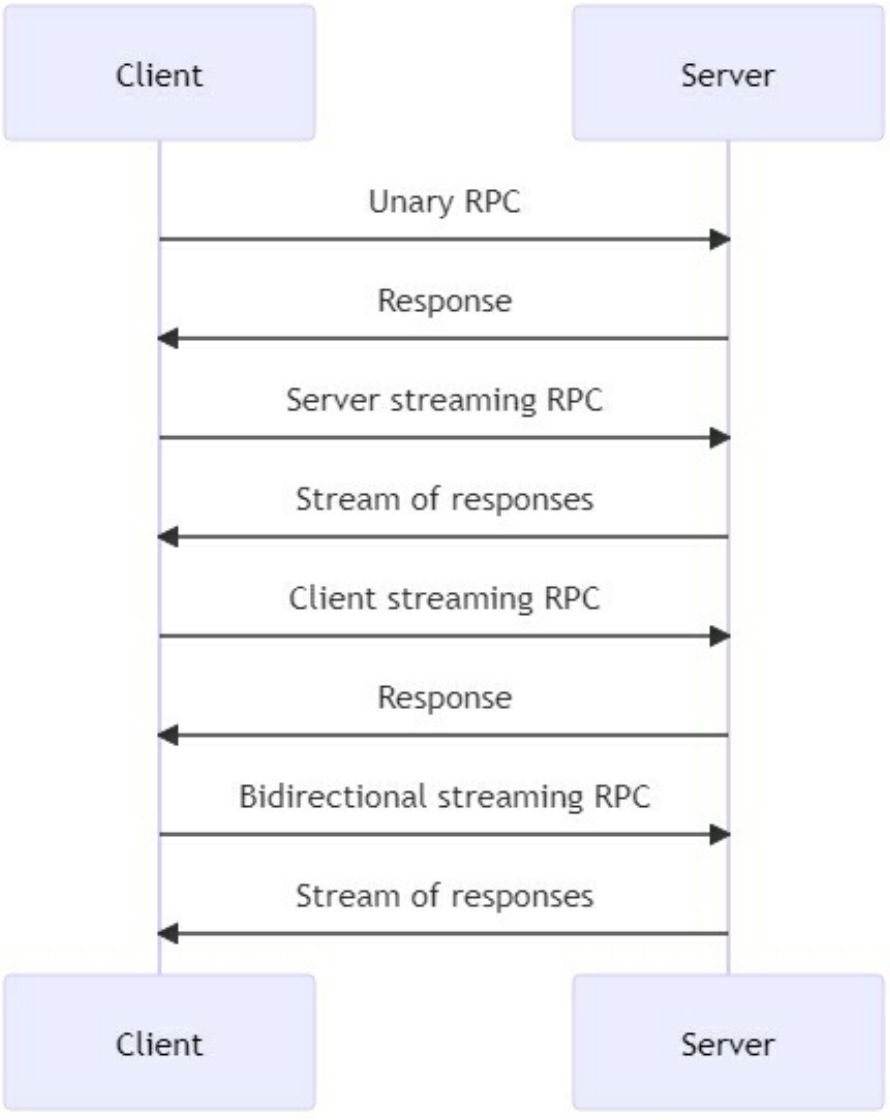


Fig 7.2 Working of gRPC

To leverage gRPC for server-side caching, we need to define a service in a .proto file. This service should have methods for fetching data that can be potentially cached. gRPC will automatically generate code from our service definition.

Define Service

Define the service in a .proto file:

```
syntax = "proto3";
service BlogService {
  rpc GetPost(GetPostRequest) returns (Post);
  rpc ListPosts(Empty) returns (PostList);
}
message GetPostRequest {
  string id = 1;
}
message Post {
  string id = 1;
  string title = 2;
  string body = 3;
}
message PostList {
  repeated Post posts = 1;
}
message Empty {}
```

Data Fetch

In the generated service implementation, use caching when fetching the data:

```
class BlogServiceImpl final : public BlogService::Service {
public:
   grpc::Status GetPost(grpc::ServerContext* context, const
GetPostRequest* request, Post* response) override {
```

```
        // Fetch the post from cache or from MongoDB.
        Post post = fetchPost(postCollection, request->id());
        // Fill the response with the fetched post.
        response->set_id(post.id);
        response->set_title(post.title);
        response->set_body(post.body);
        return grpc::Status::OK;
    }
    grpc::Status ListPosts(grpc::ServerContext* context, const Empty*
request, PostList* response) override {
        // Fetch the list of posts from cache or from MongoDB.
        std::vector<Post> posts = fetchPostList(postCollection);
        // Fill the response with the fetched posts.
        for(const Post& post : posts) {
            Post* postInResponse = response->add_posts();
            postInResponse->set_id(post.id);
            postInResponse->set_title(post.title);
            postInResponse->set_body(post.body);
        }
        return grpc::Status::OK;
    }
};
```

Establishing gRPC Server

Set up a gRPC server that uses the service implementation:

```
int main() {
  std::string serverAddress("0.0.0.0:50051");
  BlogServiceImpl service;
  grpc::ServerBuilder builder;
  builder.AddListeningPort(serverAddress,
grpc::InsecureServerCredentials());
  builder.RegisterService(&service);
  std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
   std::cout << "Server listening on " << serverAddress << std::endl;
  server->Wait();
  return 0;
}
```

In this way, gRPC helps to abstract away the complexities of remote procedure calls, enabling you to focus on the application logic itself. By defining services for your application's operations, you can seamlessly integrate caching, improve performance, and scale your application effectively. The gRPC uses the HTTP/2 protocol, which is much more efficient in terms of performance compared to HTTP/1. This is because HTTP/2 supports request/response multiplexing, which allows multiple requests and responses to be in flight at the same time. It also supports server push, which can help to optimize the caching mechanism further.

Cache Eviction Strategies

Cache eviction is a critical process in the management of cache memory, a high-speed storage layer that holds data that are likely to be used again by the system. The primary purpose of cache eviction is to make room for new entries by eliminating some existing entries from the cache. This process becomes necessary when the cache is full and a new entry needs to be accommodated. The method or policy that determines which existing entry should be removed is known as a cache eviction strategy. There are several different strategies used for cache eviction, but in this discussion, we will focus on three of the most common ones: Least Recently Used (LRU), Least Frequently Used (LFU), and Random Replacement (RR).

Least Recently Used (LRU)

The Least Recently Used (LRU) strategy is one of the most popular cache eviction policies. As the name suggests, this strategy involves removing the items that have not been used or requested for the longest time. The underlying assumption in this case is that if an item hasn't been requested for a while, it's less likely to be requested in the near future. For instance, in the context of a blog application where we have a cache of recently accessed blog posts, the LRU strategy would dictate that we discard the post that hasn't been accessed in the longest time when the cache becomes full.

Implementing LRU caching can be efficiently achieved using a combination of a hash map and a doubly-linked list. The hash map provides constant time access to the cache, ensuring that data can be retrieved quickly. On the other hand, the doubly-linked list allows us to add or remove items in constant time, making it an ideal data structure for maintaining the order of items based on their usage.

Least Frequently Used (LFU)

The Least Frequently Used (LFU) strategy is another common cache eviction policy. In this strategy, the cache evicts the item that is the least frequently requested. The rationale in this case is that if an item is rarely requested, it's less likely to be requested again soon. Implementing an LFU cache can be more complex than LRU, but it can also yield better results in some scenarios. It requires keeping track of the frequency of access, which can be done with a hash map for the cache and another to store the frequencies.

Random Replacement (RR)

Random Replacement (RR) is a cache eviction strategy that stands out for its simplicity. When the cache is full, an entry is selected at random and evicted to make space for the new one. This method is easy to implement but does not consider any usage pattern, which might lead to the eviction of important items that are frequently accessed.

Cache eviction strategies play a crucial role in optimizing the usage of cache space. By ensuring that the cache holds the most relevant data, these strategies can significantly improve the efficiency and performance of your backend application. However, it's important to note that the

best strategy to use often depends on the specific access patterns of your application. For example, a blog site might benefit from an LRU or LFU strategy because certain blog posts are likely to be accessed more frequently than others. On the other hand, an application with more unpredictable access patterns might be better served by a RR strategy. Therefore, understanding the nature of your application and its data access patterns is key to choosing the most effective cache eviction strategy.

LRU Implementation

The implementation of Least Recently Used (LRU) caching strategy involves two main components: a Hashmap for constant time access to data and a Doubly Linked List for constant time addition and removal of data. In the following example, we shall use C++ STL to implement an LRU Cache for our hypothetical blog application:

```
#include <unordered_map>
#include <list>
#include <iostream>
// Implementing an LRU Cache using C++ STL
class LRUCache {
public:
  LRUCache(int capacity): size(capacity) {}
  int get(int key) {
     if(cache.find(key) == cache.end()) return -1;
     moveToFront(key);
     return cache[key].second;
  }
   void put(int key, int value) {
     if(cache.find(key) != cache.end())
        removeKey(key);
     else if(cache.size() == size)
        removeKey(lru.back());
     addToFront(key, value);
  }
private:
```

```
  typedef std::pair<int, int> Pair;
  typedef std::list<int>::iterator ListIterator;
  std::unordered_map<int, std::pair<int, ListIterator>> cache;
  std::list<int> lru;
  int size;
   void addToFront(int key, int value) {
      lru.push_front(key);
      cache[key] = {value, lru.begin()};
  }
  void removeKey(int key) {
      lru.erase(cache[key].second);
      cache.erase(key);
  }
  void moveToFront(int key) {
      lru.erase(cache[key].second);
      lru.push_front(key);
      cache[key].second = lru.begin();
  }
};
```

In this piece of code, we defined a class called LRUCache to implement a Least Recently Used (LRU) cache. The LRUCache class contains two private members: a hashmap called cache and a doubly linked list called lru. The hashmap, cache, is used to store the key-value pairs of the cache. The keys are unique identifiers for the data, and the values are the actual data. The hashmap allows us to access any value in constant time, given its key, making it an efficient data structure for this purpose.

The doubly linked list, lru, is used to keep track of the order of usage of the keys. The most recently used keys are at the front of the list, while the least recently used keys are at the back. This list allows us to quickly identify and remove the least recently used key when the cache is

full and a new key-value pair needs to be added. The LRUCache class has a get() function, which checks whether a key exists in the cache. If the key does exist, the function moves the key to the front of the lru list, marking it as the most recently used key. It then returns the value associated with that key. If the key does not exist in the cache, the function returns a predefined value indicating a cache miss.

The put() function is used to add a new key-value pair to the cache. If the key already exists in the cache, the function removes the old key-value pair before adding the new one. This is done to ensure that the cache always has the most recent value for a given key. If the cache has reached its capacity, the function removes the least recently used key-value pair from the cache before adding the new one. This is where the LRU policy is applied. The moveToFront(), removeKey(), and addToFront() are helper functions that assist in maintaining the lru list and the cache hashmap. The moveToFront() function moves a given key to the front of the lru list. The removeKey() function removes a given key from both the cache and the lru list. The addToFront() function adds a new key to the front of the lru list and its associated value to the cache.

This code provides an interesting implementation of an LRU Cache. The performance of both the get and put operations is O(1), meaning they can be performed in constant time regardless of the size of the cache. This makes the LRU Cache design pattern suitable for a variety of real-world scenarios where resources are limited and efficiency is critical, such as in database queries, web page caching, and memory management in operating systems.

LFU Implementation

Implementing the Least Frequently Used (LFU) caching strategy is a bit more complex than LRU as it needs to keep track of the frequency of the access of data. Let us consider a simple implementation of LFU cache using C++ standard library (STL):

```
#include <unordered_map>
#include <list>
#include <iostream>
class LFUCache {
  int cap, size, minFreq;
   std::unordered_map<int, std::pair<int, int>> m; //Key to {value,freq};
   std::unordered_map<int, std::list<std::list<int>::iterator>> freq; //Freq to key list;
   std::unordered_map<int, std::list<int>> l; //key to iterator;
public:
  LFUCache(int capacity) {
     cap = capacity;
     size = 0;
  }

  int get(int key) {
     if(m.find(key) != m.end()) {
        freq[m[key].second].erase(l[key]);
        if(freq[m[key].second].empty()) freq.erase(m[key].second);
        ++m[key].second;
        freq[m[key].second].push_back(key);
```

```
        l[key] = --freq[m[key].second].end();
        if(minFreq == m[key].second - 1 && freq.find(minFreq) ==
freq.end())
            ++minFreq;
        return m[key].first;
    }
    return -1;
  }

   void put(int key, int value) {
    if(cap <= 0) return;
    int storedValue = get(key);
    if(storedValue != -1) {
        m[key].first = value;
        return;
    }
    if(size >= cap ) {
        m.erase(freq[minFreq].front());
        l.erase(freq[minFreq].front());
        freq[minFreq].pop_front();
        if(freq[minFreq].size() == 0) freq.erase(minFreq);
        --size;
    }
    m[key] = {value, 1};
    freq[1].push_back(key);
```

```
        l[key] = --freq[1].end();
        minFreq = 1;
        ++size;
    }
};
```

The get and put functions are quite similar to those in the LRU Cache. However, they also adjust the frequency of the elements. In this implementation, we use three data structures. m keeps track of the key to {value, freq}. freq is a map from frequency to the list of keys with that frequency. Finally, l keeps track of the iterator pointing to the position of the key in the list.

The get function increases the frequency of the accessed key and adjusts the position of the key in the freq map. The put function first checks if the key is already in the cache with the get function. If the key is not in the cache and the size of the cache is less than the capacity, it adds the key-value pair to the cache, else it removes the least frequently used key-value pair before adding the new one.

This LFU cache implementation ensures that the least frequently used data is removed when the cache is full and a new data point needs to be inserted. This pattern can be useful for certain types of data access patterns where the frequency of access is a good indicator for future accesses.

