M.Sc. (IT)

SEMESTER - II

MICROSERVICE ARCHITECTURE

SUBJECT CODE: PSIT203

Dr. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai

Programme Co-ordinator : **Prof. Mandar Bhanushe**
HOD Science and Technology IDOL,
Mumbai University, Vidyanagari, Santacruz (E).

Course Co-ordinator : **Ms. Preeti Bharanuke**
Assistant Professor (MSc.IT)
IDOL Mumbai University, Vidyanagari, Santacruz (E).

Editor : **Dr. Kamatchi Iyyer**
Director, ISME School of Management & Entrepreneurship.

Course Writers : **Ms. Shraddha Kadam**
Assistant Professor (UDIT)
University of Mumbai

: **Prof. Hiren Dand**
Assistant Professor
HOd, IT Mulund College of Commerce, Mulund (W), Mumbai

: **Miss Raina Baji**
Assistant Professor
SIES College, Sion East.

: **Miss Kirti Bhat**
Assistant Professor
Vidyalankar School of Information Technology
Vidyalankar Marg, Sangam Nagar, Wadala East, Mumbai 400042

: **Mr. Vinayak Pujari**
Assistant Professor
ICS College, Khed, Dist Ratnagiri

June 2021, Print - 1

Published by : **Director,**
Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari,Mumbai - 400 098.

DTP Composed by : **7SKILLS**
Dombivli West, Thane - 421202

Printed by :

CONTENTS

MICROSERVICE ARCHITECTURE

M.SC. (IT) SEMESTER - II

SYLLABUS

Unit	Details	Lectures
I	**Microservices:** Understanding Microservices, Adopting Microservices, The Microservices Way. **Microservices Value Proposition:** Deriving Business Value, defining a Goal-Oriented, Layered Approach, Applying the Goal-Oriented, Layered Approach. **Designing Microservice Systems:** The Systems Approach to Microservices, A Microservices Design Process, **Establishing a Foundation:** Goals and Principles, Platforms, Culture.	12
II	**Service Design:** Microservice Boundaries, API design for Microservices, Data and Microservices, Distributed Transactions and Sagas, Asynchronous Message-Passing and Microservices, dealing with Dependencies, **System Design and Operations:** Independent Deployability, More Servers, Docker and Microservices, Role of Service Discovery, Need for an API Gateway, Monitoring and Alerting. **Adopting Microservices in Practice:** Solution Architecture Guidance, Organizational Guidance, Culture Guidance, Tools and Process Guidance, Services Guidance.	12
III	**Building Microservices with ASP.NET Core:** Introduction, Installing .NET Core, Building a Console App, Building ASP.NET Core App. **Delivering Continuously:** Introduction to Docker, Continuous integration with Wercker, Continuous Integration with Circle CI, Deploying to Dicker Hub. **Building Microservice with ASP.NET Core:** Microservice, Team Service, API First Development, Test First Controller, Creating a CI pipeline, Integration Testing, Running the team service Docker Image. **Backing Services:** Microservices Ecosystems, Building the location Service, Enhancing Team Service.	12
IV	**Creating Data Service:** Choosing a Data Store**,** Building a Postgres Repository, Databases are Backing Services, Integration Testing Real Repositories, Exercise the Data Service.**Event Sourcing and CQRS:** Event Sourcing, CQRS pattern, Event Sourcing and CQRS, Running the samples. **Building an ASP.NET Core Web Application:** ASP.NET Core Basics, Building Cloud-Native Web Applications. **Service Discovery:** Cloud Native Factors, Netflix Eureka, Discovering and Advertising ASP.NET Core Services. DNS and Platform Supported Discovery.	12
V	**Configuring Microservice Ecosystems:** Using Environment Variables with Docker, Using Spring Cloud Config Server, Configuring Microservices with etcd, **Securing Applications and Microservices:** Security in the Cloud, Securing ASP.NET Core Web Apps, Securing ASP.NET Core Microservices. **Building Real-Time Apps and Services:** Real-Time Applications Defined, Websockets in the Cloud, Using a Cloud Messaging Provider, Building the Proximity Monitor. **Putting It All Together:** Identifying and Fixing Anti-Patterns, Continuing the Debate over Composite Microservices, The Future.	12

Unit I

1

MONOLITH TO THE MICROSERVICE WAY

Unit Structure

1.0 Objectives

This chapter gives the brief introduction on the evolution on microservices, concept of microservice with its benefits and how to define a microservice.

1.1 Introduction

The microservices architecture allows to build the software applications with speed and safety. It allows to build the software as a collection of independent autonomous services. These services are loosely coupled. A single software applications is formed by integrating all the services and other systems.

1.2 Monolith Application

Various business requirements are designed in the enterprise software application. A single monolith application has all the business functionalities build as a single unit. In monolith architecture, a monolith application have multiple components. Each of these components provides the business functionality. These components communicate with each other using point to point communication style, to meet the business requirement.

For example, consider the entire retail application build using monolith architecture style i.e. the whole system is build as one single unit. These application has collection of multiple components like order management, payments, product management, etc. To facilitate the business functionality, each of these components communicate to each other.

1.2.1 Disadvantages of Monolith Application

- A monolith application is designed, developed and deployed as a single unit.
- Agile development and delivery methodologies are hard to practice as most of the business capabilities cannot have their own lifecycles.
- Adding and modifying the features in the monolith application is difficult.
- If any part is updated, the entire application needs to be redeployed.
- The monolith application may take longer time to start as the size of the application grows.
- The application has to be scaled as a single unit. Scaling becomes difficult when there are conflict in resource requirements.
- The whole application can be down even if one of the service becomes unstable.
- All the functionalities have to be build on homogeneous technologies/ framework.
- Adopting new technologies or new framework is difficult.

1.3 What are Microservices?

Microservices are small and autonomous services that works together. Principles of the microservices (as James Lewis) are :

- **Microservices are ideal for big systems:**
  - It is designed to solve the problems faced by the big systems.
  - The components of the monolith applications are transformed as independent services.
  - The size of the microservice may vary as per the need of the business functionalities or the problem domain
- **Microservice Architecture is goal-oriented:**
  - Microservice architecture focuses on solving the initial problem faced by the monolith applications.
  - It does not identify specific collection of practices for software development.
- **Microservices are focused on replaceability:**
  - The whole applications requires to be redeployed in the monolith applications, if there are any changes or update in the applications.
  - Microservices overcomes this problem by only replacing (redeploying/ deploying) the services that has been modified or added.

The potential adopters face the following issues while implementing microservice architecture in their organizations:

- The organization has already built the application in microservice architecture. They are unaware that this architecture is called as microservice architecture.
- In microservice application, management, coordination & control of microservices would be difficult.
- The microservice style does not have any unique context, environment and requirements.

Definitions of term “microservice”:

- *Microservices are small, autonomous services that work together.*
  - Sam Newman, Thoughtworks
- *Loosely coupled service-oriented architecture with bounded context.*
  - Adrian Cockcroft, Battery Ventures

- *A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices.*

1.3.1 Characteristics of Microservices applications

Following are characteristics of Microservice applications:

- **Small in size:**
  - Microservices must be small in size. Small team should be able to manage the service.
  - The benefits around interdependence increase as the service becomes small.
- **Message enabled:**
  - Microservice component can communicate with other component through an interface or API.
  - These component interfaces are loosely coupled so that the two microservices can be deployed independently.
- **Bounded by context:**
  - The microservices are built or structured around the business-bounded context.
  - This microservices are more stable than the services built around the technical concept.
  - It is easy to make changes in the microservices as the business processes changes.
  - Domain boundaries are defined using the bounded context.
- **Autonomously developed:**
  - Autonomous service development helps us with Agile and rapid development of business functionalities.
  - All the services in the microservice architecture communicate with each other through the network calls via service interfaces.
  - Thus the development team only needs to focus on developing the interface and functionality of the service.

- **Independently deployable:**
  - Versioned endpoints should coexist, when changes are made in the microservice.
  - It increases the speed of release of new features and the autonomy of the team owning the microservice as they don't have to constantly orchestrate their deployment.
  - One deployed service does not affect the another service with one-service-per-host model.
  - A single microservice can be changed and deployed independently of the rest of the system.
- **Decentralized:**
  - The functionalities are distributed across multiple microservices in the microservice architecture.
  - It is important that each microservice maintain their own database required for the implementing the business functionality offered by it.
  - In this way, the microservices in the architecture can de independent from each other.
  - Because of the decentralized data management, loosely-coupled microservices are offered and different data-management techniques can be used.
- **Built and released with automated processes:**
  - Since microservices has number of moving parts, it is important to built and release the services with the automated tools.
  - Automated testing ensure that the services still work is a more complex process.
  - The deployment of the services in the uniform way everywhere can be helpful. We can also get fast feedback of the production quality of each check-in.
  - The uniform method of deployment can be used in the different environment definition.

1.3.2 Concerning characteristic of Microservice

Below is the guideline about when to use the microservices and when to avoid it:

- Microservice architecture is ideal when:
  - The enterprise architecture requires modularity.
  - The software application has to be deployed in cloud-based container native infrastructure.
- Microservice architecture is not ideal when:
  - The simple software application can solve the business problem.
- It may be difficult to divide the complex systems into microservices. In such cases, introduce the microservices in the areas with minimal impact with the help of use-cases & build the required ecosystem component around that.

1.3.3 The Microservices Way

Speed and safety are the two key aspect of the microservices. These two keys are affected with the every single decision made about the software. The microservices way is about finding the balance between these two key aspect at scale.

Speed and Safety at Scale and in Harmony.

- The Microservices Way

- **The Speed of Change:**
  - The desire of speed is the desire of immediately releasing the changes in the software into the production environment.
- **The Safety of Change:**
  - The risk of damage can be minimize by restricting the rate of change to those releases in the production environments. Thus optimizing the production environments for safety.
- **At Scale:**
  - The software should be build to work when demand grows beyond the initial expectation .This is know as building the system at scale.
  - Systems that work at scale don’t break when under pressure. Built-in mechanisms are incorporated in the systems to increase capacity in a safe way.
- **In Harmony:**
  - Microservice architecture is said to be at harmony when organization succeeds to maintain the system stability while increasing their change velocity.

- In other words, organization created a harmony of speed and safety that works for their own context.

1.4 Deriving Business Value

Successful organizations focuses on increasing the software delivery speed as they are compelled by the speed of their business. The level of safety of the software systems should be tied to specific business objectives. Balance is required between the speed and the safety in the organization.

Every organization will have its own balance. Balance in the organization will be function of its delivery speed, the safety of its systems, and the growth of the organization's functional scope and scale.

1.4.1 Speed-aligned Benefits

Fast software delivery is important for staying ahead of competition and also for achieving sustainable growth in business. Following are the benefits of the delivery speed contributing real business value for each microservice architecture:

- **Agility:** New products, functions and features are delivered more quickly by the organizations and also pivot easily if needed.
- **Composability:** It reduces the development time and also allow the reusability of the component over time.
- **Comprehensibility:** The development planning of the software systems increases accuracy, and allows new resources to come up to speed more quickly.
- **Independent deployability:** New features can be incorporated in the components and quickly released into production environment. Independent deployability also provides more flexible options for piloting and prototyping.
- **Organizational alignment:** Services aligned with the organizational structure helps the team to reduce the ramp-up time. It also encourages the team to build the complex products and features iteratively.
- **Polyglotism:** It permits the use of right tools for the right task. This allows in accelerating technology introduction and increasing solution options.

1.4.2 Safety-aligned Benefits

Loss of information or outages can cause loss in business. Thus, a safe software system is important.

Following are the safety-aligned benefits contributing real business value for each microservice architecture:

- **Greater Efficiency:** The infrastructure cost reduces as the software system's efficiency increases. The risk of capacity-related service outages also reduces because of greater efficiency.
- **Independent manageability:** It contributes to improved efficiency. The need for scheduled downtime is also reduced.
- **Replaceability:** The components can be replaced, hence reducing the technical debt which can lead to aging and unreliable environments
- **Resilience and availability:** A good customer experience is ensured with stronger resilience and higher availability of the services.
- **Runtime Scalability:** The software system is allowed to grow or shrink as per the business requirement.
- **Testability:** Implementation risks in the business can be mitigated with improved testability.

1.5 Defining a Goal-oriented, Layered Approach

Because of limitation of the monolith application, microservice architecture evolved. Despite of this, it is suggested to built the new application as monolith first. The creation and ownership of a monolith can help in identifying the right service boundaries.

The complexity of the software system increases as it scales in the form of functional scope, operational magnitude and change frequency. Once the organization passes certain scale threshold, they can switch from monolith applications to microservice architecture. A set of layered characteristics approach is adopted while adopting the microservice architecture.

1.5.1 Modularized Microservice Architecture

An application or the system is broken into smaller parts in the microservice architecture. There would be some limitations if the software system is modularized arbitrarily. There are also certain benefits of modularization, like:

- Modularization in network accessibility facilitates the automation and also provides a concrete means of abstraction.
- Modularized services can be deployed independently, resulting in the increasing delivery speed.

- Regardless of what the service boundaries are, right tool and platform can be selected for individual services. This is known as polyglot approach.
- The abstracted service interfaces allows more granular testing can be performed.

1.5.2 Cohesive Microservice Architecture

Microservice architecture must already be modularized, in order to achieve cohesion of service. Cohesion of service is achieved by defining the right service boundaries and analyzing the semantics of the system. In this layer, the concept of domain is used.