RR Implementation

Random Replacement (RR) is a cache algorithm that randomly selects a candidate item for replacement. The algorithm is quite simple and efficient, as it doesn't require maintaining any additional information about the frequency or recency of use for the items.
Below is an example of how you can implement it in C++:

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <cstdlib>
class RRCache {
private:
  int capacity;
  std::vector<int> keys;
  std::unordered_map<int, int> cache;
public:
  RRCache(int capacity) {
    this->capacity = capacity;
  }
  int get(int key) {
    if(cache.find(key) != cache.end()) {
      return cache[key];
    } else {
      return -1;
    }
  }
```

```
    void put(int key, int value) {
        if(cache.size() == capacity) {
            int indexToRemove = rand() % keys.size();
            int keyToRemove = keys[indexToRemove];
            cache.erase(keyToRemove);
            keys[indexToRemove] = keys.back();
            keys.pop_back();
        }
        cache[key] = value;
        keys.push_back(key);
    }
};
int main() {
    RRCache cache(2);
    cache.put(1, 1);
    cache.put(2, 2);
     std::cout << cache.get(1) << std::endl; //returns 1
     cache.put(3, 3); //evicts key 1 or 2 randomly
     std::cout << cache.get(1) << std::endl; //returns -1 if key 1 is evicted, 1 otherwise
     std::cout << cache.get(2) << std::endl; //returns -1 if key 2 is evicted, 2 otherwise
     cache.put(4, 4); //evicts key 2 or 3 randomly
     std::cout << cache.get(1) << std::endl; //always returns -1
     std::cout << cache.get(2) << std::endl; //returns -1 if key 2 is evicted, 2
```

```
otherwise
    std::cout << cache.get(3) << std::endl; //returns -1 if key 3 is evicted, 3 otherwise
    std::cout << cache.get(4) << std::endl; //always returns 4
   return 0;
}
```

In this code, we maintain a vector keys to store all keys currently in the cache, and an unordered map cache to store key-value pairs.

The get function directly retrieves the value of the key if it exists in the cache.

The put function checks if the cache has reached its capacity. If it has, it selects a random key from the keys vector, removes it from the cache and the vector, and then adds the new key-value pair to the cache and the vector.
One of the drawbacks of this approach is that it doesn't take into account the recency or frequency of use of the items in the cache. But the advantage is its simplicity and efficiency, making it suitable for applications where other methods of replacement are not effective or where the overhead of maintaining additional data structures is too high.

Troubleshooting Caching Errors

While caching can significantly speed up data retrieval and reduce the load on databases or external services, it also introduces a unique set of challenges that developers need to handle, especially in a backend application where multiple users might be interacting with the data simultaneously. These challenges include cache invalidation, cache stampede, and maintaining consistency in distributed systems.

Cache Invalidation

Cache invalidation is a common problem that arises when the data in the cache is no longer valid because the source data has changed. This discrepancy between the cached data and the source data can lead to outdated information being served to the user, resulting in inconsistencies and potential errors.

There are several strategies to deal with cache invalidation:

- One approach is to use a Time to Live (TTL) policy. In this method, each cached item gets a timestamp when it's added to the cache. If the item is requested and the timestamp is older than a certain threshold, the cache is invalidated, and fresh data is fetched from the source. This approach ensures that data does not remain in the cache indefinitely and is periodically refreshed. However, it may not be suitable for data that changes frequently or unpredictably.
- Another approach is the Write-Through strategy. In this method, changes to data are made to both the cache and the source data at the same time. This ensures that the cache always reflects the most recent state of the data, providing consistency. However, this method can be slower because every write operation needs to update both the cache and the source data.
- The Write-Back strategy is another method where changes are first made to the cache, and the source data is updated asynchronously. This can improve performance because write operations are faster. However, it increases the risk of data loss if the system crashes before the source data is updated.

Cache Stampede

Cache stampede is another challenge that occurs when multiple threads attempt to generate the same data at the same time, usually after a cache invalidation. This can lead to a heavy load on the database and can degrade performance.

There are several strategies to prevent cache stampede:

- Cache Locking is a technique where, when a thread fetches an empty or invalidated cache item, it immediately replaces it with a placeholder value. While this thread fetches the new data, other threads will only see the placeholder and know they need to wait. This prevents multiple threads from trying to fetch the same data simultaneously.

- Random Early Expiration is another strategy to prevent cache stampede. Instead of invalidating all cache items at once, they're assigned slightly random TTLs. This spreads out the load on the database as not all items are refreshed at the same time.

Nodes Inconsistency

In distributed systems, maintaining consistency across multiple nodes can be a challenge. If each node maintains its own cache, then changes made to data on one node will not be immediately reflected in the caches of other nodes. This can be solved with distributed cache synchronization, but it requires more resources and can become complex.

By understanding these challenges and implementing effective strategies to handle them, developers can ensure a reliable caching system that reduces unnecessary loads on the database and improves the user experience. Despite the complexities, the benefits of caching in terms of improved performance and efficiency make it a valuable technique in modern computing.

Summary

In this chapter, we explored into the concept of caching and its integral role in server-side programming. We learned that caching is a technique used to store frequently accessed data in a 'cache' to significantly improve application performance and reduce network latency. The type of caching employed largely depends on the requirements of the backend system and the nature of the data being cached. Database caching, for example, focuses on storing the results of database queries, which could be integral in reducing load on the database and enhancing application responsiveness.

We then walked through the process of implementing caching in a MongoDB-based blog application. This hands-on exercise illustrated the concept of cache insertion, cache retrieval, and cache eviction, and demonstrated various strategies to achieve each, from the straightforward time-to-live (TTL) method to more sophisticated techniques like write-through and write-back. Moreover, we discussed how Google's gRPC framework can be used to enhance server-side caching through its advanced features like HTTP/2 and binary data format, making it efficient for data transfer.

Finally, we explored cache eviction strategies such as Least Recently Used (LRU), Least Frequently Used (LFU), and Random Replacement (RR) and discussed their respective use-cases and implementation details. Along the way, we also discovered the potential pitfalls and common issues that can occur when working with caching, including cache invalidation, cache stampede, and consistency. We concluded by learning strategies to handle these issues effectively, such as using a cache locking strategy or cache population strategy to prevent cache stampede. This chapter offered a comprehensive understanding of caching in server-side programming and detailed insights on how to effectively implement and manage caching in a real-world application.

CHAPTER 8: MANAGING WEB SERVERS WITH NGINX

Web Servers Overview

Web servers are fundamental to the operation of web applications, serving as the crucial link between user interactions and server-side computations. The primary role of a web server is to store, process, and deliver web pages to clients upon request. As such, they play a critical role in the performance, security, and scalability of a web application.

Understanding the importance of web servers begins with recognizing the basic request-response cycle of the web. When a user accesses a web page, their browser sends a request to the server hosting that page. The server then processes this request, locates the requested resources, and sends them back to the client's browser for display. Without a web server, this exchange wouldn't be possible, and thus, accessing web content would be impossible.

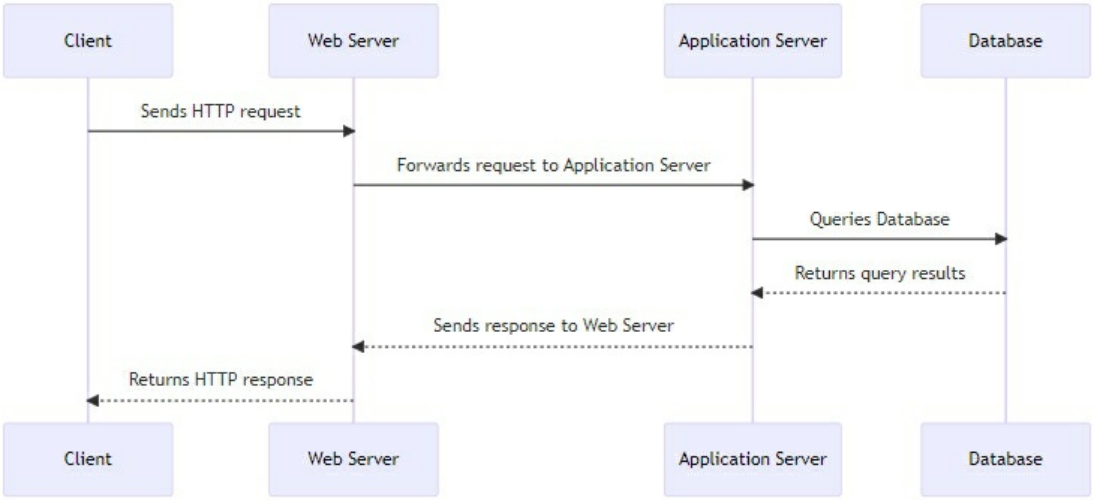


Fig 8.1 Working of Web Servers

In the above process flow,

- The client sends an HTTP request to the web server.
- The web server forwards the request to the application server.
- The application server queries the database.
- The database returns the query results to the application server.
- The application server sends a response back to the web server.
- Finally, the web server returns the HTTP response to the client.

The performance of an application can also be significantly influenced by the web servers that it uses. The speed with which a server processes requests and responds to those requests can have a significant impact on the experience that end users have. A poor user experience can occur as a result of a slow server response, which can cause delays in the delivery of content. On the other hand, a server that is quick and well optimized can deliver content more quickly, which results in a user experience that is more seamless and enjoyable.

Web servers are also extremely important in terms of security, which is another extremely important aspect. Web servers are responsible for ensuring the confidential transmission of data between clients and themselves. They can protect sensitive data by encrypting it using protocols

such as HTTPS, limit vulnerabilities, mitigate attacks, and protect themselves from being attacked. Because of this, having a secure web server is essential to retaining the trust of users and preventing breaches in data.

Scalability is another important quality of web servers to look for. A web application's growth may cause it to attract more visitors, which may result in an increase in the number of requests sent to the server. A reliable web server is able to effectively manage growing loads, ensuring that the application continues to be accessible and operates effectively even as the number of its users increases. In addition, web servers come with a variety of additional features, including load balancing, caching, compression, and even more, all of which contribute to an application's overall improved performance, dependability, and productivity.

Nginx is one of the most well-known web servers, and it is used by a significant number of high-traffic websites. In the following sections, we will investigate this server. Its prominence can be attributed to its high performance and stability as well as its rich feature set, straightforward configuration, and low consumption of available resources. This chapter will walk you through the complexities of configuring and optimizing Nginx for your C++-based backend application so that it runs as efficiently as possible.

Exploring Nginx

Nginx is a robust, high-performance, and flexible open-source software that can be utilized as a web server, reverse proxy, load balancer, and HTTP cache. Its inception in the early 2000s was a response to the C10K problem, a challenge faced by server software on handling a large number of clients concurrently. Nginx was designed to handle many concurrent connections with minimal memory usage, making it exceptionally suitable for modern web applications that need to support thousands or even millions of simultaneous connections.

Features

Web Server

Nginx can serve static content very efficiently, making it a superb choice as a web server for static websites or as a complementary server to handle static content in dynamic websites.

Reverse Proxy

A reverse proxy allows Nginx to handle all incoming network traffic and forward requests to other servers. This feature is useful for load balancing, offloading SSL connections, or for serving cached content.

Load Balancer

Nginx can distribute network traffic across multiple servers to ensure no single server gets overwhelmed. Load balancing techniques supported include round robin, least connections, and IP hash.

HTTP Cache

Nginx can store HTTP request responses. When a client requests the same resource again, Nginx delivers the cached response, thus reducing the load on your application servers and providing faster responses.

Mail Proxy

Nginx can be used as a proxy server for IMAP, POP3, and SMTP protocols, adding additional features like user authentication.

Components

Nginx consists of several components working together to handle client requests efficiently: Following is a deeper look at the key components of Nginx:

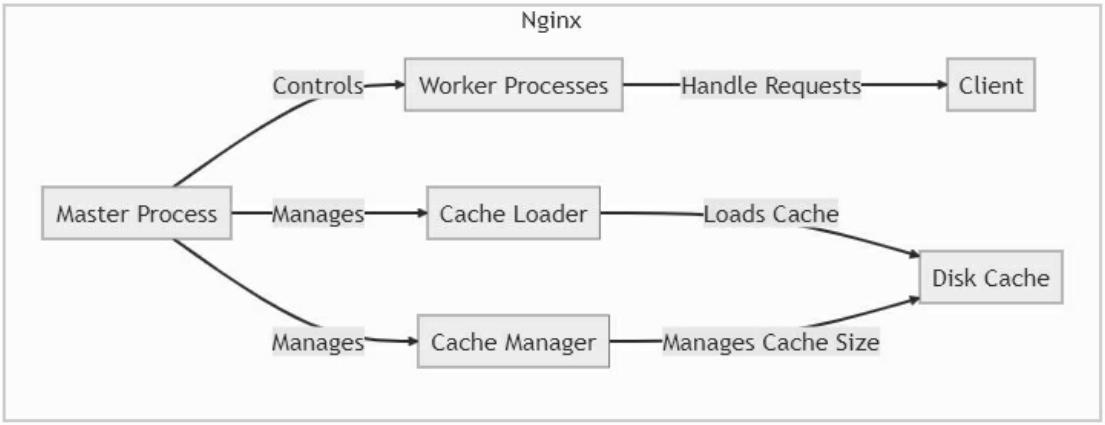


Fig 8.2 Nginx Components

Master Process

The master process is the heart of the Nginx application. It is responsible for reading and validating the configuration files. This process controls the worker processes and manages shared resources. The master process is also responsible for maintaining the overall health of the Nginx server. It monitors the worker processes and spawns new ones if a worker process fails. It also handles tasks that require privileged access, such as binding to ports and reading SSL certificates.

Worker Processes

Worker processes are the workhorses of Nginx. They handle the actual processing of requests from clients. Each worker process can handle thousands of simultaneous connections due to Nginx's event-driven architecture. This is in contrast to traditional web servers, which create a new thread or process for each connection, leading to significant overhead. The worker processes handle tasks such as establishing network connections, reading and writing content to the disk, and communicating with upstream servers.

Cache Loader

The cache loader is a process that runs when Nginx starts up and is responsible for loading the disk cache into memory. This process reads the existing cache metadata from disk, checks the validity of the cached data, and loads valid cache entries into memory. The cache loader runs once when Nginx starts, populates the in-memory cache, and then exits.

Cache Manager

The cache manager is a process that manages the disk cache size. It ensures that the cache size stays under the configured limit by removing old cache entries. The cache manager process runs periodically and cleans up the cache by removing outdated entries or entries that have not been accessed for a long time.

These above components work together to ensure that Nginx can serve a high number of concurrent requests with minimal resource usage. The modular and event-driven architecture of Nginx allows it to provide high performance, scalability, and efficient resource utilization.

Working Mechanism of Nginx

Nginx employs an event-driven architecture, as opposed to thread-based or process-based models. Each worker process in Nginx can handle thousands of connections due to its non-blocking, event-driven approach. When a request comes in, the worker process handles it and immediately returns to wait for more connections, making it capable of handling many requests simultaneously.

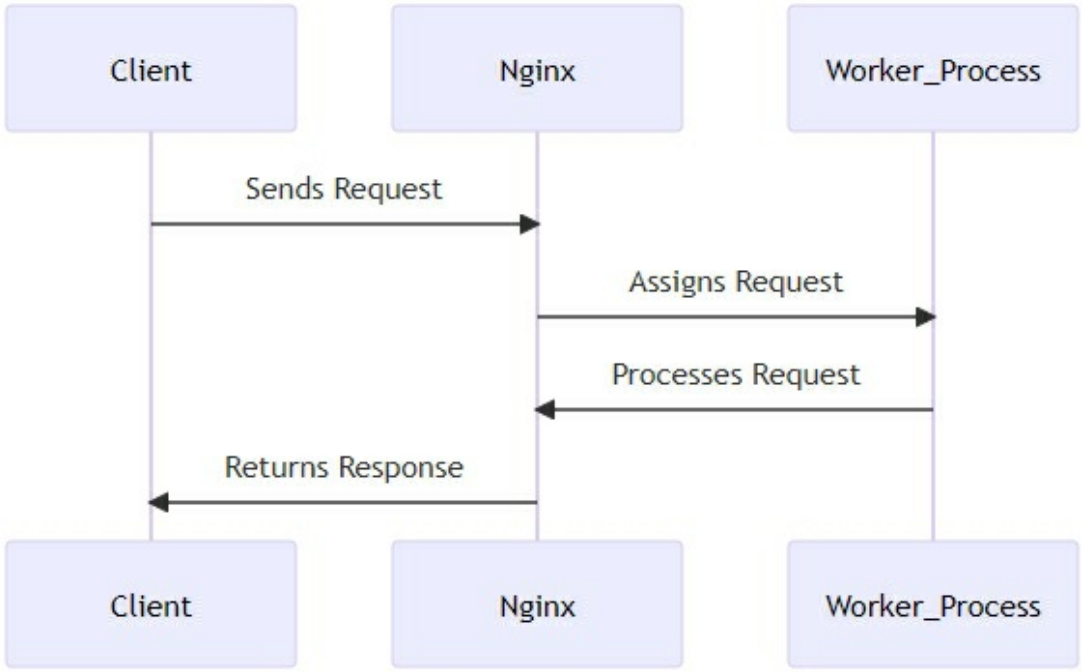


Fig 8.3 Working of Nginx

A typical workflow in Nginx would involve the master process starting up and spawning worker processes based on the configuration. When a client makes a request, one of the worker processes accepts the connection, processes the request (like serving a static file or forwarding the request to an upstream server), sends the response, and closes the connection. If the request can't be processed immediately (for instance, it requires reading a file from the disk), Nginx doesn't wait for the operation to complete and can immediately switch to another task, such as processing another client request. This non-blocking, asynchronous nature allows Nginx to achieve high performance and handle many connections simultaneously with limited resources.

Install and Configure Nginx for C++

Installing and configuring Nginx involves several steps. The given below walkthrough is a step-by-step guide to doing it in a Linux environment. The process should be similar if you're using a different operating system, but you may need to adjust some of the commands accordingly.

Update Your System

Before installing any package, it's a good idea to update your system's default applications. You can do this with the following commands:

```
sudo apt-get update
sudo apt-get upgrade
```

Install Nginx

Next, install Nginx with the following command:

```
sudo apt-get install nginx
```

Start Nginx

Once the installation is complete, you can start Nginx with this command:

```
sudo systemctl start nginx
```

You can also enable Nginx to start on boot with the following command:

```
sudo systemctl enable nginx
```

Adjust Firewall

If you have a firewall enabled, you'll need to adjust the settings to allow Nginx to function. You can do this with the following commands:

```
sudo ufw app list
```

You'll see a list of applications. Notice that Nginx Full, Nginx HTTP, and Nginx HTTPS are included. You can allow traffic on port 80 (standard HTTP traffic) with the following command:

```
sudo ufw allow 'Nginx HTTP'
```

Verify the change with:

```
sudo ufw status
```

Verify Installation

Verify that Nginx was installed correctly:

```
systemctl status nginx
```

You should see that Nginx is active and running. You can also check that Nginx is properly serving pages by navigating to your server's IP address in a web browser:

```
http://server_domain_or_IP
```

You should see the default Nginx landing page.

Configuring Nginx

The main configuration file for Nginx is located at /etc/nginx/nginx.conf. However, it is recommended that server blocks (similar to virtual hosts in Apache) be used for individual site configurations. These are available in the /etc/nginx/sites-available/ directory.

An example server block for a site could be:

```
server {
  listen 80;
  server_name myblogapp.com;
  location / {
     root /var/www/myblogapp.com;
     index index.html;
  }
}
```

This block is listening on port 80 for requests for myblogapp.com. It delivers files from the /var/www/myblogapp.com directory, presenting index.html as the directory index.

After adding your server block(s), you can enable them by creating a symbolic link to the sites-enabled directory:

```
sudo ln -s /etc/nginx/sites-available/myblogapp.com /etc/nginx/sites-
```

```
enabled/
```

After making any changes to the Nginx configuration, you should check the configuration for syntax errors:

```
sudo nginx -t
```

If no errors are reported, you can reload the new configuration into Nginx:

```
sudo systemctl reload nginx
```

Finally, Nginx is installed and configured to serve your C++ backend application.

Integrating Nginx with C++ Backend

Integrating Nginx with a C++ backend can be achieved using the FastCGI protocol, which is a variation of the Common Gateway Interface (CGI). FastCGI is designed to reduce the overhead associated with interfacing the web server and CGI programs, allowing a server to handle more web page requests at once.

To start the integration process, you first need to install FastCGI and spawn-fcgi. FastCGI is a binary protocol for interfacing interactive programs with a web server, while spawn-fcgi is a process manager that keeps the FastCGI processes alive. Both of these can be installed using the package manager of your operating system.

For example, on a Ubuntu system, you can use the following commands:

```
sudo apt-get update
sudo apt-get install libfcgi-dev spawn-fcgi
```

Once FastCGI and spawn-fcgi are installed, the next step is to configure your C++ application to use FastCGI. This involves including the FastCGI library in your C++ code and using the FastCGI API to handle requests and responses.

Following is a simple example of a C++ program using FastCGI:

```
#include <iostream>
#include <fcgi_stdio.h>
int main() {
  while (FCGI_Accept() >= 0) {
    std::cout << "Content-type: text/html\r\n"
```

```
            << "\r\n"
            << "<html>\n"
            << "<body>\n"
            << "<p>Hello, world!</p>\n"
            << "</body>\n"
            << "</html>\n";
  }
  return 0;
}
```

This program continuously accepts new FastCGI requests and responds with a simple HTML page. The final step is to configure Nginx to communicate with your C++ application using FastCGI. This is done by adding a server block in the Nginx configuration file for your C++ application. In this server block, you use the fastcgi_pass directive to specify the FastCGI server.

Given below is an example:

```
server {
  listen 80;
  server_name example.com;
  location / {
     fastcgi_pass 127.0.0.1:9000;
     include fastcgi_params;
     fastcgi_param SCRIPT_FILENAME /path/to/your/app;
  }
}
```

In this configuration, Nginx listens for requests on port 80 for the domain example.com. When a request is received, it is passed to the FastCGI server running on 127.0.0.1:9000.

Reverse Proxy

A reverse proxy is a type of server that sits in front of web servers and forwards client (e.g., web browser) requests to those web servers. Reverse proxies are used for functions like load balancing, web acceleration, and adding additional security controls.

Benefits of Reverse Proxy

Key benefits of a reverse proxy include:

- Load Balancing: A reverse proxy can distribute client requests across multiple servers, thus reducing the load on a single server and ensuring redundancy.
- Increased Security: By acting as an intermediary for requests from clients seeking resources from other servers, a reverse proxy server protects your backend servers from exposure to the internet.
- SSL Termination: By decrypting incoming requests and encrypting server responses, reverse proxies free up resources that your servers would otherwise use for encryption and decryption tasks.
- Caching: A reverse proxy can also work as a cache. When you request a web page, the reverse proxy might return a stored copy of the page, instead of sending the request to the actual server that hosts the page.

Setting up Reverse Proxy using Nginx

Let us take a look at how you can set up Nginx as a reverse proxy for your blog application.

First, you need to install Nginx, which we've already covered in the previous section. Once Nginx is set up, you can start configuring it as a reverse proxy.

Following are the steps:

Configure Nginx

Open the Nginx configuration file with a command like this:

```
sudo nano /etc/nginx/sites-available/default
```

Remove any existing content in the file and replace it with the following configuration:

```
server {
  listen 80;
  location / {
     proxy_pass http://localhost:5000;
```

```
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

This configuration tells Nginx to pass all incoming requests to your blog application, which is running on port 5000. Remember to replace http://localhost:5000 with the address and port of your application if they're different.

Check and Reload Nginx Configuration

Check the syntax of your configuration changes with the command:

```
sudo nginx -t
```

If the syntax is okay, the system will say "syntax is okay." If there's a problem, it will tell you where to look to find it.

When you're ready, reload Nginx to apply the changes:

```
sudo systemctl reload nginx
```

Once done, Nginx is now working as a reverse proxy for your blog application. It will listen for incoming requests on port 80 and pass them to your application on port 5000.

Handle HTTPS Traffic

Handling HTTPS traffic and high-volume traffic are two different aspects of managing an application with Nginx. Let us tackle them separately.

Managing HTTPS Traffic

To handle HTTPS traffic, Nginx needs to be configured to listen on port 443 (default port for HTTPS) and a valid SSL certificate needs to be installed.

Assuming you have SSL certificates from a trusted certificate authority (CA), you would add a configuration like this:

```
server {
   listen 80;
   listen [::]:80;
   server_name your_domain www.your_domain;
   location / {
       return 301 https://$host$request_uri;
   }
}
server {
   listen 443 ssl http2;
   listen [::]:443 ssl http2;
   server_name your_domain www.your_domain;
    ssl_certificate /etc/ssl/certs/your_domain.pem; # adjust for your certificate's location
    ssl_certificate_key /etc/ssl/private/your_domain.key; # adjust for your key's location
   location / {
       proxy_pass http://localhost:5000;
       proxy_http_version 1.1;
```

```
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

The first server block listens for HTTP connections and redirects them to HTTPS. The second server block handles HTTPS connections and proxies them to your application.

Handling High Traffic

To manage high traffic, Nginx can be used as a load balancer to distribute the load across several backend servers. This requires you to have your application running on multiple servers. Given below is an example configuration:

```
http {
    upstream backend {
        server backend1.myblogapp.com;
        server backend2.myblogapp.com;
        server backend3.myblogapp.com;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```

Nginx is set up to send incoming traffic to three different backend servers in this configuration.

These servers are backend1.myblogapp.com, backend2.myblogapp.com, and backend3.myblogapp.com. By default, the distribution is performed in a round-robin fashion; however, this behavior can be modified by using a variety of directives.

It is essential to keep in mind that managing high traffic requires not only optimizing your application and database but also making certain that your servers have sufficient resources, monitoring your system for problems, and more. You might also need to think about utilizing a Content Delivery Network (CDN) for your static assets or caching responses with Nginx in order to further reduce the load that is placed on your backend servers.

Load Balancers

Overview

In a web application, a load balancer is a tool that distributes traffic across multiple servers to ensure no single server bears too much demand. This increases the reliability and efficiency of your server setup. When one server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts to send traffic to it.

Nginx can be used as a load balancer in addition to its capabilities as a web server and reverse proxy server. It can use various algorithms to distribute the incoming traffic among the backend servers:

- Round Robin — Requests are distributed across the group of servers sequentially.
- Least Connections — A new request is sent to the server with the fewest current connections to clients.
- IP Hash — The IP address of the client is used to determine which server receives the request.

Setting up Load Balancers

In the context of our blog application, if you have multiple instances of the application running on different servers, you can use Nginx to balance the load among these instances. The given below walkthrough is an example configuration for setting up Nginx as a load balancer using the Round Robin method:

```
http {
  upstream backend {
    server backend1.myblogapp.com;
    server backend2.myblogapp.com;
    server backend3.myblogapp.com;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://backend;
    }
```

```
    }
}
```

In this configuration, backend1.myblogapp.com, backend2.myblogapp.com, and backend3.myblogapp.com are the servers running your blog application. The upstream directive defines the group of backend servers. In the server block, we have a location block that tells Nginx to proxy all incoming requests to one of the servers defined in the backend group. The servers are chosen in a round-robin manner.

If you want to use the least connections method, you can add the least_conn; directive to the upstream block, like this:

```
http {
    upstream backend {
        least_conn;
        server backend1.myblogapp.com;
        server backend2.myblogapp.com;
        server backend3.myblogapp.com;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```

And if you want to use the IP Hash method, you would add the ip_hash; directive:

```
http {
    upstream backend {
```

```
    ip_hash;
    server backend1.myblogapp.com;
    server backend2.myblogapp.com;
    server backend3.myblogapp.com;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://backend;
    }
  }
}
```

SSL Configuration

Secure Sockets Layer (SSL), and its successor, Transport Layer Security (TLS), are protocols used for securing the transfer of data over the Internet. They use encryption algorithms to ensure data integrity and privacy. SSL certificates provide a way for a client (web browser) to verify the identity of the server and establish a secure connection.

Using Nginx to Configure SSL

Configuring Nginx with SSL allows the server to deliver secure content to the client, enhancing the trust and privacy of the users in your blog application.

Given below are the general steps to configure SSL on Nginx:

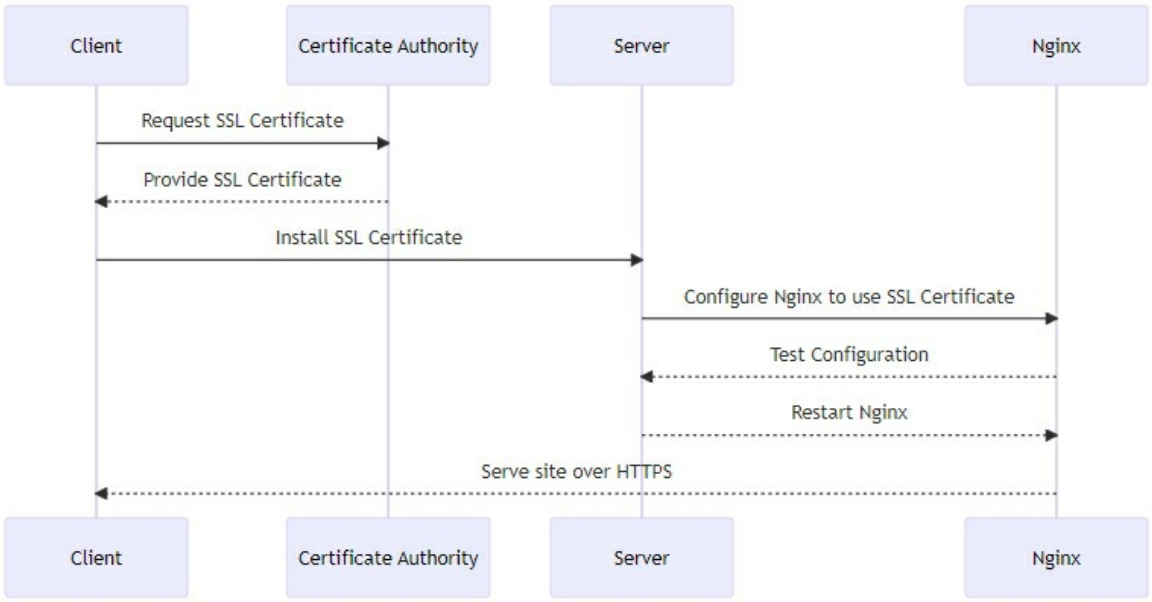


Fig 8.4 SSL Configuration using Nginx

Obtain an SSL Certificate

You need to obtain an SSL certificate from a Certificate Authority (CA). Many CAs offer free certificates, such as Let's Encrypt. For a more comprehensive solution, you may want to buy a certificate from a trusted provider.

Install the Certificate

Once you have an SSL certificate, it typically comes with a .crt (or .pem) and a .key file. You need to upload these files to your server. Generally, they are stored in the /etc/ssl/ directory.

Configure Nginx to use the Certificate

Once the certificate is installed, you need to tell Nginx to use it. Open your Nginx configuration file (usually located at /etc/nginx/nginx.conf or /etc/nginx/sites-available/default), and add the following lines to the server block that is serving your site:

```
server {
  listen 443 ssl;
  server_name myblogapp.com;
  ssl_certificate /etc/ssl/your_domain_name.pem; # or .crt
  ssl_certificate_key /etc/ssl/your_domain_name.key;
  location / {
      proxy_pass http://localhost:8080;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

Replace the paths in ssl_certificate and ssl_certificate_key with the actual paths to your SSL certificate and key files.