Benefits of cohesive microservice architecture:

- It enables software speed by aligning the system's services with the supporting organization's structure.
- Composable services are permitted to change at the pace the business dictates.
- System featuring cohesive services can be replaced easily as the dependencies of the system is reduced.
- More efficient system is created by the service cohesion as it lessens the need for highly orchestrated message exchanges between components.
- Service cohesion lessens the need for highly orchestrated message exchanges between component.

Cohesive system can be built with synthesized view of business, technology and organizational considerations.

1.5.3 Systematized Microservice Architecture

System elements are the final and most advanced layer in the microservice architecture. After modularization and service cohesion, interrelationship between the services are examined. The greatest complexity in the system is addressed, in this layer. The biggest and long-lasting benefits are also realized.

Speed of delivery is impacted in two ways in a systematized microservice architecture. The overall software is only comprehensible when the connectivity between services is known. The impacts of changes on the whole systems can be identified and assessed rapidly with agility only. Balance is maintained with speed and safety. The system availability is assured only when the interdependencies of the components are understood. A careful approach based on influence versus control is required while dealing with the complex systems.

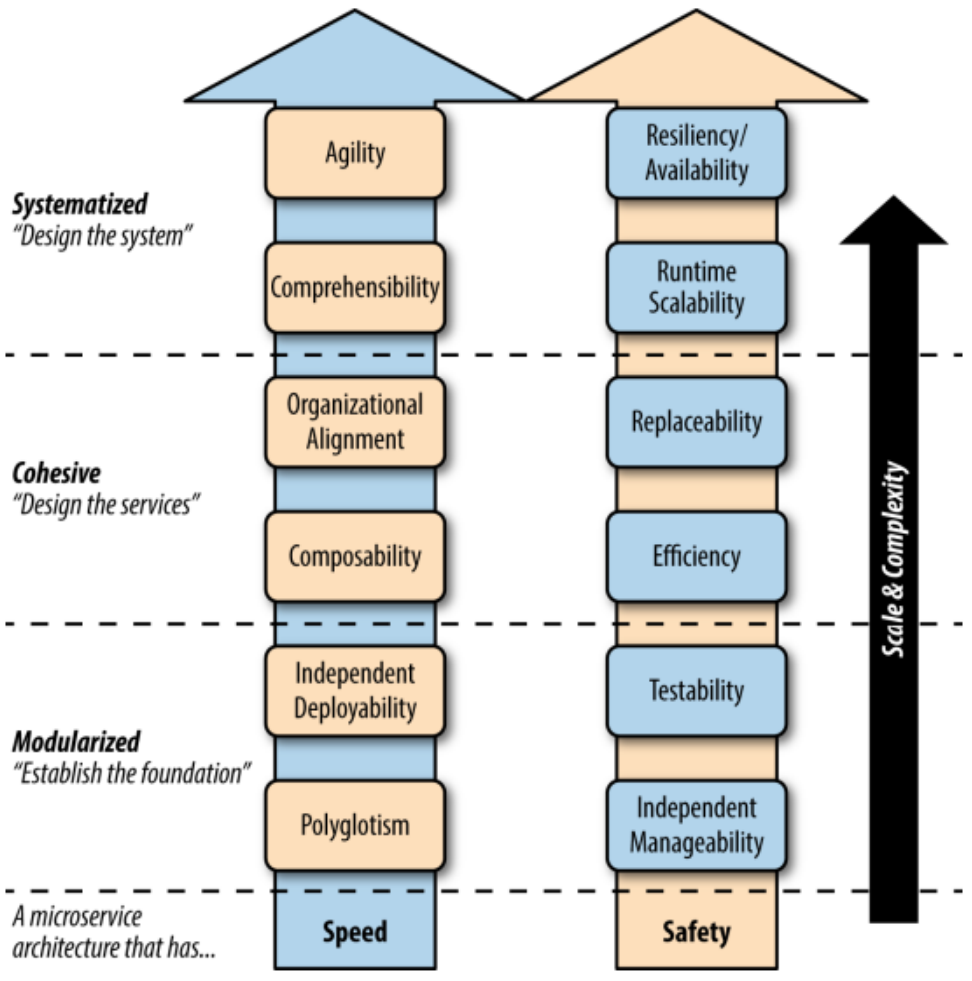
1.5.4 Maturity Model for Microservice Architecture: Goals & Benefits

Modularized, cohesive and systematized layered characteristics helps to define a maturity model that serves the following purpose:

- It classifies the benefits according to phase and goal.
- It illustrates the relative impact and priority of the benefits as scale and complexity increase.
- It also shows the activities needed to address each architectural phase

Microservice architecture can be at different phases for different goals in organization. Maturity model is meant to clarify goals and benefits. It helps the organizations to focus on their microservice strategies and prepare for what could come next.

Figure 1: A maturity model for microservice architecture goals and benefits



1.6 Applying the Goal-oriented, Layered Approach

It is important to understand how microservice architecture can be adopted by the organization to achieve the goals and benefits.

- Start with defining the high-level business objectives.
- These objectives should be weigh against the dual goals of speed and safety.
- Consider the distinct benefits that are targeted within that context.
- The maturity model can be use to determine the complexity of the goal and also to identify the best approach to achieve it.

1.7 Summary

A monolith application is designed, developed and deployed as a single unit. It is difficult to add or modify the features of the monolith applications. In microservice architecture, software systems are made of replaceable components, hence overcoming the disadvantage of the monolith applications.

Microservices way is goal-driven approach to building adaptable, reliable software. To maintain balance between speed and safety at scale is the essence of microservices.

The goals and the benefits of microservices related to speed and safety are introduced. Layered approach - modularized, cohesive & systematized is used to build the microservice architecture. Organizations can use maturity model to target the right goals and benefits for applying a microservice architecture.

1.8 Review Questions

1. What is monolith application? Enlist and explain its disadvantage.
2. Differentiate between monolith application and microservice architecture.
3. What is microservice architecture? Explain the characteristics in detail.
4. “Speed and Safety at Scale and in Harmony. - The Microservices way.” Explain the statement.
5. Enlist and explain the speed-aligned benefits of microservice architecture.
6. Enlist and explain the safety-aligned benefits of microservice architecture.
7. Write a short note on layered approach to build the microservice architecture.
8. Explain in detail the maturity model for building the microservice architecture.

1.9 Bibliography, References and Further Reading

1. Microservice Architecture - Aligning Principles, Practices, and Culture (First Edition) by Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen, Oreilly
2. Building Microservices - Designed Fine-Grained Systems (First Edition) by Sam Newman, Orielly
3. Building Microservices with ASP.NET Core - Develop, Test, and Deploy Cross-Platform services in the cloud by Kevin Hoffman
4. Microservices for the Enterprise - Designing, Developing, and Deploying by Kasun Indrasiri & Prabath Siriwardena, Apress Media

2

DESIGNING MICROSERVICE SYSTEMS & ESTABLISHING FOUNDATIONS

Unit Structure

2.0 Objectives

The objectives of this chapter are to understand the model-driven and design-driven approaches for developing applications of a microservices system. The tools and services required for meeting the objectives of developer and operation teams. In this chapter, our objective is also to discuss about the team size, communication modes and the level of freedom.

2.1 Introduction

Design, complexity, and systems thinking are the domains required for developing applications of a microservices system. Model-driven & design-driven approaches about the application that encapsulates the essential complexity and systems thinking are highlighted in this chapter. We will review the capabilities model for microservices environments. The platforms that represent the tools and services for developer and operation teams to allow them to meet their objectives will be introduced. Company culture including team size can affect the resulting output of your teams will also be reviewed.

2.2 The Systems Approach to Microservices

To develop applications in the microservices way, it is essential to conceptualize the system design as much more than the individual service designs. It does not imply that the design of services can be ignored. It is not enough to just think in the terms of services, but it is necessary to consider how all aspects of the system can work together to form emergent behaviour. Emergent behaviours are greater than the sum of their parts. It includes the runtime behaviour of the microservice applications that emerges when the individual services are connected together and the organization behaviour that gets us there.

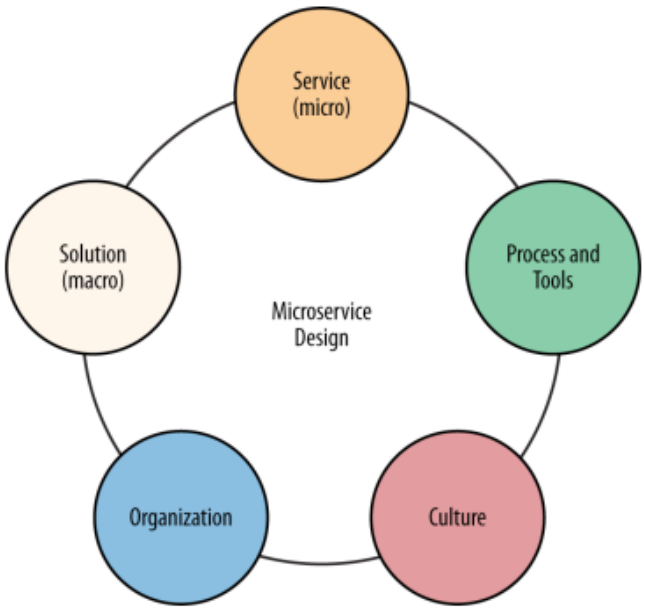
A microservices system encompasses all the important system factors like the structure of the organization, the people who work there, the way they work and the output they produce. In microservice system the runtime architectural elements such as service coordination, operational practices, error handling, interconnection of multiple elements, change in one part of the system and its impact on another part of the system are also important.

Right decisions should be implemented at the right times. These decisions influence the behaviour of the system and also produces the expected behaviour of the system. It is difficult to handle all the elements at the same time. It is challenging

to conceptualize all the moving parts of the microservice system. It is difficult to understand how the interconnected parts work together and the complex emergence that results. It is difficult to predict the results that can arise from change to the system. So, the microservice system designers develop a model. The model-based approach helps to conceptualize the system of study and also easy to understand the parts of the system.

A microservice design model consist of five parts: Service, Solution, Process and Tools, Organization and Culture.

Fig1: The microservice system design model



2.2.1 Service

It is essential to implement well-designed microservices and APIs in microservice system. the services form the atomic building blocks from which the entire microservice system is built. The complex behaviour of the set of the component in microservice system can be understood if one can get the design, scope, and granularity of the service.

2.2.2 Solution

A solution architecture represents a macro view of the solution. It is distinct from the individual service design. In the individual service designing, the decisions are bounded by the need to produce a single output- the service itself. Whereas, in the solution architecture designing, the decisions are bounded by the need to coordinate all the inputs and outputs of multiple services. The designer can induce more desirable system behaviour in the macro-level view of the system.

2.2.3 Process & Tools

The behaviour of the system depends on the process and tools used to build the microservice system. Tools and processes are related to software development, code deployment, maintenance, and product management in the microservice system. It is important to choose the right processes and tools for producing good microservice system behaviour.

2.2.4 Organization

Often how we work is a product of with whom we work and how we communicate. Organizational designs include the structure, direction of authority, granularity and composition of teams from the microservice system perspective.

Organizational design is context sensitive. If you try to model 500+ employee enterprise structure after a 10-person start-up (and vice versa), you may be in terrible situation. It is important for the microservice system designer to understand the implications of changing these organizational properties. Microservice system designers should also know that good organizational designs lead to good service design.

2.2.5 Culture

Culture can be defined as a set of values, beliefs, or ideals that are shared by all of the workers within an organization. Culture of your organization shapes all of the atomic decisions that people within the system will make. It makes it powerful tool in the system design endeavour.

Culture of an organization is context-sensitive feature of your system. It is difficult to measure an organization's culture. Many business and technology leader evaluate the culture of their team in more instinctual way. Organization's culture can be sensed through daily interactions with team members, team products and the customers the organization cater to.

Often culture is an indication of the impact of other parts of your system. Ideals shared on how people do their work and how they work will in turn shape the organizational view. Both are interconnected in nature.

2.3 Embracing Change & Putting It Together

In microservice system, time is an essential element. The decision about the organization, culture, solutions, services, processes, and tools should be rooted in the notion that change is inevitable. The system design cannot be purely deterministic; instead, design adaptability into the system as a feature.

Following are the reasons why the designer should design adaptability into the system:

- It is not possible to determine the end state of the organization and solution design looks like.
- The context in which design decision were made will not remain the same. The good decisions become obsolete very quickly because of changes in requirement, market and technology.

A good microservice designers should not be working to simply produce a solution. They should understand the need for adaptability and endeavours to continually improve the system.

All the design elements together form the microservice system. These elements are interconnected to each other. Change in one element may impact the other element in the system and sometimes unpredictable impact on other elements. The system changes over time and is unpredictable. The behaviour of the system is greater than the behaviour of the individual components. The system adapts to changing contexts, environments, and stimuli.

The microservices system is complex and also produces undesirable behaviours and outcomes from that system is not easy task.

2.4 Standardization & Co-ordination

In organizations, most people work within constraints because of the wrong type of system behaviour or organization failing as a result of particularly bad behaviour. The system designer decides on some behaviour or expectation of the system. This behaviour or expectation may be universally applied to the actors within the system. Some parts of the systems are standardized by introducing policies, governance, and audits for policing the behaviour of the system.

Control of the complex system is illusion. If the actor in the system makes a poor decision, the system behaviour may be unpredictable or may not work as expected, despite of the rules, checks and governance applied.

Control act as system influencers that increases the likelihood of the expected results. The system designers should develop the right standards, apply these standards and measure the results of the changes. This will help in mastering the system design and making the system work as expected. However, control of the system comes at the cost. Standardization and adaptability are enemy of each other. If too many parts of the system are standardized, you risk creating something that is costly and difficult to change.