Restart Nginx

After updating the configuration, save the changes and exit the text editor. Test the configuration to verify there are no syntax errors:

```
sudo nginx -t
```

If the test is successful, restart Nginx to apply the changes:

```
sudo systemctl restart nginx
```

After you've configured Nginx to use SSL, you can verify that everything is working correctly by navigating to your website and checking that the URL begins with "https://". This indicates that your site is using the HTTPS protocol, which means that SSL is enabled.

In addition to this, most modern web browsers will display a padlock icon in the address bar when you visit a site that has SSL enabled. Clicking on this padlock will typically give you more information about the SSL certificate, including the issuing authority and the domain name it was issued for.

Before you set up SSL, it's important to ensure that your domain name is correctly pointed to your server's public IP address. SSL certificates are issued for specific domain names. If the domain name doesn't match the one specified when the certificate was issued, the certificate won't work, and users will see security warnings when they try to access your site.

Managing Static and Dynamic Assets

Nginx can be a very effective tool for serving dynamic content and assets in your blog application. It is designed to handle a large number of concurrent connections with low memory usage, which makes it excellent for serving static assets such as CSS, JavaScript, and image files. But, with the help of proxying, it can also effectively serve dynamic content.

Serving Static Content

In a typical web application, static content such as images, CSS, and JavaScript files are served from a specific directory in your application structure. Nginx, being a powerful and flexible web server, can be configured to efficiently serve these static files.

To serve static content with Nginx, you need to define a location block in your Nginx configuration file. This block tells Nginx that any request for a resource that matches a specific pattern should be served from a particular directory on your server.

Following is an example of how you might set this up:

```
server {
  listen 80;
  server_name myblogapp.com;
  location /static/ {
     alias /path/to/your/static/files/;
  }
  location / {
     proxy_pass http://localhost:8080;
  }
}
```

In the above example, any request that starts with http://myblogapp.com/static/ will be served by Nginx directly from the directory specified in the alias directive (/path/to/your/static/files/). This can significantly improve your application's load time as Nginx is very efficient at serving static files.

Serving Dynamic Content

For dynamic content delivery, Nginx can be set up as a reverse proxy. This indicates that Nginx

will forward client requests to the actual servers (your application) that are able to fulfil those requests, and then it will relay the responses back to the client. When you have an application server that is running on a specific port (for example, port 8080), and you want to make it accessible via standard HTTP or HTTPS ports (80 or 443), this is a particularly helpful feature to have.

The following is an illustration of one possible configuration for Nginx to carry out the function of a reverse proxy for a web application that is running on port 8080:

```
server {
  listen 80;
  server_name myblogapp.com;
  location / {
     proxy_pass http://localhost:8080;
     proxy_set_header Host $host;
     proxy_set_header X-Real-IP $remote_addr;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
     proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

The configuration provided above is designed to route all incoming HTTP requests to a backend application that is running on localhost:8080. In this setup, Nginx acts as a middleman, forwarding the requests to the application, waiting for the application to generate a response, and then delivering that response back to the client. This process is known as proxying.

It's important to note that any modifications made to the Nginx configuration file necessitate a syntax check and a restart of the Nginx service for the changes to take effect. You can check the syntax of your configuration file using the command sudo nginx -t. If the syntax is correct, you can then restart Nginx with sudo systemctl restart nginx.

Summary

In this chapter, we focused on Nginx as we gained an understanding of the critical function that web servers play in full-stack application stacks. It is possible to use Nginx as a web server, reverse proxy, load balancer, and HTTP cache with this versatile open-source software. Nginx is very powerful. It is well-known for its high performance and stability as well as its rich feature set, straightforward configuration, and efficient use of resources. The procedure of installing and configuring Nginx on a C++ development environment was broken down into its component parts and thoroughly explained.

We dove deep into the nitty-gritty details of how to configure a reverse proxy for our blog application by utilising Nginx. A type of server known as a reverse proxy is one that sits in front of web servers and transmits client requests to the web servers that are sitting behind it. This enables a better control over traffic, which in turn provides improved security, load balancing, and the ability to dynamically add, remove, or upgrade servers. We also gained familiarity with the use of Nginx for the management of high traffic and HTTPS connections, which brought to light the server's impressive capacity for managing multiple connections concurrently while consuming only a small amount of memory.

In the latter portion of this chapter, the primary focus was placed on configuring SSL and load balancing with Nginx. Load balancing is an essential component of any large-scale application because it ensures the most effective distribution of network traffic across multiple servers, thereby preventing any one server from becoming a bottleneck and slowing down the application as a whole. The process of achieving load balancing is made much simpler and more effective when using Nginx. In addition, SSL configuration was discussed, along with the steps required to properly establish a secure connection for the purpose of data transfer. Finally, we covered how to use Nginx in a blog application to serve dynamic content or assets, demonstrating Nginx's versatility in managing both static and dynamic content. This was the final topic of our discussion. Because it has so many useful features, Nginx is an excellent option for back-end development.

CHAPTER 9: TESTING YOUR C++ BACKEND

Why Testing Matters?

Testing is an essential step in the process of developing software from start to finish. Testing is the process of analyzing the operation of a software program in order to identify any flaws or inconsistencies that may exist within the program. Testing is essential to ensuring that the software functions as intended and provides a positive experience for end users. It also helps to maintain the software application's reliability and robustness, which is an added benefit.

Unit testing is the initial stage of the testing process. Unit testing is a subset of software testing that examines and verifies the functionality of a software's individual components and modules. Validating that every component of the software works as intended is the goal of this process. In the context of C++, testing individual functions or classes is what is meant by the term "unit testing." Given a particular set of arguments, the objective is to verify that each function and class performs as anticipated. This includes dealing with exceptional circumstances and error states. Unit testing in C++ can be performed using any one of a number of available frameworks, including Google Test and Catch2. These frameworks offer a collection of tools and functionalities that, when combined, simplify the process of writing and managing unit tests.

The subsequent level of testing is known as integration testing. Testing of this kind is carried out with the purpose of locating flaws in the interaction between the integrated components being tested. To put it another way, integration testing concentrates on analyzing how the various components of the software interact with one another. This is of utmost significance in a backend environment, which is characterized by frequent communication between the various services and components that make up the system. It is possible to test the integration points of REST APIs using software such as Postman or Insomnia. This will ensure that the APIs function as intended when combined.

End-to-End testing, also known as E2E testing, is a methodology that determines whether or not the flow of an application is operating as intended from the very beginning to the very end. End-to-end testing is performed with the intention of determining which parts of the system are dependent on one another and checking to see that the data integrity is preserved throughout all of the various system components and systems. It involves testing an application's entire functional flow without paying any particular attention to the logic that lies beneath the surface of the system. In the context of backend development, testing the complete request and response cycle of an API would be an example of what is meant by E2E testing. This includes testing that the API provides an appropriate response to various kinds of inputs, that it deals with errors in an appropriate manner, and that it performs well under a variety of conditions.

Ultimately, it is important to note that testing is an integral part of the software development lifecycle. It contributes to ensuring that the software application is robust and reliable, and that it offers a pleasant experience to its users. Each level of testing, from the testing of individual components to the testing of the interaction points between different modules to the testing of the complete functional flow, provides its own distinct value and contributes to the overall quality of the software application. Unit testing, integration testing, and end-to-end testing are examples of these types of testing. As a result, it is essential to commit both time and resources to the process of developing an all-encompassing testing strategy and infrastructure for the software projects

you are working on.

C++ Testing Frameworks: Doctest and Google Test

Doctest and Google Test are widely used testing frameworks in the C++ development environment. They provide a robust and flexible platform for writing, managing, and running tests. These frameworks support a range of assertions to validate code functionality, helping developers ensure their code is correct and reliable.

Doctest

Doctest is a relatively new C++ testing framework that is inspired by the unittest {} functionality of Python and the doctest Python module. It is extremely light and is claimed to be the lightest feature-rich C++ testing framework. It can be used for writing test cases, and it also supports a form of executable documentation.

The key advantages of Doctest are:

- Lightweight: Doctest adds minimal compile time compared to other testing frameworks. This makes it particularly suitable for large projects with many tests.
- Ease of Use: Doctest allows tests to be written directly in the C++ code without requiring any additional source files. This makes it easier to set up and use.
- Flexibility: Doctest supports various test case scenarios, like parametric tests, subtests, and templated test cases, and allows logging inside tests and sections.
- Integration: Doctest provides integration with the system's logging module, so you can write tests and log messages using the same syntax.

Google Test (gtest)

Google Test, also known as gtest, is one of the most popular C++ testing frameworks. It was developed by Google and is used in thousands of projects both within and outside of Google.

The key advantages of Google Test are:

- Rich Functionality: Google Test supports a wide range of testing needs, from simple tests to complex functional tests. It provides assertions to handle comparisons, and its advanced features allow you to control how tests are run.
- Portability: Google Test works on various platforms, and its code does not have to be modified when moving between different platforms.
- Flexibility: Google Test supports automatic test discovery, advanced options for running tests, and the ability to create complex test hierarchies.
- Robust: Google Test can handle exceptions and is designed to work well with the Google C++ Mocking Framework.

Both Doctest and Google Test are excellent choices for testing in C++. However, the best choice

depends on your specific requirements. Google Test is a more mature and feature-rich framework and is excellent for complex applications. In contrast, Doctest is lightweight and quick to compile, making it more suitable for applications where compile time is a critical factor.

Install Google Test

Following are the steps to install it in your development environment.

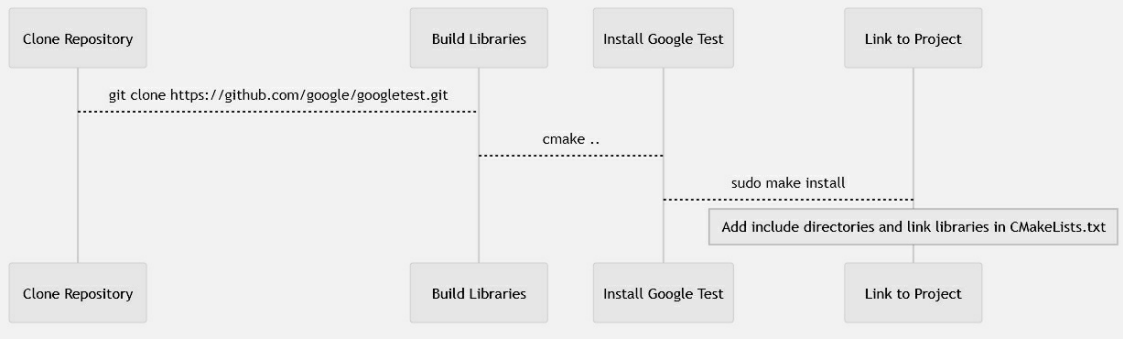


Fig 9.1 Process of Installing Google Test

Clone the Google Test Repository

Firstly, you will need to clone the Google Test repository to your local machine. You can do this by using the git command in your terminal:

```
git clone https://github.com/google/googletest.git
```

Build Google Test Libraries

Once the repository is cloned, you need to build the Google Test libraries. Google Test uses CMake for building, which is a common open-source system used to manage software builds. Navigate into the googletest directory that was created when you cloned the repository:

```
cd googletest
```

Then, create a new directory for the build files and navigate into it:

```
mkdir build
cd build
```

Then, run the CMake command to build the project. This will create the necessary Makefiles for you:

```
cmake ..
```

After running the cmake command, you can build the project using make:

```
make
```

Install Google Test

After building the libraries, you can install them on your system. You can use the make install command to do this:

```
sudo make install
```

Link Google Test to your Project

After installing the Google Test libraries, you will need to link them to your C++ project. You can do this by adding the necessary include directories and link libraries in your CMakeLists.txt file:

```
include_directories(/usr/local/include)
link_directories(/usr/local/lib)
...
add_executable(your_test your_test.cpp)
target_link_libraries(your_test gtest gtest_main)
```

In the above snippet, your_test.cpp should be replaced with the name of your test file. gtest and gtest_main are the Google Test libraries.

Perform Unit Testing for Data Loss

Let us create a simple Google Test unit test to verify a hypothetical data saving function in your backend application. For the below sample use-case, assume we have a class DataStorage with a function SaveData that saves data, and GetData which retrieves it.

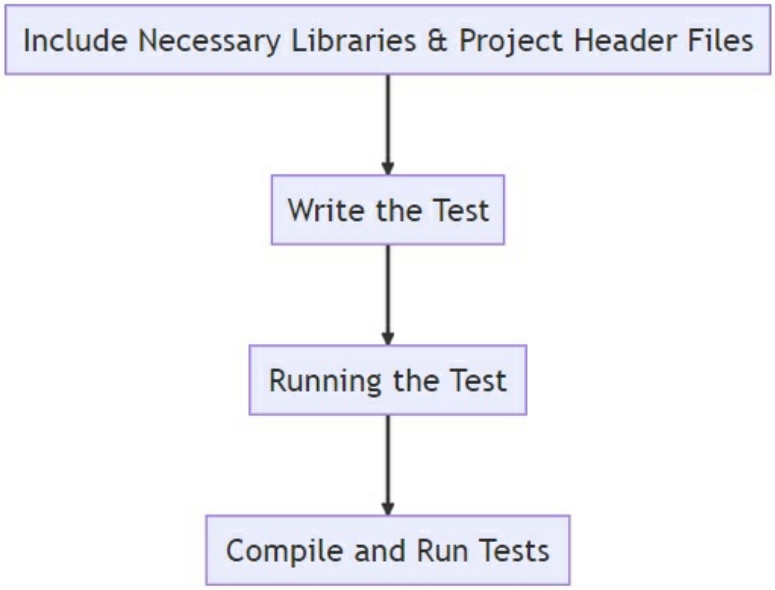


Fig 9.2 Process of Unit Testing

Include Necessary Libraries & Project Header Files

First, you need to include the necessary libraries and your project header files in your test file.