Standardizing process

The system behaviour can be more predictable by standardizing the way people work and tools they use. For example, the component deployment time may be reduced by standardizing a deployment process. This may help in improving the overall changeability of the system as the cost the new deployment decreases.

Standardizing the way people work in the organization implies that standardization of the work produced, kind of people hired and culture of an organization. The Agile methodology is an example of process standardization. Agile helps to introduce the changes in small increments and allows the organization to handle change easier. The output that is produce begins to change and software releases become smaller and measurability becomes a feature of the product. There are follow-up effects to culture and organizational design. Companies also employ some form of tool standardization. In large organization, they have departments who define the types of tools their workers are allowed to utilize.

Standardizing outputs

Team is defined as group of people who takes a set of inputs and transform them into one or more outputs. The way of setting a universal standard for what output should look like is known as output standardization. When the expected output does not meet the standard output is known as failure.

A team takes a set of requirements and turn those requirements into a microservice. In the microservice system, service is the output. API in the microservice system is the face of that output provides access to the features and data. The consumer has no visibility of the implementation behind API. Output standardization in the microservices context, means developing some standards for the APIs that expose the services. In an effort to improve the usability, changeability and overall experience of using the service, some organizations even standardize how the interface should be designed.

Standardizing people

The types of people work within the organization can also be standardize. For example, the minimum skill requirement can be introduced for the people who want to work in the microservice team. An effective way of introducing more autonomy into the microservices system by standardizing skills or talent. People implementing services should have required skills with the help of which they can make better decisions and create the expected system behaviour.

Organizations should define the minimum level of the skill and experience for their personnel. The organizations that prioritize skill standardization often set very high specialist requirements in order to reap system benefits.

Standardization trade-offs

Standardizing helps exert influence over the system. Organization should not choose just one of the standard to utilize. Introducing different modes of standardization can create unintended consequences in parts of system as they are not mutually exclusive.

For example, the APIs that all microservices expose are standardize as the organization wants to reduce the cost of the connecting elements in the solution architecture. A set of rule for the types of APIs that developers are allowed to develop should be defined. Establish a review process to police this standardization. For example, some organizations standardize a way of documenting the interfaces created by the developers. Swagger is one of the popular example of an interface description language.

The developer can produce limited types of APIs to use by constraining the types of API. It might be the case that the development tool that is used in the organization may not support the interface description language that is been choose. Standardizing may have unintended consequences on the team's work process as standardization attempt to remove uncertainty from the system. This comes at the cost of reducing innovation and changeability.

The number of possible outcomes is reduced because of standardization. The system can be shaped by setting constraints and boundaries for the actions that people within the system can take. These benefits come at a cost. The autonomy of the individual decision-makers is also constrained because of standardization. It is challenge for designer to introduce standardization to achieve the best emergent system outcome as well as employ standards and constraints that complement each other.

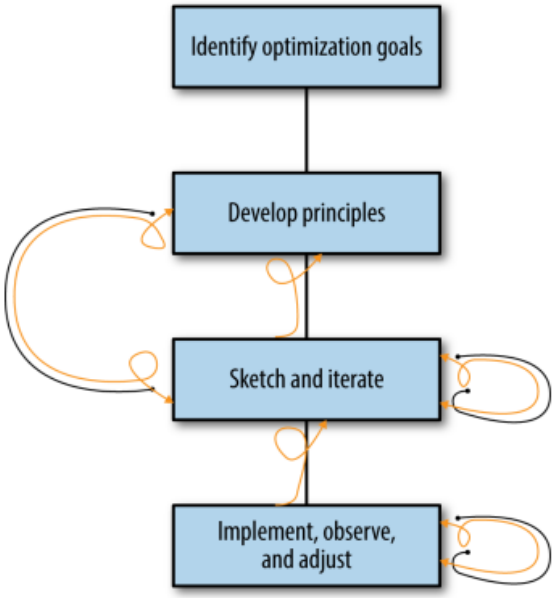
2.5 A Microservices Design Process

The very first step of a service design process is to design the process itself.

—Marc Stickdorn, author of This is Service Design Thinking

Using the right design process is the secret to great design. A good professional designer employs a process that helps them continually get closer to the best product. They don't apply expert advice or make false assumptions about the impact of their design decisions. They work with the process that helps to understand the impact of the assumptions and the applicability of advice as they change the system.

Fig 2: Microservice system design process



A framework for design process that can be use in the microservice system designs is illustrated in fig 2. The process can be customized to fit the unique constraints and context. The design activities may be used in different order or some activities may not be applicable to the goals or some steps may be added in the microservice system design.

2.5.1 Set Optimization Goals

The first task is to identify the goals that make sense in particular situation. This is an important step, as every decision made after this is a trade-off in favour of the optimization goal.

Initially, there can be list of desirable outcomes for the system to be created. As the designing of the system processes, the system designers may find it difficult to pull the system into many directions at the same time. It is easier to design if there is a small set of optimization goals. A single optimization goal provides clarity and also has higher likelihood of succeeding. Optimizing single or some goals doesn't mean that other qualities of the systems are undesirable.

2.5.2 Development Principles

Optimizing system goals helps in forming a set of principles. Principles outline the general policies, constraints and ideals that should applied universally to the actors within the system. This principles guides actor for decision-making and behaviour. Principles should be simply stated and easy to understand. These principles should have profound impact on the system they act upon.

2.5.3 Sketch the System Design

It is important to establish a good starting point for the system design, if there is no existing application or solution architecture in place. It may not be possible to create a perfect system in first or the information regarding the system may not be available or there can be time constraints.

It is good approach to sketch the important parts of the systems. This helps in evaluation and iteration of the system.

Modelling and communication tools can be chosen for conceptualizing the organizational and solution architectures. This step will help process the abstract concept in mind into tangible form that can be evaluated. The purpose of this step is to continually improve the design until comfortable moving forward.

The goal is to sketch the core parts of the system, which includes:

- the organizational structure - team size, direction of authority, details of the teams
- the solution architecture - requirement of infrastructure, and the way in which the services are organized.
- the service design - the expected output and the output details
- the process & tools - the process to deploy service and the tools needed for it

The following evaluation should be done against the goals and principles:

- Will the system foster these goals?
- Do the principles make sense or need to change?
- Does the system design need to change?

When the risk of starting over is small, sketching is powerful. Good sketches are easy to make and easy to destroy. Modelling of the system requires heavy investment of time or efforts. The chances of sketches been thrown out would be less if more efforts are taken to sketch the system. Change would be cheap at this stage of system design. The purpose of the iterative sketching stage is to participate in the process of designing.

2.5.4 Implement, Observe & Adjust

Designers should make small changes, assess the impact of those changes, and continually prod the system behaviour toward a desired outcome. Feedback of the system plays the important role in designing the process. Making changes to one

small part of the system is more difficult as the changes may result in a ripple effect of changes that impact other parts of the system with low visibility.

It is unrealistic that the microservice system provides perfect information about all aspects of the system across all the domains. We can gain essential visibility into system by identifying the few measurements. These measurements should provide the information about system behaviour. This type of metric is known as key performance indicator (KPI) and microservice designers should be able to identify the right ones.

Information gathered about the system by identifying KPIs is useful. Future behaviour of the system should also be predicted with these metrics. The uncertainty about the future is one of the challenges that designers face. We could build boundaries in exactly the right places and make perfect decisions about the size of our services and teams with the perfect information about the how the system might need to change. We make assumptions without perfect information. In the existing application, designers can observe the existing patterns - components that change often, requirements that are always in flux, and services that can expect high usage. But for the new application often have little information to start with. To identify the brittle points of the application, the only way is to ship the product and see what happens.

The risk of making poor decisions increase the chances of the technical debt. Initially, the team creates an application with small set of features with low complexity. The feature set grows over time and the complexity of the deployed application grows. It becomes difficult to make changes in the system. At this point, the team agrees that the application needs to be redesigned and modularized to improve its changeability. The redesign work is continually deferred as the cost of the work is too high and difficult to justify.

It is impractical to overdesign and overengineered the system for future flexibility. An incredibly complex, adaptable system that is built for massive amounts of change that never seems to happen. A good microservice designers should examine the current state and make small, measurable changes to the system, rather than predicting the future.

2.6 The Microservices System Designer

Characteristics/Qualities of the microservice designers:

- Should be able to enact change to a wide array of system concerns.
- Should be able to decide the system boundaries which reflects the boundaries of the company.

- Should focus on a particular team or division within the company and build a system that aligns with the parent company's strategic goals.
- Should be responsible for all the elements of the bounded system.
- Should introduce small changes within the system in order to produce behaviour that will align with the desired goal.

2.7 Goals & Principles

Goals and principles help to design choices and guide the implementation efforts. The companies that provide the higher degree of autonomy to developer teams find this guidance more helpful.

2.7.1 Goals for the Microservices Way

While making decision, it is good idea to have a set of high-level goals. These goals guide us about what to do and how to go about doing it. The ultimate goal in building application in the microservices way: finding the right harmony of speed and safety at scale. This goal allows to build a system that hits the right notes for the organization when given enough time, iterations, and persistence. it might take a very long time for you to find that perfect harmony of speed and safety at scale if you are starting from scratch. We have access to proven methods for boosting both speed and safety and so no need to reinvent established software development practices. Instead, you can experiment with the parameters of those practices.

The following four goals lead to practice that aids both safety and speed of change:

- Reduce Cost: Will this reduce overall cost of designing, implementing, and maintaining IT services?
- Increase Release Speed: Will this increase the speed at which my team can get from idea to deployment of services?
- Improve Resilience: Will this improve the resilience of our service network?
- Enable Visibility: Does this help me better see what is going on in my service network?

Reduce cost

The ability to reduce the cost of designing, implementing, and deploying services allows you more flexibility when deciding whether to create a service at all. Reducing costs can increase your agility because it makes it more likely that you will experiment with new ideas.

Increase release speed

The common goal is increasing the speed of the "from design to deploy" cycle. One can also view this goal as shortening the time between the idea and deployment. The ability to increase speed can also lower the risk for attempting new product ideas or even things as simple as new, more efficient data-handling routines. In deployment process the speed can be increased by automating important elements of the deployment cycle and speeding up the whole process of getting services into production.

Improve resilience

The system should not crash when errors occur. The resilient system can be created when you focus on the overall system and not on single component or solution.

DevOps focuses on improving resilience by automating testing. The tests are constantly run against checked-in code by making the testing part of the build process. This increases the chances of finding errors that could occur at runtime.

Enable visibility

Enabling runtime visibility means to improve the ability of the stakeholders to see and understand what is going in the system. Good set of tools for enabling visibility during the coding process which gives reports on the coding backlog, how many builds were created, the number of bugs in the system versus bug completed, and so on. We also need the visibility into the runtime system like monitoring and logging of the operation-level metrics. Some monitoring tools can act when things go bad.

Trade-offs

Trade-offs also needs to be considered. Reduction in the cost may affect runtime resilience or speeding up the deployment might mean that you to lose track of what services are running in production and reduce the visibility into the larger service network. It is important to balance various goals and find the right mix for the organization.

2.7.2 Operating Principles

It is important to have a set of principles along with a set of goals. Principles offer more concrete guidance on how to act in order to achieve those goals. Principles do not set out required elements but offers examples on how to act in identifiable situations. Principles can be used to inform best practices.

Netflix

Netflix's cloud architecture and operating principles are:

- **Antifragility:** The internal systems is strengthened to withstand the unexpected problems. It promotes this by including "Simian Army" set of tools which "enforce architectural principles, induce various kinds of failures, and test our ability to survive them". Software has bugs, operators make mistakes, and hardware fails. Developers are incentivized to learn to build robust system by creating failures in production under controlled conditions. Error reporting and recovery systems are regularly tested, and real failures are handled with minimal drama and customer impact.
- **Immutability:** The principle of immutability assert autoscaled groups of service instances is stateless and identical. This enables Netflix's system to "scale horizontally". Each released component is immutable. A new version of the service is introduced alongside the old version, on new instances, then traffic is redirected from old to new. The old instances are terminated only after making sure that all is well.
- **Separation of Concerns:** Each team owns a group of services. They own building, operating, and evolving those services, and present a stable agreed interface and service level agreement to the consumers of those services.

Unix

Principles of the UNIX system:

1. Create a program that performs one task well. Build a new program to do the new task. Do not complicate the old program by adding new features.
2. The output of the program should be input of another program. Do not add extra information to the output. Try to avoid columnar or binary input format. Keep less user interaction for the input.
3. Within weeks try to design and build the software/ operating systems. Do not hesitate to throw away the clumsy parts and rebuild them.
4. To lighten a programming task, use tools in preference to unskilled help. Even if you have to detour to build the tools and expect to throw some of them out after you have finished using them.

Suggested principles:

Set of principles guides the software developers and architects in the organization. Every organization will have their own set of principles.

Following are the general principles that can be used as starter material until the organization can set their own principles that fit their work.

- **Do one thing well**: It is difficult to decide what constitute the "one thing" in microservice implementation. The boundaries of the microservices should be decided.
- **Build afresh**: Create a collection of powerful tools that are predictable and consistent over a long period of time. It may be better to build a new microservice component rather than attempt to take existing component already in production and change it to do additional work.
- **Expect output to become input**: The output of one program should be the input of another program.
- **Do not insist on interactive input**: Do not engage human in every step. The script should handle both the input and output on their own. Reducing the need for human interaction increases the likelihood that the component can be used in unexpected ways. At runtime microservice component need not deal with the human interaction. Reducing the dependency on human interaction in the software development process can go a long way toward increasing the speed at which change occurs.
- **Try early**: Microservice components should be "tired early" fits well with the notion of continuous delivery and the desire to have speed as a goal for your implementations. You can learn from your mistakes early and also encourages the team to get in the habit of releasing early and often. Early release helps you to get the feedback and you can improve quicker.
- **Do not hesitate to throw it away**: When you adopt the try early principle, throwing away the early attempt is easier. The component that has been used overtime may not be needed any longer. It also important to throw away the components that have been running in production for a long time.
- **Toolmaking**: The "right tool" should be used for building a solution. One should be able create tools to reach a goal. Tools are a means and not an end.

2.8 Platforms

Good platforms increase the harmonic balance between speed and safety of change at scale from a microservice architecture perspective. Speed and safety are thought

as the opposing properties that requires a trade-off to be made. Right tooling and automation give you an opportunity to cheat the trade-off.

For example, the safety of changes that are made to the system are improved with the principle of immutability. As each deployable unit needs its own associated release mechanisms, infrastructure and management, the release cost for immutability is inherent. The added cost can reduce the speed at which changes can be made. The introduction of containerization tools like Docker makes independent deployability easy and also reduces the associated costs. Both speed and safety of changes are optimized when immutability is combined with containerization.

The conceptual world can be pass to the actual world with the platform. There are many companies establishing and even sharing their microservices platforms. Every company is doing this in their own way which presents some choices to anyone who wants to build their own microservice environment. The platform should be selected with careful considerations like whether to purchase the existing OSS platform or build from scratch, whether the company will provide same types of services as per requirement of the organization, etc.

2.8.1 Shared Platforms

In large enterprises it common to create a shared set of services like hardware, databases, etc for everyone to use. These services are centred around the common infrastructure for the organization. Shared capabilities are platform services that all teams use. These are standardized things like container technology, policy enforcement, service orchestration/interop, and data storage services. It is important in the organization to narrow the choices for these elements in order to limit complexity and gain cost efficiencies.

Centralised shared capabilities that can offer consistent, predictable results are deployed in the organization where they highly value safety of changes. Whereas in some organizations that desire speed at all costs are likely to avoid shared components as much as possible as it has the potential to inhibit the speed at which decentralized change can be introduced. Capability reuse is less important than the speed of delivery in these speed centric companies. You will need to experiment with different forms of shared capabilities to see what works best for the unique context.

Following are the shared services platforms:

- **Hardware Services**: Organizations deal with the work of deploying OS- and protocol-level software infrastructure. There is a team of people in some companies, who are charged with accepting shipments of hardware, populating those machines with a baseline OS and common software for monitoring,

health checks, etc., and then placing that completed unit into a rack in the "server room" ready for use by application teams. With Virtual machines OS can be virtualise. It also automates most of the work of populating a "new machine" and placing it into production. Containers like Dockers can also be used to solve this problem.

- **Code management, testing, and deployment**: The application code can be deployed into the server once ready. That is where code management (e.g., source control and review), testing, and (eventually) deployment come in. Few options like Amazon platform offer all these services and some of them are tied to the developer environment, especially testing.
- **Data stores**: Many storage data storage platforms are available today, from SQL-based system to JSON. It is not possible for the organization to support all the possible storage technologies. Some organizations struggle to provide proper support for many storage implementations that they onsite. Organizations can focus on few selected storage platforms and make those available to all the developer team.
- **Service orchestration**: The technology behind service orchestration or service interoperability is another one that is commonly shared across all teams. There is a wide range of options here.
- **Security and identity**: Another shared service is platform-level security like gateways and proxies. There are number of security products available. Companies like Netflix have developed their own frameworks like Security Monkey. Sometimes shared identity services are external to the company.
- **Architectural policy**: Additional policy services are also shared which are used to enforce company-specific patterns or model often at runtime through a kind of inspection or even invasive testing. Netflix's Simian Army is one of the examples of policy enforcement at runtime. It is a set of services designed to purposely cause problems on the network to test the resiliency of the system. Another kind of policy tooling is one postmortems is another policy tooling that standardize the way outages or other mishaps are handled after the fact.

2.8.2 Local Capabilities

The capabilities that are selected and maintained at the team or group level is called local capabilities. It helps team to become self-sufficient and allows them to work at their own pace. It also reduces the number of blocking factors that a team encounter while they work to accomplish their goals. Team should be allowed to make their

own determination on which developer tools, frameworks, support libraries, config utilities, etc are best for their assigned jobs. Sometimes these tools are created in-house or even they are even created in-house or even open-source, community projects. It is important that the team making the decision is also the one taking responsibility for the results.

Common local capabilities for microservice environments are:

- **General tooling**: A power local capability to automate the process of rolling out, monitoring, and managing VMs and deployment packages. Jenkins is the popular open-source tool. Netflix created Asgard and Animator for this.
- **Runtime configuration**: Many organizations who are using microservices have found pattern in rolling out new features in a series of controlled stages. It allows team to assess a new release's impact on the rest of the system. Twitter's Decider configuration tool is an example of this capability.
- **Service discovery**: Service discovery tools make it possible to build and release services that, upon install, register themselves with a central source. It then allows other service to discover the exact address/location of each other during runtime because of which various team can make changes to the location of their own service deployments without fear of breaking some other team's existing running code. Apache Zookeeper, CoreOS' etcd, HashiCorp's Consul are some popular service discovery tools.
- **Request routing**: The actual process of handling request begins once you have machines and deployments up and running and discovering services. Request-routing technology is used by systems for converting external calls into internal code execution. Netty and Twitter's Finagle are popular open-source services.
- **System observability**: Distributed environments is getting a view of the running instances-seeing their failure/success rates, spotting bottlenecks in the system, etc. Twitter's Zipkin, Netflix's Hystrix are examples of tools for this task.

2.9 Culture

Company's culture is one of the critical foundation elements as it does not only set the tone for the way people behave inside an organization, but also affects the output of the group. The code that is produced by the team is the culture. You should consider the following three aspect of culture for microservice efforts:

- **Communication**: The way teams communicate has a direct measurable effect on the quality of the software
- **Team alignment**: The size of your teams also has an effect on output. More people on the team means essentially more overhead.
- **Fostering innovation**: Innovation is essential for the growth and long-term success.

2.9.1 Focus on Communication

Communication dictates output. It leverages understanding to improve the group's output. Organizational metrics are significantly better predictors of error-proneness in code. Code complexity and dependencies are also the measure. The process of deciding things like the size, membership, even the physical location of teams is going to affect the team choices and, ultimately, the team output. You can set up your teams to make things easier, faster and to improve overall communication by considering communication needs and coordination requirements.

2.9.2 Aligning your Team

Team alignment affects the quality of code. The alignment of the team structures can be used to improve the speed, resilience, and visibility for the microservice efforts. The size of the team plays the important role. As the size of the group grows, the number of unique communication channels grows in a nonlinear way. It is the common problems and needs to be kept in mind as you design your teams. There are a number of other factors in establishing your teams including responsibilities, deliverables, and skillsets that need to be present within a team.

2.9.3 Fostering Innovation

Fostering innovation within the organization is also the important element in the organization. Companies want to make innovative thinking common within the organization. Creative and innovative ideas are needed to adopt a microservice approach for developing software. It is important to spend some time in exploring innovation and also to understand how it can affect the organization. Innovation is usually thought of as an opportunity to improve what a team or company already has or is currently doing.

Innovation process can be very disruptive to an organization. Changing the way, we do things sometime can be needless and even threatening exercise. Therefore,

the act of innovation can be difficult. Sometime innovation process may also look chaotic from outside. Innovating can mean coming up with ideas that might not work, that take time to get operating properly, or even start out as more costly and time consuming than the current practice.

Few key principles adopted by companies to enable innovations are:

- **Provide a level of autonomy to the team**: Allow teams to determine the best way to handle details within the team. Team leaders should be taught to provide context for the teamwork and guidance on meeting goals, but not to control what the team does.
- **Build in a tolerance for some level of chaos**: There should be some understanding developed that it is OK if some things look a bit disorganized or messy. Fostering innovation is setting the boundaries that allows team to act on their own within these safe boundaries. It prevents teams from taking actions that threaten the health and welfare of the organization.

2.10 Summary

In this chapter, the microservices system model and a generic design process for influencing the system is introduced. Decisions made about the organizational design, culture, solution architecture, process, and automation can result in unintended consequence to the system as a whole. It is important to maintain the holistic perspective and to continue observing and adjusting as required.

We have also studied about the common platform capabilities and culture of the organization and team. The quality of the output depends upon the communication among the member, team size and the level of innovation supported in the team.

2.11 Review Questions

1. With the suitable diagram, explain the microservice design model.
2. Write a short note on standardization and coordination.
3. Explain microservice system design process.
4. Enlist the qualities/ characteristics of microservices system designer.
5. Write a short note on “Goals for the microservices way”.
6. Enlist and explain the operating principles of Netflix.
7. Enlist and explain the operating principles of UNIX.
8. Enlist and explain the operating principles that can act as starter material for the company.
9. What are shared and local capabilities of the platform?
10. Write a short on the culture of the organization as the foundation element for the microservice way.

2.12 Bibliography, References and Further Reading

1. Microservice Architecture - Aligning Principles, Practices, and Culture (First Edition) by Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen, Oreilly
2. Building Microservices - Designed Fine-Grained Systems (First Edition) by Sam Newman, Orielly
3. Building Microservices with ASP.NET Core - Develop, Test, and Deploy Cross-Platform services in the cloud by Kevin Hoffman
4. Microservices for the Enterprise - Designing, Developing, and Deploying by Kasun Indrasiri & Prabath Siriwardena, Apress Media

Unit 2:

3

SERVICE DESIGN

Microservices Boundaries:

- Microservices is an architectural style that structures an application as a collection of loosely coupled services.
- It is a distinctive method of developing a software system and it always tries to focus on reducing the size of the code so that it could be understood.
- As we know that before developing the microservices application, the first idea that comes in our mind is that it should be small and the team who is going to work on this project should also be a small team.
- Team should focus on the quality of the product value and not on the quantity of the product value, if we focus on quality of the product then, our application would be reliable, more productive and produce more efficiency.
- It is more important to focus on how the services are being used within our system.
- Most company focus on quality of the product, not the quantity of the product. In the business context Eric Evans said that to achieve this, there is concept of domain driven design that focuses on the quality of the product.

Domain Driven Design:

- A Domain Driven Design is a way of looking at a software from a top down approach. When we are developing a software our focus shouldn't be primarily on the technology, but should be primarily on the business or whatever activity we are trying to assist with the software.
- The domain can be any business domain like railways, aviation, banking, insurance.
- Break your existing system into small portions to achieve the goal and increase the productivity as well as increase the services.
- There are many ways to break your large system into small systems. All the Heavy and Large system are built by using C, C++, R Language.

- Domain driven Design defines the boundaries of a subsystem in the context of a larger system. In order to achieve the offers of model centric, they rely on a view for Software system design.
- These qualities make it better and reliable in a business context.

Bounded Context:

We should be very careful while implementing and combining a small system into a larger system in the business context.

- Business Context promotes the Object-model first approach for the services, and follows the concept the domain driven design.
- Bounded Context combines the details of a single domain such as a domain model, data model, application services etc. and defines the integration points with other bounded contexts or domains.
- When the code is combined, Application becomes buggy, unreliable and difficult to find where the Bug is, and most of the time, the team members are also confused because the code is too large.
- While developing the microservices, the concept should be clear and specific in our mind that we should follow the best approach to complete the application and always try to make a code in short so that every team member can understand the code.
- Each component in the system follows its own bounded context which means the model for each component lives within their bounded scope and are not shared across the bounded contexts.
- Domain Driven Design is used in the system to find the properly identified bounded context within the system. These are the more effective ways to design the microservices boundaries.

API Design for Microservices:

- Microservices become more effective and more valuable when their code and their components are able to communicate with each other.
- A good API Design is important in a microservices architecture and because of that, all data exchange between services happens, either through message passing method or API Calls.
- There are two ways to create an API for microservices, they are as follows.

Message-Oriented:

- In a microservices architecture, it is important that services should be updated on time in a way that it doesn't affect its code's ability to work with their component of the system.
- In the same ways, it is vital and important for the API to connect these components to be designed with safety in mind.
    - Most of the time it is needed to address this necessary thing by taking a message-oriented approach.
    - Example: Netflix follows message formats to communicate internally via TCP/IP Protocol and for external customers it uses the JSON over the HTTP by using mobile phones, browsers.
    - TCP/IP protocol is a connection-oriented protocol and it communicates with each other by using the IP Address and port number. Message passing can happen at both the sides at the same time.

Hypermedia-Driven:

- Hypermedia API design is based on the way as that HTML works in a web browser and HTTP message are transmitted over the internet, it contents data and actions encoded in HTML format.
- Hypermedia provides links to navigate a workflows and template input to request information. In the same ways as we use the links to navigate the web and forms to provide the input.
- Hypermedia API message contains data and actions which provides necessary elements for it to work dynamically with client services and application.
- Example: When we use google chrome or other application on a laptop, our system processor makes a queue and it's works on the fundamentals of FIFO(First in first out). To run the application on the system, CPU allocate its process id and port number.
- HTTP message are sent to an Ip Address and a port number i.e. 80/ 443/88 and message contains the data and action which includes data in a HTML format.