```
#include <gtest/gtest.h>

#include "data_storage.h" // This should be the path to your DataStorage class
```

Write the Test

Next, you can write the test itself. Given below is an example of a test that checks for data loss:

```
TEST(DataStorageTest, CheckDataLoss) {
  // Initialize the data storage
  DataStorage ds;
   // Set up some test data
```

```
  std::string testData = "Test Data";
  // Save the data
  ds.SaveData(testData);
  // Retrieve the data
  std::string retrievedData = ds.GetData();
   // Check if the retrieved data is the same as the saved data
  EXPECT_EQ(testData, retrievedData);
}
```

In this test, we're first initializing our DataStorage object. We then set up some test data and save it using the SaveData function. We retrieve the data using GetData, and then use the EXPECT_EQ macro to check if the retrieved data is the same as the saved data.

Running the Test

You can compile and run your tests using CMake and Make. Given below is an example of how you might do this:

```
mkdir build
cd build
cmake ..
make
./run_tests
```

In the above snippet, run_tests is the executable created by your CMakeLists.txt file that runs your tests.

This is a very simplified sample program, but you can extend this to fit your needs. You might, for instance, have more complex data than a simple string, or you might have multiple tests for different functions or use cases. The key thing is that you want to test your code often and thoroughly to ensure it behaves as expected.

Perform Integration Testing for MongoDB and gRPC

An integration test aims to verify that different components of your system work together as expected. With a gRPC server and MongoDB database in your system, an integration test could involve verifying that requests to the gRPC server result in the correct actions in the database. Given below is an example of how you might structure such a test using Google Test. This test will focus on the CRUD operations of our blog post application.

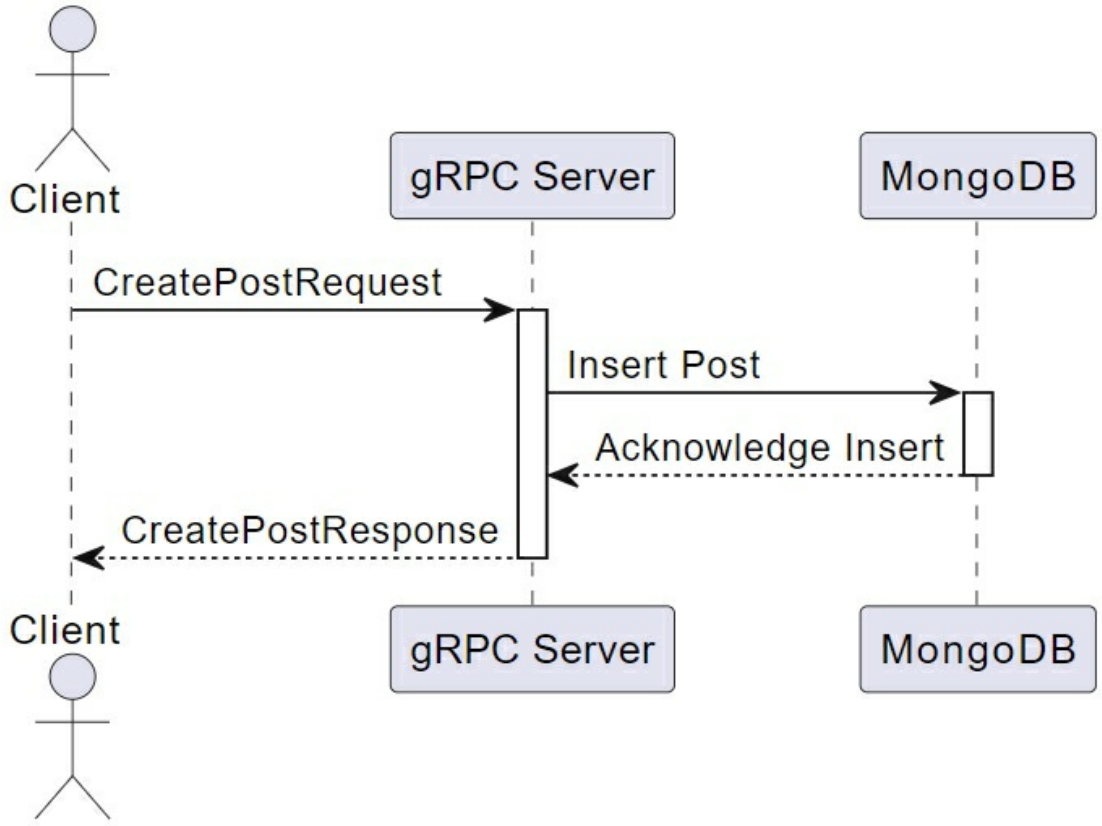


Fig 9.3 Integration Testing between MongoDB and gRPC

Set up Test Environment

The first thing you need is a testing environment, this could be a separate database specifically for testing or an in-memory database.

```
#include <gtest/gtest.h>
#include <grpcpp/grpcpp.h>
#include "blog.grpc.pb.h" // the generated protobuf file
class IntegrationTest : public ::testing::Test {
```

```
protected:
  void SetUp() override {
      // Setup goes here.
      // This could include connecting to your MongoDB database.
      // Connecting to your gRPC server.
  }
  void TearDown() override {
      // Cleanup goes here.
      // This could include disconnecting from your MongoDB database and gRPC server.
  }
};
```

Implement Tests

You could now implement tests to verify that the gRPC server and MongoDB database work together as expected.

```
TEST_F(IntegrationTest, CreatePost) {
   // Create a new post request
  blog::CreatePostRequest request;
  blog::Post* post = request.mutable_post();
  post->set_title("Test Title");
  post->set_content("Test Content");
  // Make the gRPC call
  grpc::ClientContext context;
  blog::CreatePostResponse response;
   grpc::Status status = stub_->CreatePost(&context, request,
```

```
&response);
    // Check the status of the gRPC call
   ASSERT_TRUE(status.ok());
    // Check that the post was created in the database
    // This will depend on how your MongoDB is set up.
    // You could query the MongoDB database to check if the new post exists.
}
// You could implement similar tests for updating, deleting, and reading posts.
```

Run Tests

Once your CMakeLists.txt file is set up, you can use CMake to generate the build files. This is typically done in a separate build directory to keep your project organized. The command 'cmake ..' is used to generate these files, with the '..' indicating that CMake should look for a CMakeLists.txt file in the parent directory. After generating the build files, you can use Make to compile your code and build your project. The 'make' command will build all targets specified in your CMakeLists.txt file. If your CMakeLists.txt file is set up correctly, this should include building your tests. To run your tests, you would typically use a command like './run_tests', where 'run_tests' is the name of the test executable created by your CMakeLists.txt file. This will run all of your tests and output the results to the console.

This is a simplified example, and the exact details may vary depending on how your project is set up. For example, your project may have different dependencies, or you may organize your source files differently. However, the overall process of writing tests, building your project with CMake and Make, and running your tests should be similar.

Continuous Testing and Test Automation

Continuous testing is an integral part of the modern software development process. The objective is to test early and test often, so that issues can be detected and resolved as quickly as possible. It typically goes hand-in-hand with Continuous Integration (CI) and Continuous Deployment (CD) to form the CI/CD pipeline.

Continuous testing involves automating the testing process so that tests are run each time a change is made to the codebase. This can catch bugs before they get deployed to production and ensure that your application is always in a deployable state.

Sample Program to Setup Continuous Testing

Let us use an example of GitHub Actions, which provides a way to automate software workflows, including testing, right from GitHub. The given below walkthrough is how you could set up continuous testing with GitHub Actions for your C++ project:

Create a new workflow

In your repository, create a new file in the .github/workflows directory to define the workflow. You might name it test.yml.

Define the workflow

The workflow is defined using YAML syntax. At a high level, you'll specify when the workflow should be triggered, what environment it should run in, and what steps it should take.

A basic workflow might look like this:

```
name: C++ CI
on:
 push:
   branches: [ master ]
 pull_request:
   branches: [ master ]
jobs:
 build:
   runs-on: ubuntu-latest
   steps:
```

```
- uses: actions/checkout@v2
- name: install dependencies
  run: sudo apt-get install libgtest-dev cmake -y
- name: build
  run: cmake .
- name: test
   run: make && ./test_app # or however you run your tests
```

This workflow is triggered whenever a commit is pushed or a pull request is opened against the master branch. It runs on an Ubuntu environment.

The steps are as follows:

- Checkout the code from the current repository.
- Install the necessary dependencies (Google Test and CMake).
- Build the application with CMake.
- Run the tests with make and the test executable.

Once you've defined this workflow, tests will be automatically run each time a commit is pushed to the master branch or a pull request is opened against it. If a test fails, you'll be notified, and you can look at the logs to see what went wrong. Remember, the main goal of continuous testing is to catch issues as early as possible and ensure your application is always in a deployable state. With this setup, you can be confident that your tests will always be run, and that issues will be caught before they make it to production.

Troubleshooting and Solutions

Let us explore some common issues and solutions associated with testing in a C++ environment, specifically with Google Test and continuous testing setup.

Google Test Compilation Issues

You might encounter errors while trying to compile Google Test or your test code. Verify you've installed all necessary dependencies and you're including the correct header files. If you're using an older version of C++, ensure it's compatible with your version of Google Test. Always look at the error message details; they often provide valuable clues about what's wrong.

Tests Failing Unexpectedly

If tests that you expect to pass are failing, start by examining the error messages and test output. Try to isolate the issue by running a subset of your tests or simplifying your test code. Make sure you've correctly set up any necessary test data or mock objects.

Flaky Tests

Sometimes, tests may pass or fail inconsistently. This could be due to race conditions, reliance on system state, or non-deterministic factors such as random number generation. Try to make your tests as deterministic as possible, and isolate them from factors you can't control. Use mock objects or fixed seeds for random number generation, if necessary.

Troublesome with Setting up Continuous Testing

If you're having problems setting up a continuous testing environment with GitHub Actions, start by checking the GitHub Actions documentation and looking at example workflows. Make sure your workflow file is in the correct location and has the correct syntax. You can check the Actions tab on your GitHub repository to see the results of running workflows and any error messages.

GitHub Actions unable to Locate Files/Directories

Remember that GitHub Actions checks out your code in a specific directory (/github/workspace). If you're referencing files or directories in your tests or workflows, make sure the paths are correct.

Trouble with GitHub Actions Secrets

If you're using secrets in your workflows (e.g., for deployments), make sure you've correctly set them up in the repository settings, and that you're correctly referencing them in your workflows. Don't hardcode sensitive information in your workflow files!

Summary

This chapter focused on the important function that testing plays in back-end development as well as the many different forms that testing can take. Each of the three types of testing—unit testing, integration testing, and end-to-end testing—plays an important part in ensuring that individual software components, interactions between components, and overall system functionality all work as designed. Testing not only ensures that the code is correct, but it also helps prevent errors from occurring again, facilitates refactoring, and acts as a form of documentation. The introduction to C++ testing frameworks such as Google Test and Doctest shed light on the ease of use as well as the benefits brought to applications written in C++ by these tools.

We dove deep into the particulars of installing and utilizing Google Test in a C++ environment, as well as the process of writing tests to prevent data loss and performing integration testing with MongoDB and gRPC. We gained an understanding of the more practical aspects of using Google Test and witnessed firsthand the robustness and versatility of the tool through the process of implementing test cases and running them. We went through the steps of putting together an integration testing environment for a blog application. Throughout the process, we emphasized how important it is to conduct tests in an environment that is as similar as possible to the production setting.

In the last section of the chapter, we discussed the idea of continuous testing as well as the process of setting it up. Continuous testing ensures that the code is validated each time a change is pushed to the codebase, which enables problems to be discovered at an earlier stage and facilitates the development of a robust development pipeline. The most common troubleshooting scenarios were investigated, and useful insights into fixing problems with test compilation, unexpected test failures, flaky tests, and difficulties in setting up continuous testing environments were provided. In addition, the process of troubleshooting and finding a solution was covered, with an emphasis placed on the significance of these skills in the context of the landscape of real-world software development.

CHAPTER 10: SECURING YOUR C++ BACKEND

Backend Security Overview

When it comes to backend development, security is of the utmost importance. It helps protect sensitive data and ensures that your application continues to function without any problems. The role that backend developers play in ensuring application security is more important than it has ever been in this day and age, when data breaches and cyberattacks are becoming increasingly commonplace. The backend of an application serves as the primary safeguard for the user data and the overall integrity of the application. Backend security breaches can have devastating effects, ranging from identity theft and fraud for users to financial loss and irreparable damage to reputation for companies. These effects can be devastating for both parties. As a result, the process of securing the application's backend ought to be a top priority for every organization.

The concept of confidentiality, integrity, and availability is colloquially referred to as the CIA triad in the field of cybersecurity. These three fundamental facets are at the center of backend security. A commitment to confidentiality ensures that sensitive information is only viewable by those who have been given permission to do so. Integrity guarantees that the data will continue to be true and reliable throughout the entirety of its lifecycle. Accessibility guarantees that the systems will always be reachable by those who require them at the exact moment they are required.

The database constitutes one of the most fundamental parts of the backend security system. The backend developers are responsible for implementing access controls in the correct manner in order to restrict who can view or modify the data. SQL injection is a common form of attack that involves the manipulation of input by an attacker in order to execute arbitrary SQL commands. Attacks using cross-site scripting (XSS) and cross-site request forgery (CSRF), which are directed primarily at end users but can have severe repercussions for backend systems, are another source of concern.

Authentication and authorization are also crucial components of backend security. Authentication is the process of confirming a user's identity, while authorization is the process of deciding what actions an authenticated user is permitted to take. It is possible for unauthorized users to gain access to sensitive data and functionalities if authentication and authorization are not implemented correctly.

Backend development also involves another major concern, which is API security. Insecure application programming interfaces (APIs) can result in data breaches as well as unauthorized access to backend systems. To ensure the safety of application programming interfaces (APIs), developers should employ encrypted communication protocols (such as HTTPS), authenticate and authorize API requests, and validate and sanitize input data.

The security of the server is also very important. This involves securing the configuration of the server, maintaining the most recent version of the software, and monitoring for any unusual activity.

It is necessary to give careful consideration to the issue of security throughout the entirety of the backend, from the database to the server. Backend developers have a significant part to play in the process of putting these security measures into action, which not only ensures the proper

functioning of the application but also the users' well-being and continued faith in it.

Database Security

Database security is a broad and complex domain that forms a critical part of backend application development. The necessity of robust database security cannot be overstated as it serves to protect the most valuable asset of modern businesses: data. As databases often store sensitive and proprietary information - customer details, transaction records, financial information - they are an attractive target for cybercriminals. If a database is compromised, the consequences can range from data loss and legal repercussions to irreversible damage to a company's reputation.

There are numerous threats to database security. Unauthorized access is a significant concern, with attacks ranging from internal employees with excessive privileges to external hackers breaching perimeter defenses. This can lead to data theft, corruption, or even complete deletion of records. Injection attacks, such as SQL injection, pose another serious risk. The maliciously crafted user input is used to manipulate database queries, allowing unauthorized viewing or manipulation of data.

Databases can also suffer from inadequate encryption. If data is not encrypted properly, or at all, it can be read by anyone who gains access to it. This is particularly risky for data in transit or stored in backups. Another aspect of database security is the risk of data leaks. A misconfigured database can inadvertently expose sensitive information on the internet, making it easy pickings for cybercriminals. Even the hardware where the database resides can be a security concern. Physical attacks, natural disasters, and hardware failures can all lead to data loss. Although these issues might seem less 'cyber' in nature, they fall under the broader umbrella of database security, necessitating measures like physical security and regular data backups. Beyond these threats, databases can be vulnerable due to outdated software. Older versions of database management systems can have known security flaws that attackers can exploit. Regular patching and updates are vital to mitigate this risk.

In the era of regulations like GDPR and CCPA, database security also has legal implications. Failing to adequately protect user data can result in hefty fines and legal sanctions, making database security a compliance issue as well. As such, a multifaceted approach to database security is needed. This includes stringent access controls, effective encryption, regular software updates, secure configurations, robust physical security, and compliance with relevant laws and regulations. In essence, every step must be taken to turn the database from a potential security weak point into a strong, resilient part of an application's defense.

Database security is a vital aspect of backend development, involving the safeguarding of data from threats ranging from cyber attacks to physical disasters. By implementing robust database security measures, companies can protect their valuable data, maintain user trust, and ensure business continuity, while also complying with legal requirements.

Secure MongoDB Database

Securing your MongoDB database involves a few crucial steps that can significantly enhance the security of your data. We will use the blog application as an example to demonstrate these steps:

Enable Access Control

By default, MongoDB runs without access control, which means anyone with access to the server can access all databases and perform any operation. We need to enable access control and define users who can access the database.

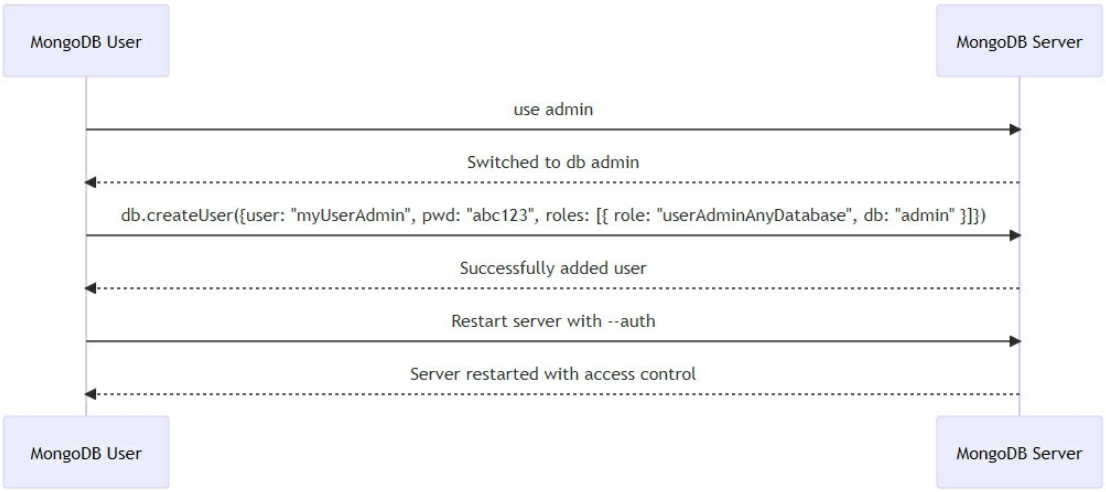


Fig 10.1 Enabling Access Control in MongoDB

Following is the syntax:

```
# Switch to the admin database
> use admin
# Create a new user
> db.createUser(
 {
  user: "myUserAdmin",
  pwd: "abc123",
   roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
 }
```

```
)
```

This command creates a user myUserAdmin in the admin database with a password abc123. The user is granted the userAdminAnyDatabase role, which allows them to create and manage users on any database.

As a next step, restart the MongoDB server with access control enabled:

```
mongod --auth
```

Enforce Authentication

After enabling access control, clients must authenticate themselves to connect to your MongoDB server. Use the following command to authenticate:

```
mongo --username myUserAdmin --password abc123 --
authenticationDatabase admin
```

Use TLS/SSL

Configure MongoDB to use Transport Layer Security (TLS) or its predecessor, Secure Sockets Layer (SSL), for all incoming and outgoing connections. It protects data while it is "in transit" - moving from one location to another. You'll need to generate (or purchase) a TLS/SSL certificate, then start the server with the --tlsMode and --tlsCertificateKeyFile options:

```
mongod --tlsMode requireTLS --tlsCertificateKeyFile
/etc/ssl/mongodb.pem
```

Role-based Access Control

Provide each user only the privileges that they need. If a user only needs to read documents, don't give them the ability to write to the database. In our blog app, for instance, you might have a "writer" role that can add and edit posts, and a "reader" role that can only view posts.

```
# Switch to the blog database
> use blog
# Create a writer
> db.createUser(
 {
   user: "writer",
```

```
  pwd: "abc123",
   roles: [ { role: "readWrite", db: "blog" } ]
 }
)
# Create a reader
> db.createUser(
 {
  user: "reader",
  pwd: "abc123",
   roles: [ { role: "read", db: "blog" } ]
 }
)
```

Encrypted Storage Engine

In order to ensure the security of your MongoDB database, it's crucial to utilize the encrypted storage engine provided by MongoDB. The WiredTiger storage engine offers encryption at rest, which means that your data is encrypted while it's stored on the disk in a non-operational state. However, it's important to note that this feature is only available in MongoDB Enterprise.

Regular Updates

In addition to using encrypted storage, it's also essential to regularly update your MongoDB server. By doing so, you can ensure that you're benefiting from the latest security enhancements and patches. Regular updates can help to protect your database from potential vulnerabilities and threats.

Network Exposure

Limiting network exposure is yet another crucial component of the MongoDB security system. It is not recommended that your MongoDB server be directly accessible via the internet. Instead, it should be protected by a firewall, and only the ports that are absolutely necessary should be left open. This could assist in preventing unauthorized users from accessing your database. You can restrict connections to only a certain set of IP addresses by configuring the bindIp option in the mongodb.conf file that you use. For instance, if you set bindIp to 127.0.0.1, you'll only be able to accept connections from the localhost.

```
net:
  bindIp: 127.0.0.1
```

Limiting network exposure is yet another crucial component of the MongoDB security system. It is not recommended that your MongoDB server be directly accessible via the internet. Instead, it should be protected by a firewall, and only the ports that are absolutely necessary should be left open. This could assist in preventing unauthorized users from accessing your database. You can restrict connections to only a certain set of IP addresses by configuring the bindIp option in the mongodb.conf file that you use. For instance, if you set bindIp to 127.0.0.1, you'll only be able to accept connections from the localhost.

User Authentication and Authorization

Securing a gRPC service involves several steps that revolve around authentication and encryption. Below, I'll explain how to add security to gRPC in the context of our C++ blog application.

SSL/TLS Encryption

The first step in securing gRPC is to use SSL/TLS encryption. It ensures that the data transmitted between the client and the server is private and integral. The given below walkthrough is how to do it:

First, generate the certificate and private key files:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

Then, on the server side, create a grpc::SslServerCredentials object and use it when starting the server:

```
std::string server_address("0.0.0.0:50051");
grpc::SslServerCredentialsOptions::PemKeyCertPair key_cert_pair = {read_key("key.pem"), read_cert("cert.pem")};
grpc::SslServerCredentialsOptions ssl_options;
ssl_options.pem_key_cert_pairs.push_back(key_cert_pair);
std::shared_ptr<grpc::ServerCredentials> creds = grpc::SslServerCredentials(ssl_options);
BlogServiceImpl service;
grpc::ServerBuilder builder;
builder.AddListeningPort(server_address, creds);
builder.RegisterService(&service);
std::unique_ptr<grpc::Server> server(builder.BuildAndStart());
```

On the client side, create a grpc::SslCredentials object and use it when creating the stub:

```
std::shared_ptr<grpc::ChannelCredentials> creds =
```

```
grpc::SslCredentials(grpc::SslCredentialsOptions());
auto channel = grpc::CreateChannel("localhost:50051", creds);
auto stub = BlogService::NewStub(channel);
```

Token-Based Authentication

Token-based authentication, such as JWT (JSON Web Tokens) or OAuth, is a crucial aspect of backend security. This method of authentication allows the server to verify the identity of the client, ensuring that only authorized users can access certain resources.

The process begins when the client logs in, typically through an HTTP-based login API. Once the client's credentials are verified, the server returns a token. This token is a unique string of characters that represents the client's session. The client then includes this token in the metadata of every subsequent gRPC call, effectively proving their identity with each request.

Following is an example of how this might look on the client side:

```
grpc::ClientContext context;
context.AddMetadata("authorization", "Bearer " + token);
```

In this code snippet, the client creates a new gRPC context and adds the token to the context's metadata. The key "authorization" is used to indicate that the value is an authorization token.

On the server side, you can intercept incoming calls to check for the token. Given below is an example of how this might look:

```
class AuthInterceptor : public grpc::experimental::Interceptor {
public:
  void Intercept(grpc::experimental::InterceptorContext* context)
override {
   auto metadata = context->client_metadata();
   auto it = metadata.find("authorization");
   if (it == metadata.end()) {
     context->Finish(grpc::StatusCode::UNAUTHENTICATED, "No
token provided");
     return;
```

```
  }
  std::string token = it->second;
  // Validate the token...
 }
};
```

In this code snippet, the server intercepts the incoming call and checks the client's metadata for an authorization token. If no token is found, the server ends the context with an "UNAUTHENTICATED" status code and a message indicating that no token was provided.

If a token is found, the server then validates the token. This usually involves checking the token's signature and expiry time to ensure that it's valid and hasn't expired. If the token is valid, the server can then process the client's request. If the token is invalid or expired, the server can reject the request and return an appropriate error message.

It's important to note that token-based authentication does not replace SSL/TLS. SSL/TLS is a protocol for encrypting network connections, and it's crucial for ensuring that the token and other sensitive data are not leaked in transit. Even with token-based authentication, you should still use SSL/TLS to secure your connections and protect your data.

Network Security

Network security is a critical aspect of backend security that cannot be overlooked. It involves implementing measures to prevent unauthorized access, misuse, modification, or denial of network-accessible resources. This is especially important for servers that are exposed to the internet, as they are prime targets for cyber attacks.

One of the most basic and effective measures for network security is the use of a firewall. A firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules. It establishes a barrier between a trusted internal network and an untrusted external network, such as the internet. By blocking unauthorized access, it protects your server and the data it holds. However, having a firewall is not enough. It's also important to ensure that only the necessary ports are open on your server. Each open port represents a potential point of entry for attackers, so minimizing the number of open ports can significantly reduce your server's vulnerability.

In addition to controlling which ports are open, you should also limit your server to only accept connections from trusted IP addresses. This can be achieved through IP whitelisting, which is a security feature often used for limiting and controlling access only to trusted users. IP whitelisting allows you to create lists of trusted IP addresses or IP ranges from which your users can access your server.

Following is an example of how you might configure your server to only accept connections

from certain IP addresses:

```
net:
  bindIp: 192.0.2.1,203.0.113.1
```

In the above sample program, the server is configured to only accept connections from the IP addresses 192.0.2.1 and 203.0.113.1. Any connection attempts from other IP addresses will be rejected. It's important to note that network security is a broad field that goes beyond the scope of gRPC or C++. It involves a wide range of practices and technologies, including but not limited to intrusion detection systems (IDS), intrusion prevention systems (IPS), and secure network architectures.

Adding Security to Web Servers

Adding security to a web server like Nginx involves many steps to ensure secure communication, authentication, authorization, and protection against common web attacks. The given below walkthrough is how you can secure your Nginx setup for our blog application:

HTTPS

As we discussed earlier, using SSL/TLS to set up HTTPS is one of the first steps to secure your web server. It encrypts the traffic between the client and the server. We've already covered how to set this up in a previous section.

HTTP Security Headers

HTTP security headers provide another layer of security by helping to mitigate attacks and security vulnerabilities. Some important headers include:

- X-Content-Type-Options: Prevents the browser from MIME-sniffing a response away from the declared content-type.
- X-Frame-Options: Protects your visitors against clickjacking attacks.
- X-XSS-Protection: This header enables the Cross-site scripting (XSS) filter in your browser.

Add these headers to Nginx by updating your configuration file (usually located at /etc/nginx/nginx.conf or /etc/nginx/sites-available/default):

```
server {
  ...
  add_header X-Content-Type-Options nosniff;
  add_header X-Frame-Options SAMEORIGIN;
  add_header X-XSS-Protection "1; mode=block";
  ...
}
```

Remember to reload or restart Nginx to apply the changes.

Rate Limiting

Rate limiting is a technique for limiting network traffic. It sets a limit on how often a client can repeat a specific request within a certain timeframe. In Nginx, you can set up rate limiting with the limit_req module.

Following is a sample code snippet:

```
http {
  ...
  limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;

  server {
    ...
    location /login {
      limit_req zone=mylimit burst=20 nodelay;
      ...
    }
  }
}
```

This configuration limits requests to the /login location to 10 requests per second. If a client exceeds the limit, Nginx responds with a 503 Service Unavailable error.

Hide Nginx Version Number

By default, Nginx will include its version number in the HTTP response headers and error pages. This information can be useful for attackers. To hide the version number, add the following line to your configuration file:

```
server_tokens off;
```

Use Web Application Firewall (WAF)

A Web Application Firewall (WAF) is a critical component of any web server security strategy. It acts as a shield between your web application and the internet, examining incoming traffic to block any malicious requests. This can protect your application from common web attacks such as SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF).

Nginx offers a commercial WAF module that integrates directly with the server. However, there are also free and open-source options available, such as ModSecurity. ModSecurity is a widely used WAF that can be configured to detect and block a wide range of attacks. However, using a

WAF is just one part of a comprehensive security strategy. It's also crucial to keep your server and all associated software up to date. Regular updates not only provide new features and performance improvements but also patch security vulnerabilities that could be exploited by attackers. In addition to this, regular monitoring of server logs is essential. Logs can provide valuable information about the server's operation and any potential security incidents. If an incident does occur, a swift response is crucial to minimize the impact and prevent further damage.

Summary

In this chapter, we discussed the significance of as well as the requirement to implement security measures at a variety of levels within the backend infrastructure. After providing an overview of the requirements for security, we proceeded to investigate the potential flaws that could be found in each layer of a backend system and the significance of adopting a multi-pronged strategy for securing the system.

We went into depth about the importance of database security, primarily illustrating our points with MongoDB as a reference. We gained knowledge on how to protect our MongoDB database from being accessed by unauthorized parties by implementing user authentication and role-based authorization. In addition, we brought attention to the importance of performing regular patching and updates, IP whitelisting, and encrypting data both while it is stored and while it is in transit. The following section concentrated on gRPC security, during which we investigated various authentication and access control methods, including token-based authentication, role-based access control, and SSL/TLS for encrypted communication.

The importance of web server security was emphasized in the concluding section, with Nginx serving as an example. We investigated a variety of methods for securing a Nginx server, such as configuring HTTPS for encrypted communication, adding HTTP security headers to protect against common web attacks, implementing rate limiting to protect against brute-force and DoS attacks, hiding the Nginx version number to avoid exposing potential vulnerabilities, and utilizing a Web Application Firewall (WAF) to safeguard against a variety of web-based threats. The significance of implementing a comprehensive security strategy that includes routinely monitoring logs, updating software and systems, and promptly reacting to security incidents was emphasized throughout this chapter.

CHAPTER 11: DEPLOYING YOUR APPLICATION

Deployment Overview

In this chapter, we shall look into the exciting world of deployment - the final step in the development process where your application, meticulously designed and carefully tested, becomes available to users. Deployment involves configuring, hosting, maintaining, and making an application accessible to end-users. To start with, the process of deployment involves several stages: pre-production (where we ensure the codebase is final and ready for production), compilation (turning source code into machine code), packaging (gathering all the necessary components into a distributable format), installation (putting the package onto the server), and activation (starting the program).

The deployment process begins with the pre-production stage. The development team ensures that the codebase is finalized and ready for production. This involves a thorough review of the code, final testing, and approval from all stakeholders. It's crucial to ensure that all features are implemented as per the requirements, all bugs are addressed, and the application is fully functional and ready for use.

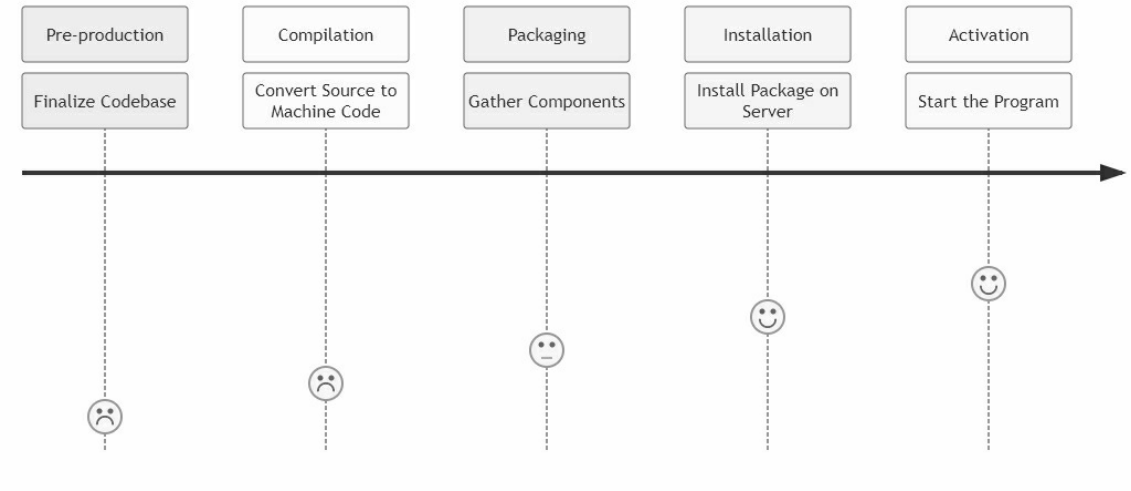


Fig 11.1 Phases of Deployment

Following pre-production is the compilation stage. Compilation is the process of translating the source code written by developers into machine code. This machine code is a low-level code that can be directly executed by the computer's processor. The compiler checks the source code for any syntactic or semantic errors and, if it finds none, translates it into machine code. In the context of languages like C++, this is a crucial step. However, for interpreted languages like Python or JavaScript, this step is handled at runtime.

After successful compilation, the next step is packaging. Packaging involves gathering all the necessary components of the application, including the compiled code, libraries, dependencies, and resources, into a distributable format. This package, often in the form of a .zip file or a Docker container, contains everything needed to install and run the application on a server. It's crucial to ensure that the package includes all the necessary components and that they are compatible with the target environment.

The installation stage follows packaging. During installation, the application package is

transferred onto the server where it will be hosted. This could be a physical server in a data center, a virtual machine in a cloud environment, or a container in a Kubernetes cluster. The installation process involves extracting the package, placing the files in the appropriate directories, setting up the necessary environment variables, and configuring the server to run the application.

The final stage of deployment is activation. Activation involves starting the application and making it available to users. This could involve starting a server, such as Apache or Nginx, that delivers the application to users' browsers. It could also involve starting a background service or daemon that keeps the application running continuously. Once the application is activated, it should be accessible to users via the internet. Even after deployment, the ops team maintains and supports the application in production. Monitoring, logging, and alerting ensure uptime and performance. Issues get triaged and debugged with developers. Future updates begin the deployment process anew. Robust deployment practices ensure applications reliably and securely transition from codebase to powering production systems. Streamlined deployment capabilities allow development velocity to accelerate.

Deployment Strategies

Deployment strategies are primarily based on how we manage the transition from the previous version of an application to the new one. The most common strategies are "Recreate" (all at once), "Ramped" (also known as rolling update or incremental), "Blue/Green", "Canary", and "A/B Testing". The Recreate strategy involves shutting down the previous version and then deploying the new one. Ramped deployment rolls out the update to a small set of users and gradually increases the rollout percentage as confidence in the update grows. Blue/Green deployment has two identical environments, where one is live (Green) and the other is idle (Blue). Canary deployment is similar to ramped but offers more control over the percentage of users. A/B Testing directs a small percentage of users to the new version, while the rest use the old one, allowing for a comparison of performance.

Deciding on a strategy requires considering downtime acceptance, target audience, robustness of testing, the complexity of the application, and the team's deployment expertise. For example, Blue/Green is more complex but offers immediate rollback if needed. Canary and A/B Testing can provide valuable user feedback, but might be overkill for small, internal applications.

On top of this, various deployment platforms can assist with this process, ranging from traditional servers to modern cloud services like AWS, Google Cloud, and Azure, to platform-as-a-service (PaaS) options like Heroku and Netlify. Depending on the application's requirements and the resources available, different platforms may be more appropriate.

The crux of deployment is a successful transition from a development environment to a production environment, and this necessitates thorough planning and consideration of various factors. In the following sections, we shall get practical, and look into deploying our blog application, leveraging the concepts we've discussed so far.

Deployment Preparation

The stage of preparing for deployment is a pivotal phase in the software development lifecycle. It's the juncture where you ascertain that your application is thoroughly primed and polished for usage by the public. This stage is not to be taken lightly, as it sets the stage for the user experience and the overall reception of your application. The upcoming sections will look into the critical elements that ought to be part of your comprehensive deployment checklist, ensuring that no stone is left unturned in your quest for a successful and smooth deployment. Following is the best checklist you may refer to:

Environment Setup

It is crucial that you create a setup that mirrors your production environment. This means you should be using the same operating systems, the same versions of your programming language interpreter/compiler, the same versions of your dependencies, and so on. It's also important to think about the kind of load your application is likely to experience, as this could influence the type of servers you need.

Configuration Management

Ensure that the application configuration is abstracted from your code. It should be managed via environment variables or configuration files which can be modified according to the deployment environment. In our blog application, database connections, server host, and port number are examples of configurations that might change between environments.

Version Control

A good deployment process includes version control not just for the code, but also for the database schema, configuration files, and even for the deployment process itself. A change log that tracks the version history of the application should be maintained.

Database Migration

Ensure that any changes to your database schema can be applied in a predictable and repeatable way. In the context of our blog application, if we make changes to the schema that stores blog posts, these changes should be applied to the production database in a controlled manner.

Backup and Recovery

Before deploying, it's vital to set up a backup and recovery strategy. In the event of a failure, you should have a clear procedure to restore your application data.

Performance

Make sure your application has been thoroughly tested for performance and can handle the expected load. This could involve stress testing (evaluating the system under extreme loads) and load testing (evaluating it under expected loads).

Security

Security should be a top priority when deploying your application. Sensitive data, like user

passwords in our blog application, should be handled securely. Encryption for sensitive data, secure data transmission protocols, and protection against common web attacks should all be considered.

Monitoring and Alerts

It's important to know how your application is performing in production. Implement a monitoring system that alerts you when things go wrong, and can provide you with information to help debug the issue.

Document Everything

Finally, documentation can be a lifesaver. While doing this, ensure you document your deployment process, not just for yourself, but for anyone else who might need to understand it.

You can help ensure that your deployment goes without a hitch and that your application runs smoothly in production if you pay attention to each of these aspects and address them accordingly. It is important to keep in mind that this is not a one-time event; every time you get ready for a deployment, you should run through this checklist to ensure that everything is in order.

Sample Program: Deploying Blog Application on AWS

Deploying your C++ backend application on Amazon Web Services (AWS) encompasses a series of steps and the utilization of multiple AWS services. In this context, we will primarily focus on AWS Elastic Beanstalk, a robust and user-friendly service provided by AWS. Elastic Beanstalk is designed to simplify the deployment process by abstracting many of the underlying details, thereby enabling developers to launch their applications swiftly and efficiently.

This service is particularly beneficial for developers as it automates the intricate process of application setup, capacity provisioning, load balancing, auto-scaling, and application health monitoring. With Elastic Beanstalk, you can quickly deploy and manage applications in the AWS cloud without worrying about the infrastructure that runs those applications. This leaves you free to focus on writing your C++ code while AWS handles the deployment details.

Following is a step-by-step walkthrough on how you can do it:

Preparing Application

Before you can deploy your application, you need to package it in a way that AWS can understand. This usually means creating a zip file or a Docker container containing your application's code and any associated dependencies.

In our case, since our C++ application is built using gRPC, MongoDB, and Nginx, we will use Docker to package everything together. The Dockerfile could look something like this:

```
# Use an official C++ runtime as a parent image
FROM gcc:9.2.0
# Set the working directory in the container to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
ADD . /app
# Install any needed packages specified in requirements.txt
RUN apt-get update && \
    apt-get install -y libgrpc++-dev protobuf-compiler && \
   make
# Make port 80 available to the world outside this container
```