Data and Microservices:

- Data is a general concept that refers to some existing information or knowledge.
- From a System Engineer point of view, we need to identify the data as well as process the data into the database, and also retrieve and manipulate the data from the database.
- Data are stored in the centralized database server so that it would be easy to manage and monitor the data.
- The biggest reason for this is the absence of strong, centralized and uniform control of the entire system of distributed systems, which makes a formerly efficient processed data.

Shipping, Inc:

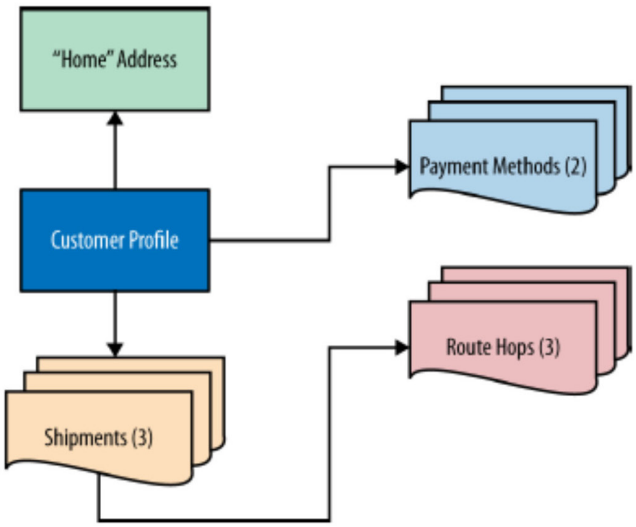
- Suppose we have to create and implement the application for a shipping company and the shipping company has told to us create one application in such a way that it follows the microservices nature and for the delivery parcel company product should be tracked from one place to another place and it would go on one by one .
    - It must follow the process so that the data could be checked on priority and it goes from various data warehouses and is delivered to the destination.
    - We are building a native application of whatever the product is purchased by the customer so that they can trace the product.
    - Two microservices depend on the system design of the shared tables and data in it. There are two techniques to use to avoid data-sharing in complex use cases.

Event Sourcing:

- Event Sourcing deals with an event store of immutable logs of events, in which each logs i.e. a state change made to an object represents an application state.
- An event store is like a version control system. In microservices architecture, we can persist using aggregates as sequence of event.

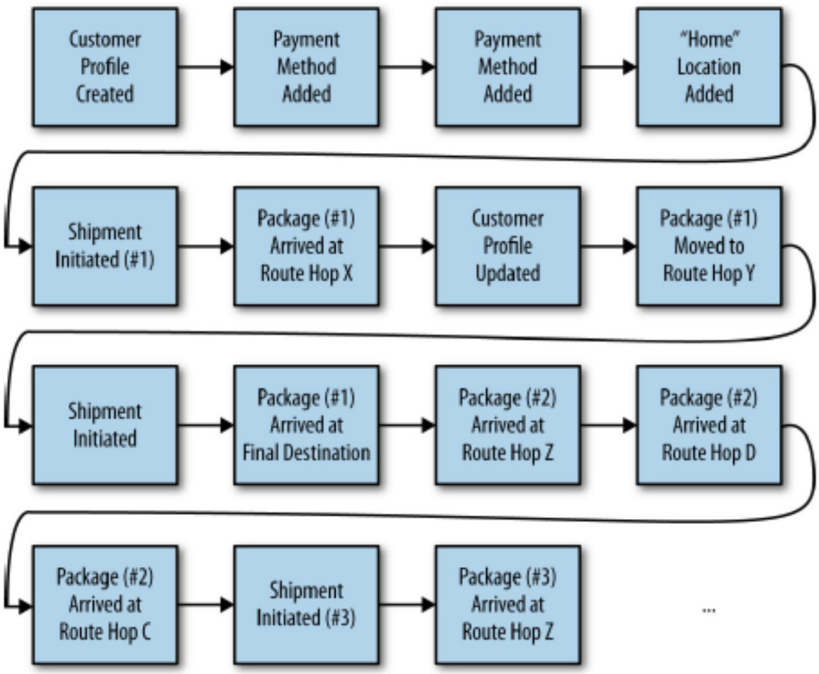
- Events are facts which represent some actions happened in the system and these are immutable i.e. which cannot be changed or be retrieved.
- Example: Events of the e-commerce system are ordercreated, paymentdebited, orderapproved, orderrejected, ordershipped, orderdelivered.

Figure 3.1: Data model for Shipping Inc. using "current state' approach



The corresponding events-based model is shown in Figure 3.2

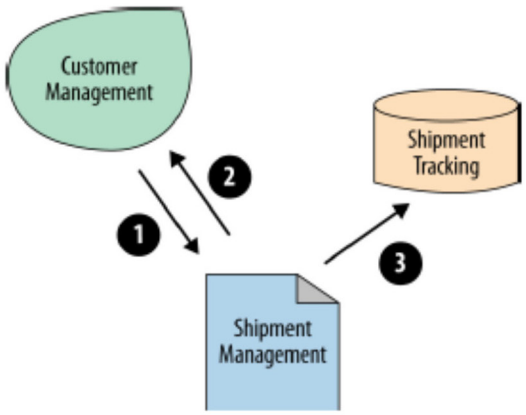
Figure 3.2: Data model for Shipping Inc. using event sourcing



System Model for Shipping, Inc:

- A good start for a microservice system is to identify the bounded context in the system.
    - Below figure shows that bounded context of system design.

Figure 3.3: High-level context map for Shipping Inc's microservice architechure



- Customer Management is used to create, edit, enable, and disable customer's details and provides a view of representation of a customer.
- Shipment Management is responsible for creating an entire lifecycle of a package from pick-up and drop facility from origin to destination.
- Shipment Tracking is used for providing the user interface for the user to track his delivery from a mobile or web browser.

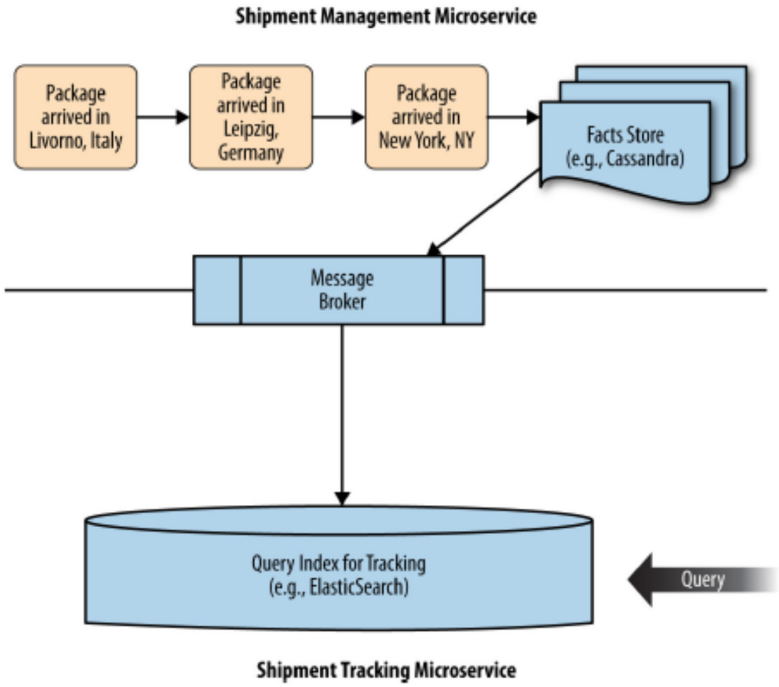
CQRS:

- CQRS stands for Command Query Responsibility Segregation and its name suggests that it segregates or separates the application into two parts.
- Commands are used to perform an action to change the status of segregation or separation and query are used to provide the query model for the segregation.
- Command query responsibility segregation is used for an independent architectural style and mostly used with the event sourcing for developing a model.
- Command query responsibility segregation is a design pattern for data model and data query. Instead of updating, editing, inserting, and deleting the data

from one database or a data model, we can use separate database or data model for maintaining and executing these tasks.

- We can use different databases for writing operation and querying operation.
- Below is the Shipment Management Microservices architecture diagram.

Figure 3.4: Data flow in command-query responsibility segregation (CQRS)-based model for Shipping Inc.



Distributed Transaction and Sagas:

- If we represent a data and a database in such a way that it follows the data coupling model between microservices, it will become difficult to execute the services very smoothly.
- While in the same way, by following the data workflow to represent these models, it is also a problem to solve these queries.
- Most of the models and delivery services are used and follow the old model where each step go through one by one and if any of the step fails, all the model and services fail, due to some mistake like RDMS (Relational Database Management Model) called transaction.

- To avoid these faulty, Sagas came in the market and are used for long lived and distributed transaction.
- Sagas are very powerful because they allow running transaction-like, reversible workflow in a distributed, loosely coupled environments.
- In Sagas, it follows each and every step for developing small and big microservices , during this, if any one of the step fails then it looks back to the previous step and it detects what was the error and tries to solve that particular error which was initiated.
    - A saga has its own sequence local transaction and each service in a saga performs its own transaction and publishes an event.
    - Other services listen and perform other transactions as precisely as the previous services are done. If any one of the services and transaction are fails due to some reason and before moving to the next phase it tries to reduce the error and fix it as soon as possible.
    - Examples: First of the in the order services, first it triggers the order services tab and user selects different dishes from particular restaurant and he is going to place this particular order for the restaurant. Before getting the order placed on the restaurant, he needs to pay for that particular order and once the payment services finish the transaction it notifies the order servicing payment is received and after finishing the payment, the order is being prepared. The delivery executive goes to the restaurant and picks the order so that he will deliver product on time to user.

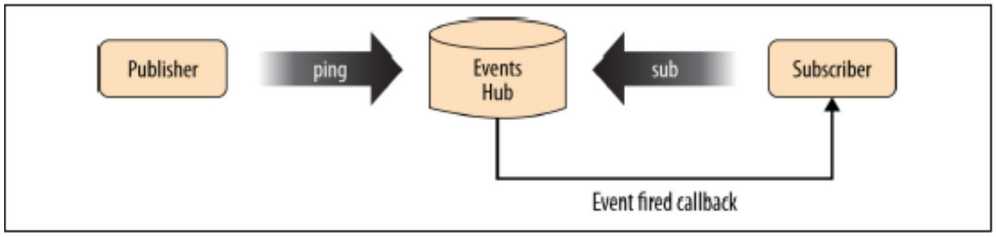
Asynchronous Message-Passing and Microservices:

- Asynchronous message passing play an important role in to keep the things loosely coupled in a microservices architecture.
- Loosely coupling means reducing the dependencies of a class that uses different classes directly.
- Tightly coupling means that classes and objects are dependent on one another.
- Asynchronous Message passing uses loosely coupled message passing to communicate with each other.
- The Messaging Passing workflow is a just simple publish or subscribe workflow as we have seen and most of the time we work like a Https API

works, where there is a client and server and communication between them happens only by using the message passing .

- First, the client sends the message to server for the services like file services, image servicing, text and video services and if the these types of services are available at server side then the server will search in its database and according to that search, will return the data to the intended client by connecting the port number and protocol.

Figure 3.5: Asynchronous message-passing implemented with a PubSubHubbub-inspired flow



3

SYSTEM DESIGN AND OPERATIONS

Independent Deployability :

- Independent deployabilty is one of the primary principles of the microservices architectural style.
- Independent deployability means each microservices should be deployable completely different to each other.
- It is the idea that we can make a change to a microservices and deploy it into the production stage without having to use of any other services.
- Scaling hardware resources on the premises would be very costly to manage and monitor. To reduce the cost of the resources, we need to buy the resources so that we could monitor, maintain and configure the resources on time, there is no need to pay money for the monitoring, maintaining and scaling the resources.
- The shipping Inc. is confident that their security team will easily allow deployment of safe microservices to a private and public cloud.
- There is another benefit of using the independent deployment in the microservices as an operational cost and flexibility.
- For Customer Management and Shipping Management, there would be two different teams for deployment of separate microservices.
- Customer Management is used to create, edits, enables, and disables customers details and provide a view of a representation of a customer.
- Shipment Management is used to responsible for creating an entire lifecycle of a package from pick-up and drop facility from origin to destination.

Docker and Microservices:

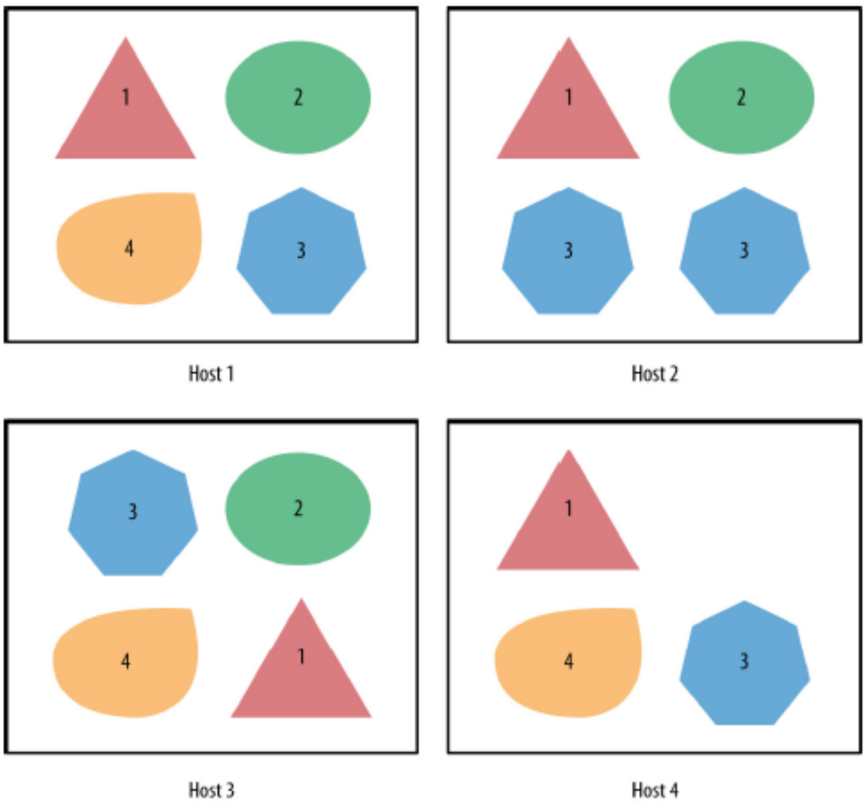
- Docker is a tool designed to make it easier to create, deploy and run applications by using containers.

- Containers allow a developer to package up an application with all parts of it and its needs. The docker is a container and most widely used for deployment of microservices.
- By doing so, you can deploy of your application at any Operating System like windows, Linux and IOS Devices.
- Docker is an open source tool that means any one can download it from docker website, you need not pay any money for downloading it on your machine and expand their meet according to your wish.
- Docker allows applications to use the same kernel as a Linux kernel as the system that they are running on and allows only those applications to be shipped with things no already running on the host laptop/ computer.
- Linux containers use a layered filesystem architecture known as union mounting and allows us a great feasibility to use these services which are not found in the conventional VM architecture.
- Run only one service at a time so that the processor can run smoothly, “Run only one process for one container”. If there are a lot of processes running on the same time, it will run the first process and it will go on and on, following the same procedure to execute.
- Docker containers and microservices architecture are two ends of the road that lead to the same and achieve the goal of continues delivery and operational efficiency.

The Role of Service Discovery:

- If you are using the docker containers for your package and deployment of your microservices then you can simply use simple configuration for multiple microservices.
- This will enable and help us to discover and communicate with each other and which is used in local development and fast prototyping.
- You can deployment at least three or more than three docker host or machine a number of container on each of them.

Figure 3.1 Microservice deployment topology with nonuniform service distribution



- Each number of instances of docker containers contains different number of shapes, size and color as described as the above picture.
- There might be a possibility that many of the services are hosted on the same machine so that we can identify the host by just the IP Address.
- If we allocate the IP as per the microservices then after assigning the IP address to the host, assigning that will become very difficult to us as well as to understand.

The Need for an API Gateway:

- API Gateway includes security, transformation, orchestration and routing to secure the service and product.
- **Security:**
    - We develop the microservices architecture in a way that it has high degree of freedom. There are lots of other services we develop for more moving path in single application.

- Mature microservices include more complex community and enterprise application and more of the microservices are deployed, more dangerous security is found in the application.
- To secure our microservices application, API endpoints provided by various microservices are secured using a capable API gateway.
- API gateway is capable for providing a unique entry point for external customers, independent of the number so that external customers will be able to interact with out microservices application.
- API Gateways are required to separate the external public API and Internal API and allows and defines the boundaries to be added.
- API which are called by microservices can be frontend and backend. Frontend can contain and it can be called directly by API Client such as mobile application, Web Application, Web Services Etc.
- So, it is always recommended to secure our microservices application from the external threat and vulnerability, better always use the API Gateways.

- **Transformation and Orchestration:**
  - As we know that microservices has a single capability and in microservices we develop the application in such a way that it should be small, reliable, portable and secure.
  - The UNIX has some statement about the features and algorithms that do one thing and do it well. It means that Unix follows and uses the single capability approach to work smoothly and facilitate the orchestration by piping the input and output.
  - To make microservices useful and needful, always use the orchestration framework like Unix and Linux piping, like the web API.
  - For example, Framework is something like if we consider an example in which we have a photo and we want to setup the photo in such a way that it should look good. To keep and to make the photo look good, we need a photo frame of wood. So, here the rectangle frame of wood defines the framework for the photo that we have.
  - Microservices due to their nature, has narrow specialization, has a small size and are very useful deployment unit for the team production.
  - Webservices are distributed in nature, this is quality of webservice that makes them very powerful interface where the user interacts with the product.

Routing

- Discovery system is used to find the system services to find out the system microservices architecture.
- Routing is the process of forwarding the packets from one network to another network based on the network architecture.
- Consul and etcd skyDNS provide a DNS based interface to discovery. DNS is required to convert IP to name and vice versa. It can be used for debugging and to find out the bugs.
- DNS queries only look for local based zone domain and IP mapping and microservices domain mapping with an IP and Port combination.
- Gateway is required for converting the one technology to another technology and API Gateway can interface with directly http/https or DNS interfaces for discovery system.
- Load Balance is used to achieve the same goal and API Gateways are used to secure routes for a packet from one network to another network to the microservices

4

ADOPTING MICROSERVICES IN PRACTICE

Solution Architecture Guidance:

- Solution architecture is totally different from single / individual service design element which make a micro sight of solution.
- There are some levels that we need to identify the solution. For that, following as below.

How many bug fixes/features should be included in a single release?

- When the microservices are ready for production, but before the release of microservices on platform, first of all we need to identify the bug, need to release the alpha and beta code for the user and programmer to check whether there are any bugs in that or not.
- Alpha release is provided for the user level interface, so that every user can interact and see and image and find out the bug in the microservices.
- Beta version are provided for the programmers to identify the bug and solve these bugs as soon as possible before the production stage.
- The big reason to choose limiting the number of changes in the microservices is to reduce uncertainty and to find out the bug in the code.
- If we release the system component for the production release and if there are multiple bugs caught and uncertainty, then there is no use of that and it will increase number of interactions between those things.
- if you release a component that contains 5 changes and it causes problems in production, you know that there are 10 possible ways in which these 5 changes could interact to cause a problem.

How do I know our microservice transformation is done?

- It is not possible to maintain and create system microservices for the entire life and vital information system is never done.
- This is true , some experience on which the architects and developers spent a lot of time and trying to complete the system solution but they are also unable to change architecture and system design for entire lifecycle, because each and every time we can see there are lots of changes coming nowadays so we cannot fix it for the entire lifecycle.
- one of the advantages of microservices is that change over time is not as costly or dangerous as it might be in tightly coupled large-scope release models.

Organizational Guidance:

- A microservices system design includes structure, direction of authority, granularity, and composition of terms.
- A system designer knows the implications of changing the small modules and its properties that will be good for the organization design.
- As a system designer of the organization, you can access structure and associated culture of the microservices and most of the organization makes the software for better understanding and addressing the software architecture.
- A microservices architecture is made of small-2 services, business-aligned and which expose the wisdom of the services.
- Most of the organization are not following the protocol of the microservices architecture as same path and in that case, it will become extremely difficult for the organization.
    - o How are responsibilities divided between the teams?
- Are they aligned to business domains and meet the technology?
- How big are the teams?
- What skills do they have?
- How to power distributed between the teams?
- The above are some microservices related questions which help us to understand the organizational factors and adoption efforts.

- Small teams are made so that they could make small, reliable, portable services for specific business domains and these small teams are responsible for changing the role for the delivery of microservices.

Culture Guidance:

- Culture is a word for the way of life of groups of people and which integrated pattern of human knowledge, belief and behavior.
- Culture is very important for the organization which gives the shape and design for the organization in which atomic decision are made.

1. How do I introduce change?

- When we develop and design the architecture of microservices and movement when all the module is done before the transition and in the production phase, we need to keep in our mind, whatever we are making for microservices that will follow the greenfield environment.
- While developing and designing the microservices architecture, keep in mind that no mistakes are made otherwise the movement will come when we realize that we have made some mistake and we think to come back on previous phase then it will become very difficult to rectify those error which are occurred during this phase.
- To apply a refactoring strategy to the organizational design, we need to follow some practices:

1. Devise a way to test changes.
2. Identify problem areas in your organizational design.
3. Identify safe transformation.

- When we come back to previous phase to rectify the issue, we can measure and observe the performance of the application and you can verify the code and change the code according to requirement and combine them so that requirement can meet for the organizational behavior.

2. Can I do microservices in a project-centric culture?

- Team who implements the features, application, or services for the microservices architecture continue to support our microservices to improve the work of the code its lifetime.

- Teams are hired to support and to improve the microservices and if there is any fault or a problem is coming to create a new feature or module to the existing system design and add feature. So that any of the team's member can update the data in existing system code and design.

Can I do microservices with outsourced workers?

- You can build or extend your in-house team by hiring software and developer teams and be in control of the whole app development process but it might not be cost nor time effective and you can gain much more with IT Outsourcing.
- Outsource structure of microservices system can be lent to be developed, to an external organization.
- Outsource workers or teams are hired for building for microservice system design, architecture and to meet the capability requirement of the owning organization.
- Outsource workers or teams should be right in size and build the microservices in such a way that it should meet the requirement of the system design and must be capable to take good decision.
- Outsource Teams cannot adopt the simplicity and regulation, protocol and organizational culture which means that ir is important elements in decision making skills.

Services Guidance:

- Make sure that microservice design ensures the agile or independent development and deployment of the services.
- Always focus on the scope of the microservices but understand that its not about making the services smaller.
- In a microservice system, the services form the atomic building blocks from which the entire organism is built.
- Below are some question and issues while implementing well-designed microservices and API.

Should all microservices be coded in the same programming language?

- Microservices should not be designed and implemented in the same language. It should be coded and developed in such a way that it must be portable and reliable for the all the platforms and environments.
- Every time new-new languages are being developed and the markets are changing so that it meets the specific requirement of the microservices so that it must be portable and reliable for the all the platform and environment.

Unit 3

5

INTRODUCTION TO ASP.NET CORE AND DOCKER

Chapter Structure:

5.0 Objectives

This chapter will make you understand the basic concepts of ASP.NET Core, developing ASP.NET applications using the C# language, understanding Docker and its basic commands.

5.1 Introduction To .Net Core

.NET Core is an open-source and general-purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform in nature supporting Windows, MacOS, and Linux and can be used to build device, cloud,

and Internet of Things applications. .NET Core applications can execute on both .NET Core and traditional .NET Framework (.NET framework 4.x). It was initially launched as .NET 5 but later on it was completely rewritten from scratch and renamed to .NET Core 1.0. It allows the developers to build all kinds of software including Web, Desktop, Mobile, Cloud, Gaming, Internet of Things etc.

Components of .NET Core are:-

1) CoreFX – CoreFX is similar to base class library (BCL) in the traditional .NET framework (i.e.) it is the sum total of all .NET libraries that comprise the framework. It is a set of modular assemblies which is available as NuGet packages and completely open source which is available on GitHub.

2) CoreCLR – CoreCLR is a lightweight, cross-platform runtime of .NET Core. It includes the garbage collector, JIT compiler, primitive data types and low-level classes for handling exceptions and performing other tasks. Garbage collection is the clean-up of unused object references. JIT compilation is compiling the Intermediate Language (IL) code in the .NET assemblies into native code. Exception handling includes try/catch, throw statements and it is a part of the runtime and not the base class library.

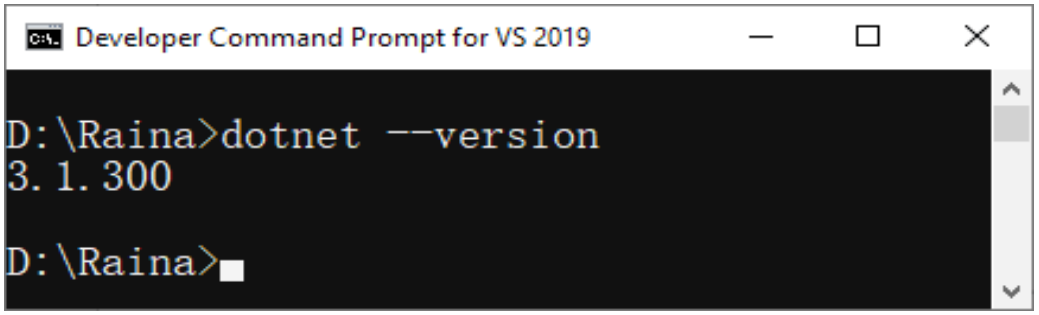
5.2 Introduction To Asp.net Core

ASP.NET Core is the open-source version of ASP.NET that runs on MacOS, Linux, Windows and Docker. ASP.NET Core was first released in the year 2016 and it is a re-design of the old versions of ASP.NET. It is a collection of small, modular components that can be plugged into the application to build web applications and Microservices. Within ASP.NET there are APIs for routing, JSON serialization and MVC controllers. ASP.NET Core is available on GitHub. The cross-platform web server used in ASP.NET Core is Kestrel. It is included by default in ASP.NET Core project templates.

5.3 Installing .Net Core

.NET Core is made up of two components - runtime and SDK. The runtime is used to run a .NET Core apps and the SDK is used to create .NET Core apps and libraries. The .NET Core runtime is always installed along with the SDK. The latest version of .NET Core is 3.1. To install .NET Core, follow the instructions mentioned at the website https://dotnet.microsoft.com/download. After the installation process is over execute the command as shown in Fig. 5.1 in Windows to check the version of the core installed.