```
EXPOSE 80

# Run the app when the container launches

CMD ["./myapp"]
```

This Dockerfile specifies that the parent image is gcc:9.2.0 (a C++ runtime), sets the working directory to /app, copies the current directory (your C++ code) into the Docker image, installs the necessary libraries, exposes port 80 for the web server to use, and then runs your application.

Creating Elastic Beanstalk Application

Creating an Elastic Beanstalk Application involves a few straightforward steps. First, you need to access the AWS Management Console, which serves as the central hub for managing all your AWS services. From there, you'll need to open the Elastic Beanstalk console, which is specifically designed for application deployment and management.

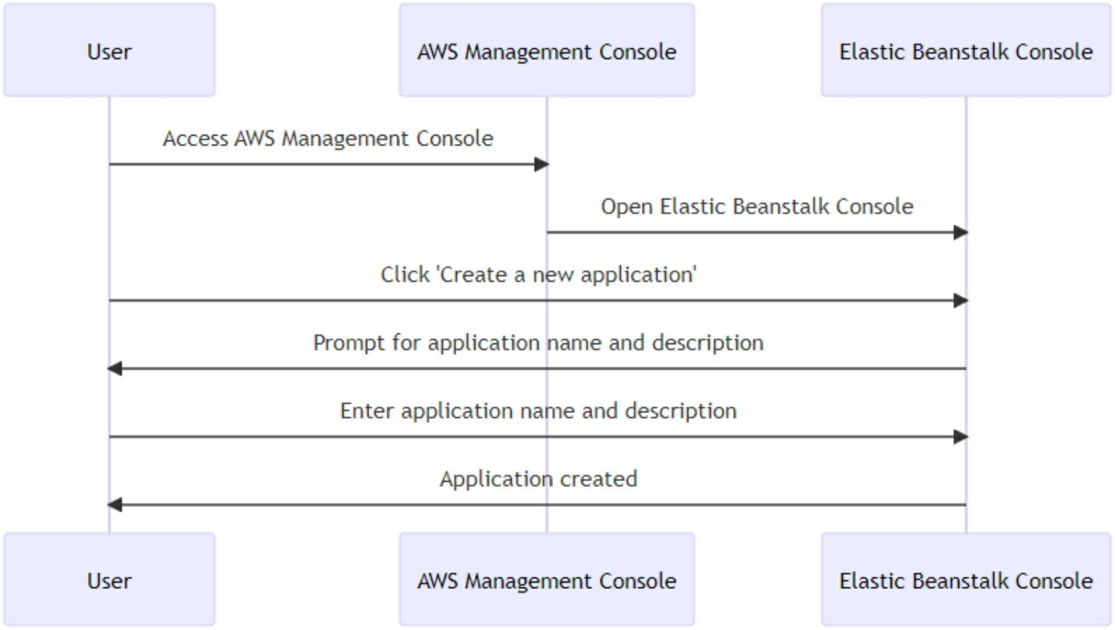


Fig 11.2 Accessing Elastic Beanstalk using AWS

Once you're in the Elastic Beanstalk console, you'll see an option to "Create a new application." Clicking on this will prompt you to provide a name for your application, which should be unique and descriptive to help you identify it among other applications. You'll also be asked to provide a brief description of your application. This description is optional but can be helpful in providing additional context about the application's purpose, its key features, or any other relevant information.

Creating Environment

Once you've successfully created an application, the next step involves setting up an

environment. In the context of AWS, an environment functions like a virtual server, providing the necessary infrastructure for your application to run.

To create an environment, navigate to the "Environments" section and select the "Create one now" option. In the subsequent options, choose the "Web server environment" and then proceed by clicking on "Create".

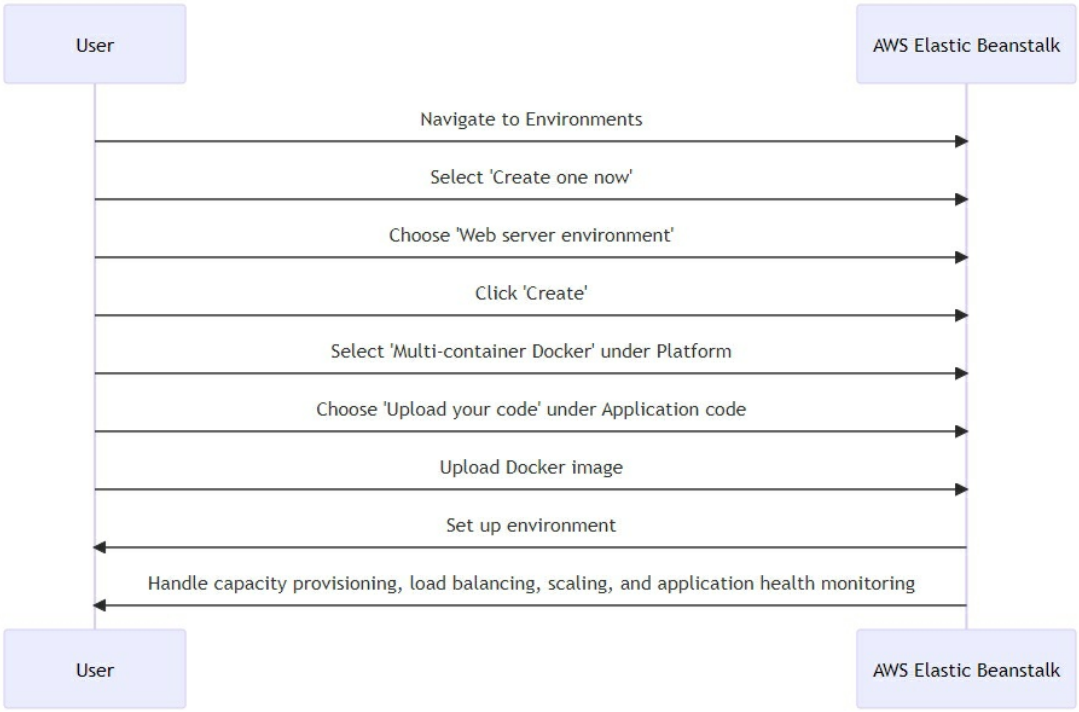


Fig 11.3 Creation of Environment

In the ensuing page, you'll be required to specify your platform and application code. Under the "Platform" section, select "Multi-container Docker" from the dropdown menu. This selection indicates that you intend to use Docker, a popular platform used for automating the deployment, scaling, and management of applications.

Next, under the "Application code" section, opt for the "Upload your code" choice. This allows you to upload the Docker image you created in the initial step. This Docker image encapsulates your application, providing a self-sufficient environment that includes everything your application needs to run, including the code, a runtime, libraries, environment variables, and config files.

Configuring Environment

The configuration of your environment is a crucial step in the deployment process. This is where you set up the necessary parameters that your application needs to function correctly. To do this, navigate to the "Configuration" option located on the sidebar of your AWS Elastic Beanstalk dashboard.

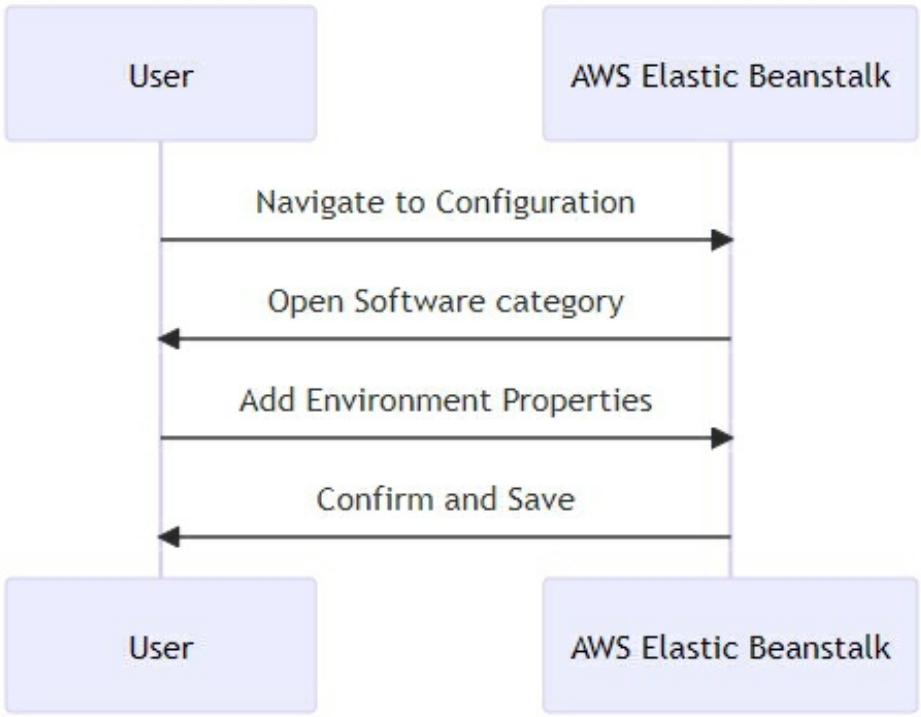


Fig 11.4 Configuration in Beanstalk Dashboard

Once you're in the Configuration section, look for the "Software" category. This is where you can specify the environment properties for your application. These properties are essentially key-value pairs that represent environment variables your application will use.

For instance, if you're deploying a blog application that uses MongoDB as its database, you would need to provide the MongoDB connection string as an environment property. This string is crucial as it allows your application to connect and interact with the database.

Deploying Application

Deploying your application is the final step in the process. Once your environment is correctly configured, you can proceed by clicking on the "Upload and Deploy" button. This will prompt you to select the zip file or Docker image that you created in the earlier steps. After selecting the appropriate file, click on the "Deploy" button. AWS Elastic Beanstalk will now take over and handle the rest of the process.

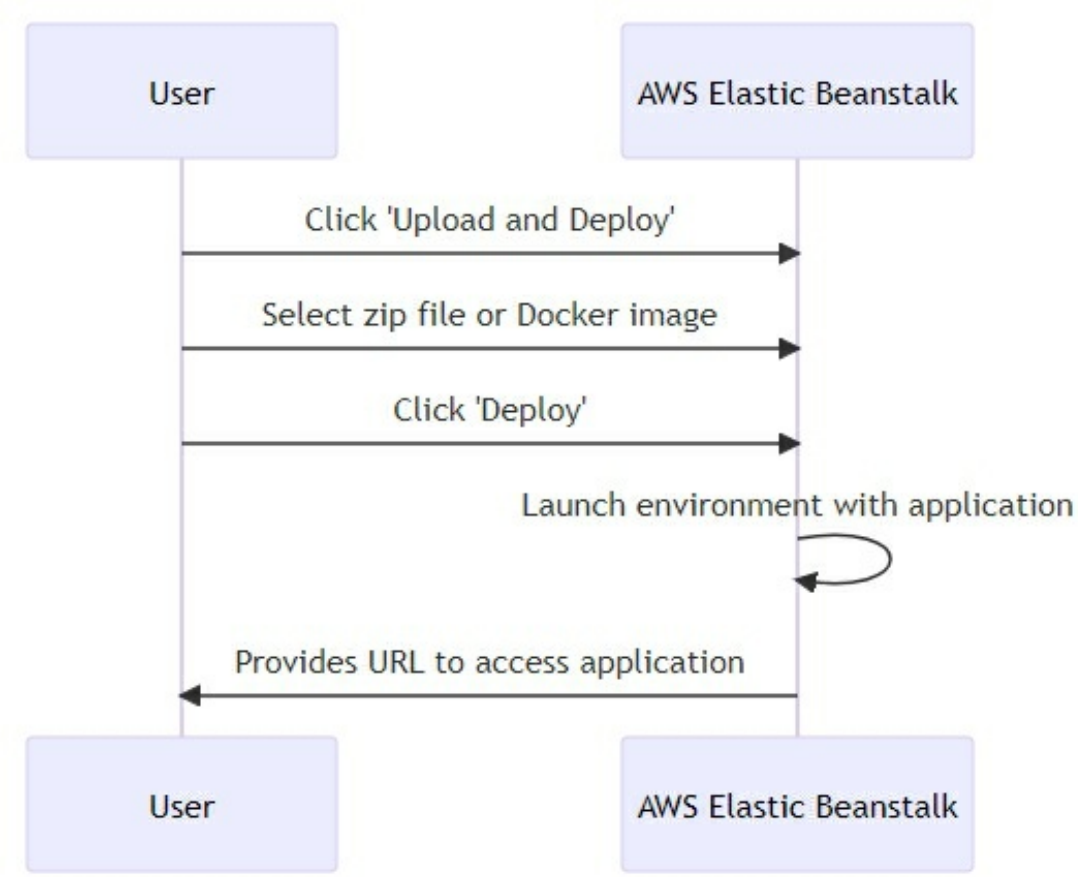


Fig 11.5 Application Deployment

During this phase, AWS will launch an environment that contains a running instance of your application. This environment is essentially a virtual server configured specifically for your application. Once the deployment is successful, AWS will provide a URL through which you can access your application. This URL is displayed prominently in the Elastic Beanstalk console.

This marks the completion of your application's deployment. Your application, which was once confined to your local development environment, is now hosted on a robust and scalable infrastructure, ready to serve users from around the world.

Setting up Continuous Deployment

AWS Elastic Beanstalk integrates with AWS CodePipeline for continuous deployment. With CodePipeline, you can set up a workflow so that every time you push a change to your code repository (like GitHub), AWS automatically deploys the new version of your application.

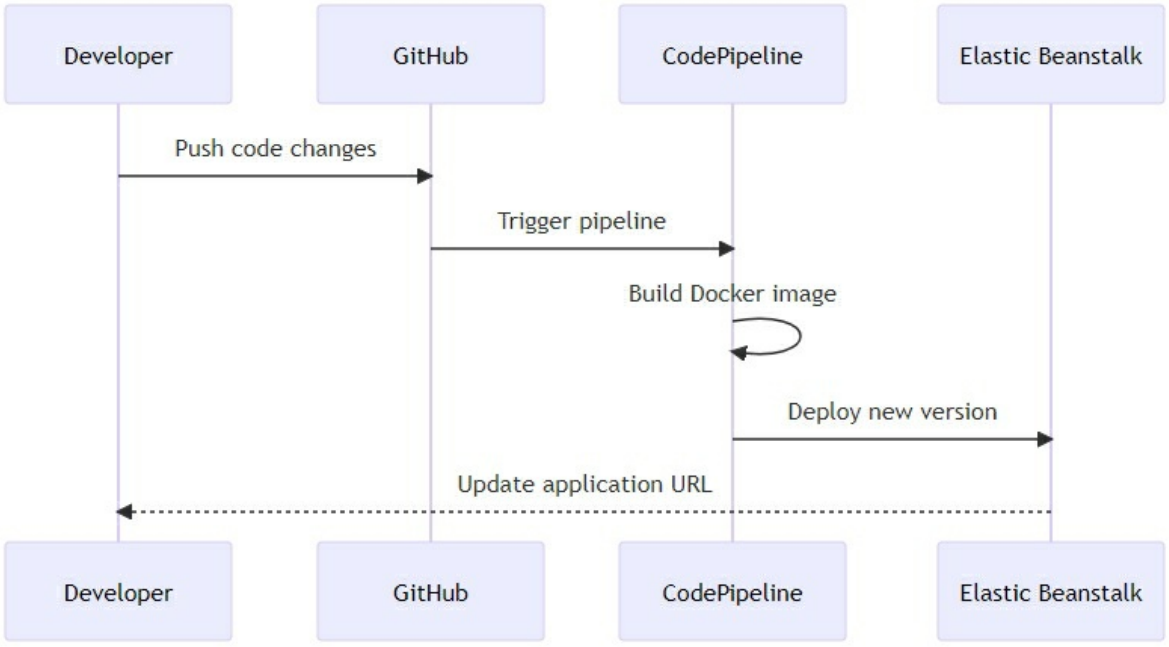


Fig 11.6 Process of Setting Continuous Deployment

To set this up, navigate to the AWS CodePipeline console, and create a new pipeline. Connect it to your code repository, and set the deploy stage to be your Elastic Beanstalk environment. As a next step, every time you push a code change, AWS will automatically build your Docker image, and update your Elastic Beanstalk environment with the new version of your application.

Deployment Verification

After deploying your application, it's crucial to verify that everything is working as expected. The given below walkthrough is how you can confirm the successful deployment of your application:

AWS Elastic Beanstalk Dashboard

The first step is to check the AWS Elastic Beanstalk dashboard. When the deployment process is complete, the status of your environment should change to "Green" with a health status of "Ok". This indicates that AWS successfully deployed your application and can interact with it.

View Application Version

On the same dashboard, you can verify the deployed version of your application under the "Application versions" tab. This tab shows a list of all application versions that have been deployed. You should see the most recent version at the top of the list, tagged as "Deployed".

Accessing the Application

Try accessing your application through the URL provided by Elastic Beanstalk. The URL is listed on the dashboard in the "Environment details" section. If your application loads successfully, then it has been deployed correctly. In our case, for the blog application, you should be able to view the blog posts or even make new posts, update, or delete existing ones.

Checking Logs

If your application isn't loading as expected, AWS provides logs to help diagnose the problem. In the AWS Elastic Beanstalk dashboard, go to the "Logs" link in the sidebar. Request the last 100 lines of logs or a full log dump. Check the logs for any error messages that may indicate what went wrong with the deployment.

Monitoring

AWS also provides extensive monitoring tools. In the AWS Elastic Beanstalk dashboard, go to the "Monitoring" link in the sidebar. This page provides graphs of various metrics like CPU usage, network traffic, and more. If these metrics are within expected ranges, your application is likely running correctly.

Remember that successful deployment does not guarantee that every feature of your application works as expected. Therefore, it's essential to also manually test your application's features or use automated testing tools to ensure everything works correctly after deployment.

Summary

In this chapter, we explored into the essential realm of application deployment. We began by establishing the significance of deployment in backend development. The deployment process serves as the vital bridge connecting the development phase of your application with its delivery to the users. It's not simply about making your application live; rather, it encompasses several considerations such as readiness for scale, potential bugs, and other unanticipated issues that can surface in a real-world scenario.

We then explored the essential preparations for deployment, creating a mental and procedural checklist to ensure the application's readiness. This involved making sure that all features have been thoroughly tested, security measures are in place, the database is secure and appropriately indexed, and the server can handle anticipated traffic loads. The importance of choosing the right deployment platform, like AWS, to match the application’s specific needs was also highlighted.

Lastly, we practically deployed our blog application on AWS, demonstrating the process from configuring AWS Elastic Beanstalk to the point of going live. The process emphasized the importance of monitoring and testing post-deployment, checking logs, validating successful deployment, and confirming the application's functionality. Remember, successful deployment is a milestone in the application development journey, but the real test begins when users start interacting with your application in the wild.

Thank You

Epilogue

As we wrap up this journey of mastering backend development in C++, we reflect upon the vast expanse of knowledge this book has tried to encapsulate. The motive has always been to provide a comprehensive yet practical guide for every individual who wants to understand and implement backend technologies using C++. The book's structure, starting from the basics and gradually traversing to complex concepts, ensures a solid foundation, further enabling the development of robust and secure applications.

We began by delving into the intricacies of the C++ language, exploring its rich features and capabilities. We examined various facets such as concurrency, STL, data structures, and algorithms, thereby creating a strong understanding of the language's inherent power. We acknowledged the critical role of databases in application development, explored MongoDB, and mastered CRUD operations, laying the groundwork for robust backend development.

Our exploration of APIs, particularly REST and gRPC, helped us grasp the essence of communication between different software components. The detailed practical sessions on setting up and implementing these technologies within our ongoing blog project aimed at fostering a hands-on approach. Emphasizing security, the book guided through techniques to safeguard databases, APIs, and web servers, thus ensuring the integrity of our applications.

Testing, a crucial aspect of software development, was another significant focus area. Introduction to Google Test framework and the process of writing and running unit and integration tests provided valuable insights into creating reliable and bug-free code. A complete guide to setting up a continuous testing environment aimed at keeping the code ready for deployment at any moment.

The book culminated with an extensive guide on deploying applications using AWS, one of the most widely used cloud platforms. It emphasized preparing a checklist, selecting appropriate services, and configuring them for continuous deployment. It also imparted ways to verify successful deployment, ensuring that the reader is well-equipped to launch their applications into the real world.

As you navigated through the book, you built a real-world blog application, which served as a practical manifestation of the theoretical knowledge. This hands-on experience was designed to not only solidify the concepts but also help you understand the common challenges and learn how to troubleshoot them effectively.
Although we have covered an extensive array of topics and technologies in this book, it is essential to remember that the world of backend development is vast and continually evolving. This book serves as a launching pad for your journey in backend development with C++. Keep exploring, keep learning, and remember that every challenging project is an opportunity to apply and enhance your knowledge.

Your journey does not end at this point. As you close this book, a world of opportunities opens in front of you. With the strong foundation and practical skills, you've developed, you are now equipped to take on the challenges of real-world projects. The power of C++ and the vast

landscape of backend development now lie at your fingertips. Forge ahead, delve deeper into each topic, explore new technologies, and remember – the learning never stops. Congratulations on your remarkable journey so far, and best wishes for your path ahead in the world of backend development with C++.