Fig 5.1: Illustration of dotnet –version command



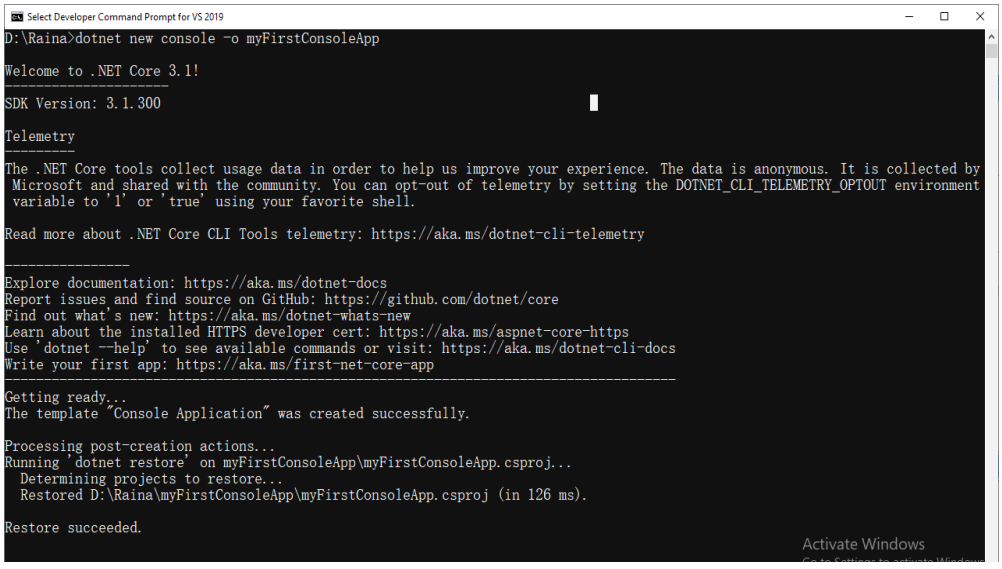
If the above command works, then the basic requirements for .NET Core are installed on the workstation. In a Windows machine, you should be able to find the .NET Core installed runtimes in the following directory: "C:\Program Files (x86)\dotnet\shared\Microsoft.NETCore.App\3.1.4".

Apart from this Visual Studio 2019 can also be installed for using .NET Core. Visual Studio 2019 is the IDE or the Programming Software for Windows. To install Visual Studio 2019 follow the instructions mentioned at the website https://visualstudio.microsoft.com/

5.4 Building A Console App

1) Open the command prompt and go to the desired folder and type the dotnet new console command as shown in Fig. 5.2. The dotnet new console command is used to generate the standard console application.

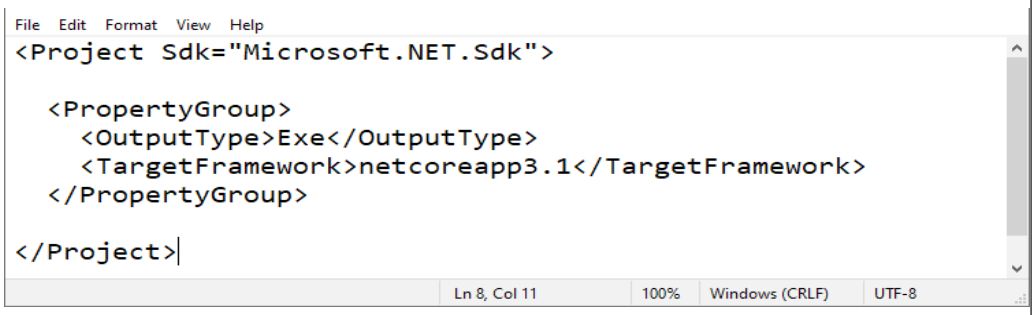
Fig 5.2: Illustration of dotnet new console command for myFirstConsoleApp



2) The folder myFirstConsoleApp consists of two files: the project file which defaults to <directory name>.csproj which in our case is called myFirstConsoleApp.csproj and Program.cs.

3) The .csproj extension file represents a C# project file that contains the list of files included in a project along with the references to system assemblies. The Project element is the root element of every ASP.NET core project file. It includes attributes to specify the entry points for the build process. The PropertyGroup element is used for representing the necessary information required to build a project. The property element name defines the property key and the content of the element defines the property value. The myFirstConsoleApp.csproj file is shown in Fig. 5.3.

Fig 5.3: Code for myFirstConsoleApp.csproj file



4) The Program.cs file contains the method Main, which is the entry point of the ASP.NET Core applications. All the .NET Core applications basically designed as console applications. The Program.cs file is shown in Fig. 5.4.

Fig 5.4: Code for Program.cs file for myFirstConsoleApp

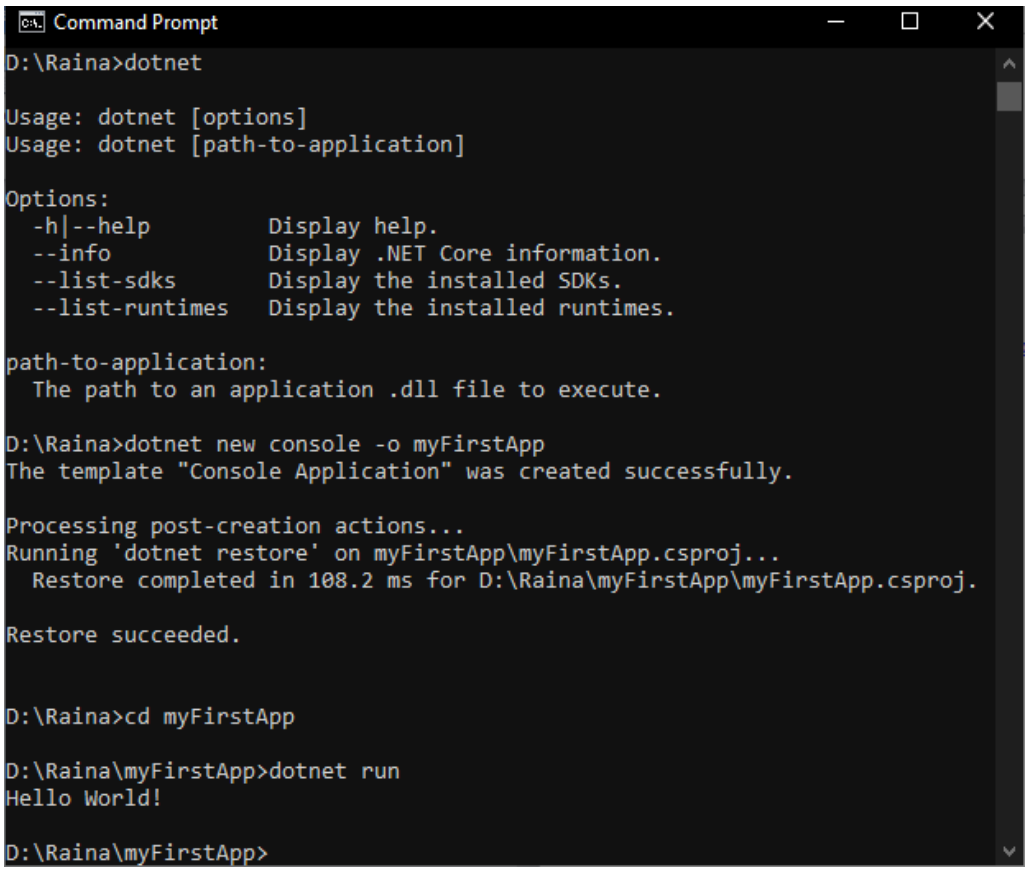
5) Change the directory to the newly created project folder and run the dotnet run command to view the output as shown in Fig. 5.5.

Fig 5.5: Illustration of dotnet run command for myFirstConsoleApp

5.5 Building Asp.net Core App

1) Using the steps discussed in the section 5.3 create another .NET project folder as shown in Fig. 5.6.

Fig 5.6: Illustration of dotnet new console command for myFirstApp



2) Go to the folder myFirstApp and open the file Program.cs in notepad and make the changes in the file as shown in Fig. 5.7.

Fig 5.7: Code for Program.cs file for myFirstApp

Program.cs - Notepad

File Edit Format View Help

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Http;
using System.Configuration;

namespace myFirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            var config = new ConfigurationBuilder()
                .AddCommandLine(args)
                .Build();

            var host = new WebHostBuilder()
                .UseKestrel()
                .UseStartup<Startup>()
                .UseConfiguration(config)
                .Build();

            host.Run();
        }
    }
}
```

In the Main method, the ConfigurationBuilder class is used to accept configuration settings from JSON files, from environment variables, and from the command line. Next the WebHostBuilder class is used to set up the web host. The Internet Information Services (IIS) or the Hostable Web Core (HWC) on Windows will not be used as servers instead a cross-platform, bootstrapped web server called Kestrel will be used. For ASP.NET Core, even if you deploy to Windows and IIS, you will still be using the Kestrel server underneath it all.

3) Go to the folder myFirstApp and create the file Startup.cs in notepad and type the code as shown in Fig. 5.8.

Fig 5.8: Code for Startup.cs file for myFirstApp

Startup.cs - Notepad

File Edit Format View Help

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Http;
using System.Configuration;

namespace myFirstApp
{
        public class Startup
        {
                public Startup(IHostingEnvironment env)
                {
                }
                public void Configure(IApplicationBuilder app,
                                      IHostingEnvironment env,
                                      ILoggerFactory loggerFactory)
                {
                        app.Run(async (context) =>
                        {
                                await context.Response.WriteAsync("Hello, world!");
                        });
                }
        }
}
```

The Startup class is like Global.asax in the traditional .NET application. This class is executed first when the application starts. The startup class can be configured using method UseStartup<T>() at the time of configuring the host in the Main() method of Program Class. The startup class supports a constructor that takes an IHostingEnvironment variable, the Configure method used to configure the HTTP request pipeline and the application and finally the ConfigureServices method, used to add scoped services to the system to be made available via dependency injection.

4) Go to the command prompt and execute the following commands.
   a. dotnet add package Microsoft.AspNetCore.Mvc
   b. dotnet add package Microsoft.AspNetCore.Server.Kestrel
   c. dotnet add package Microsoft.Extensions.Logging
   d. dotnet add package Microsoft.Extensions.Logging.Console
   e. dotnet add package Microsoft.Extensions.Logging.Debug
   f. dotnet add package Microsoft.Extensions.Configuration.CommandLine

5) Fig. 5.9 depicts the file myFirstApp.csproj.

Fig 5.9: Code for myFirstApp.csproj file

myFirstApp.csproj - Notepad

File Edit Format View Help

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
        <PackageReference Include="Microsoft.AspNetCore.Server.Kestrel" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
Version="1.1.1"/>
  </ItemGroup>

</Project>
```

6) Go to the command prompt and execute the following commands.

   a. dotnet restore

   b. dotnet build

   c. dotnet run

   dotnet restore is used to restore or download dependencies for the .NET application. dotnet build is used to compile the application. dotnet run is used to executing the application.

7) Resolve any errors if any. Fig. 5.10 shows the command prompt after the execution of dotnet run command. This indicates the server has started executing.

Fig 5.10: Illustration of dotnet run command for myFirstApp

Command Prompt - dotnet run

```
F:\Raina\MSA\Practicals>dotnet new webapi -o myMicroservice --no-https
The template "ASP.NET Core Web API" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on myMicroservice\myMicroservice.csproj...
  Restore completed in 97.96 ms for F:\Raina\MSA\Practicals\myMicroservice\myMicroservice.csproj.

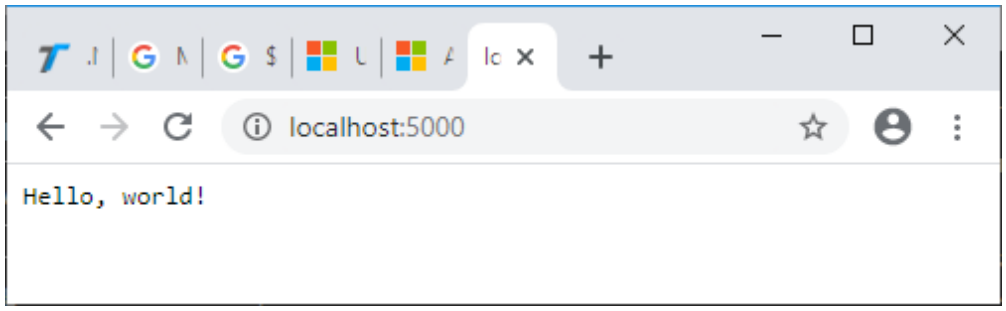
Restore succeeded.


F:\Raina\MSA\Practicals>cd myMicroservice

F:\Raina\MSA\Practicals\myMicroservice>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: F:\Raina\MSA\Practicals\myMicroservice
```

8) Go to the browser and type localhost:5000 in the address bar and hit enter to view the output as shown in Fig. 5.11.

Fig 5.11 Illustration of the output for myFirstApp



5.6 Introduction To Docker

Docker is a software containerization platform which is used for building an application along with packaging them along with their dependencies into a container. These containers can then be easily shipped to run on other machines. Containerization is considered as an evolved version of Virtualization. The same task can also be achieved using Virtual Machines. However it is not very efficient.

Compared to virtual machines containers do not have high overhead and hence enable more efficient usage of the underlying system and resources. Virtual Machines run applications inside a guest Operating System, which runs on virtual hardware powered by the server's host Operating System. Containers leverage the low-level mechanics of the host operating system by providing most of the isolation of virtual machines at a fraction of the computing power.

There are countless platforms and frameworks that either support or integrate tightly with Docker. Docker images can be deployed to AWS (Amazon Web Services), GCP (Google Cloud Platform), Azure, virtual machines, and combinations of those running orchestration platforms like Kubernetes, Docker Swarm, CoreOS Fleet, Mesosphere Marathon, Cloud Foundry etc. The main advantage of Docker is that it works in all of the above environments without changing the container format.

Docker Terminology:-

1. Images – Images are the blueprints of the web application which form the basis of containers.
2. Containers – Containers are created from Docker images and run the actual application.

3. Docker Daemon - Docker Daemon are the background service running on the host that manages building, running and distributing Docker containers.
4. Docker Client - Docker Client is the command line tool that allows the user to interact with the daemon.
5. Docker Hub - Docker Hub is a registry of Docker images. It is a directory of all available Docker images. Anyone can host their own Docker registries and use them for pulling images.

5.7 Installing Docker

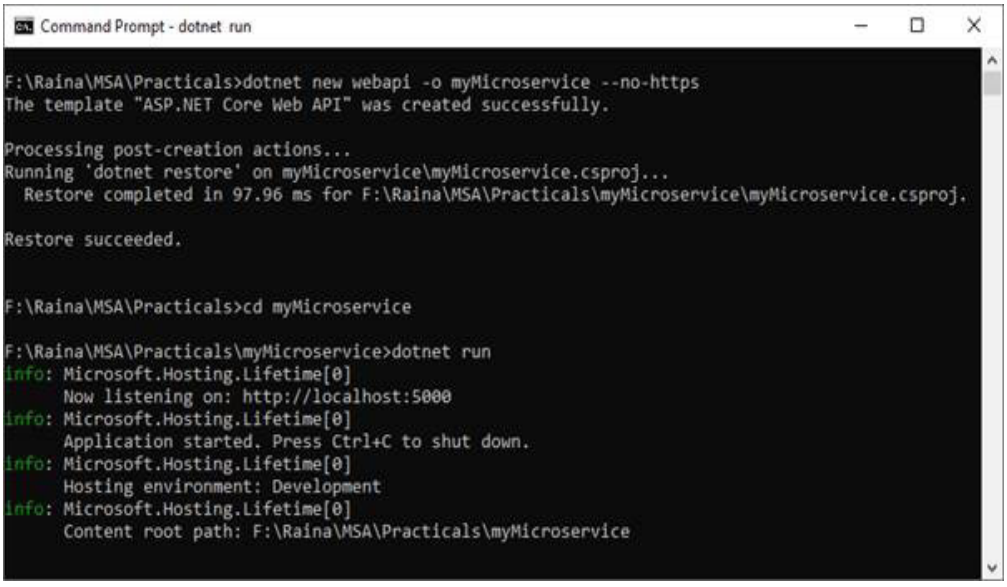
To install Docker on Windows OS Docker Desktop software can be used. Docker Desktop is the Community version of Docker for Microsoft Windows. Docker Desktop for Windows can be downloaded from Docker Hub and installed or follow the instructions on the website https://docs.docker.com/docker-for-windows/install/ for complete installation process.

Docker Toolbox is another software which is used for older Mac and Windows systems that do not support the requirements of Docker Desktop for Mac and Docker Desktop for Windows. Follow the instructions on the website https://docs.docker.com/toolbox/toolbox_install_windows/ for complete installation procedure.

5.8 Running Asp.net Core App Using Docker

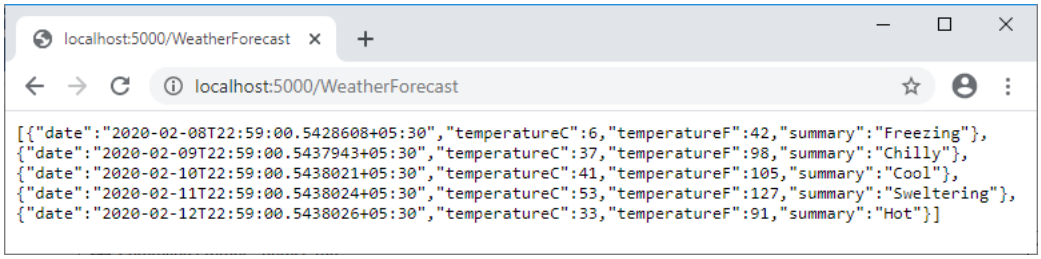
1) Open the command prompt and type the commands as shown in Fig. 5.12

Fig 5.12: Illustration of dotnet commands for myMicroservice Application



2) Go to the browser and type localhost:5000/WeatherForecast in the address bar and hit enter to view the output as shown in Fig. 5.13.

Fig 5.13: Output for myMicroservice



3) Create the files Dockerfile and .dockerignore as shown in Fig. 5.14 and Fig. 5.14a in the folder "myMicroservice". Remember to save the files in between "" while saving because they are extension less and nameless files.

Fig 5.14: Code for Dockerfile

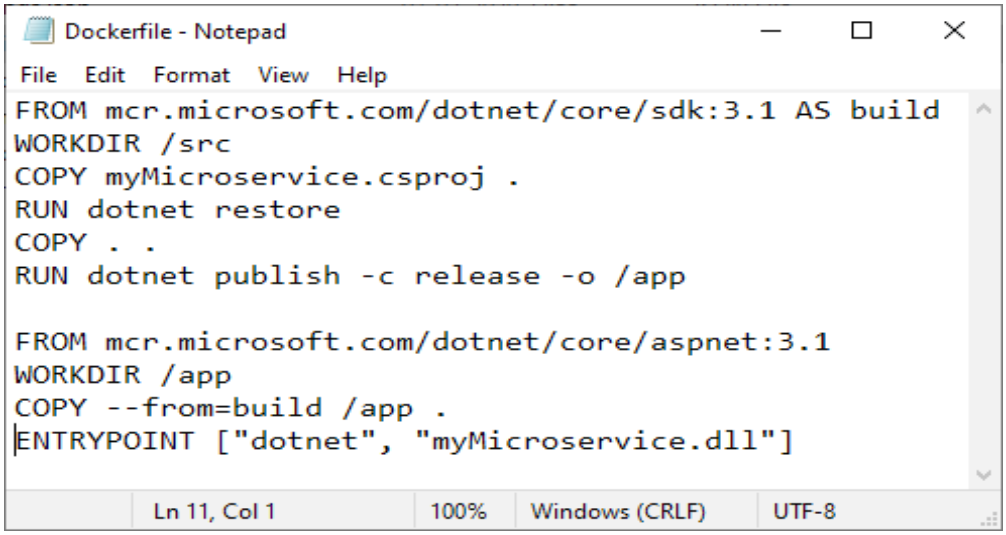
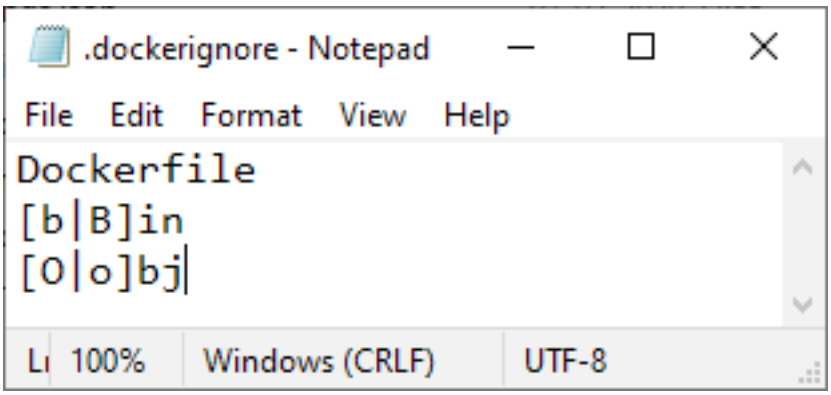


Fig 5.14a: Code for .dockerignore



4) Start the Docker Desktop software. Open command prompt and type the commands as shown in Fig. 5.15(a) and Fig. 5.15(b) .

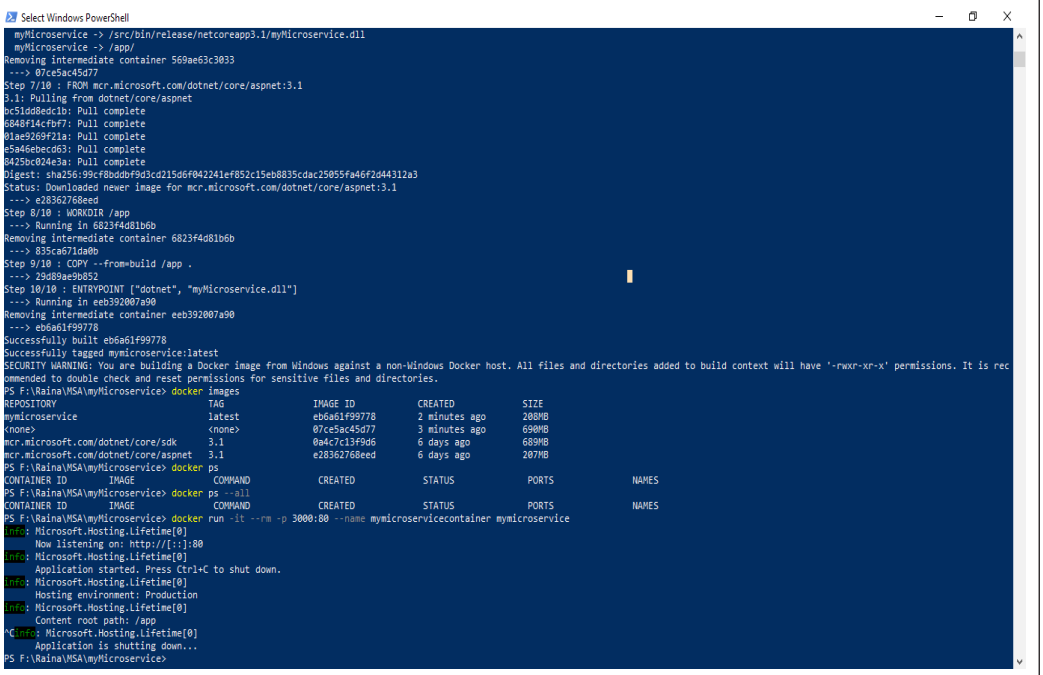
Fig 5.15(a) Docker commands for myMicroservice

Select Windows PowerShell

```
PS F:\Raina\MSA\myMicroservice> docker build -t mymicroservice .
Sending build context to Docker daemon  15.36kB
Step 1/10 : FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
3.1: Pulling from dotnet/core/sdk
dc65f448a2e2: Pull complete
346ffb2b67d7: Pull complete
dea4ecac934f: Pull complete
8ac92ddf84b3: Pull complete
d6bef01952b9: Pull complete
9311becdb99f: Pull complete
cdbf4d78a668: Pull complete
Digest: sha256:f4798993581a660740f06bdc126cf8258788a67518fb43e4275ecc6d4b226211
Status: Downloaded newer image for mcr.microsoft.com/dotnet/core/sdk:3.1
 ---> 0a4c7c13f9d6
Step 2/10 : WORKDIR /src
 ---> Running in 35a842fae8d3
Removing intermediate container 35a842fae8d3
 ---> 16d4523b7be4
Step 3/10 : COPY myMicroservice.csproj .
 ---> be91adc3e2f6
Step 4/10 : RUN dotnet restore
 ---> Running in 245035c960f9
  Restore completed in 130.03 ms for /src/myMicroservice.csproj.
Removing intermediate container 245035c960f9
 ---> 411b8b176168
Step 5/10 : COPY . .
 ---> 92029f3d3a15
Step 6/10 : RUN dotnet publish -c release -o /app
 ---> Running in 569ae63c3033
Microsoft (R) Build Engine version 16.4.0+e901037fe for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

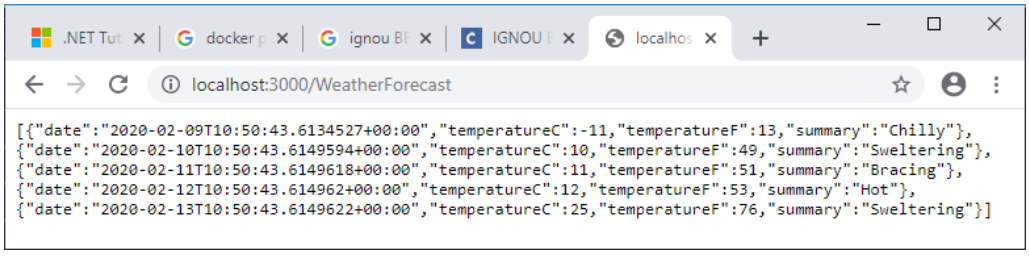
  Restore completed in 24.96 ms for /src/myMicroservice.csproj.
  myMicroservice -> /src/bin/release/netcoreapp3.1/myMicroservice.dll
  myMicroservice -> /app/
Removing intermediate container 569ae63c3033
 ---> 07ce5ac45d77
Step 7/10 : FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
3.1: Pulling from dotnet/core/aspnet
bc51dd8edc1b: Pull complete
6848f14cfbf7: Pull complete
01ae9269f21a: Pull complete
e5a46ebecd63: Pull complete
8425bc024e3a: Pull complete
Digest: sha256:99cf8bddbf9d3cd215d6f042241ef852c15eb8835cdac25055fa46f2d44312a3
Status: Downloaded newer image for mcr.microsoft.com/dotnet/core/aspnet:3.1
 ---> e28362768eed
Step 8/10 : WORKDIR /app
 ---> Running in 6823f4d81b6b
Removing intermediate container 6823f4d81b6b
```

Fig 5.15(b) Docker commands for myMicroservice



5) Go to the browser and type localhost:3000/WeatherForecast in the address bar and hit enter to view the output as shown in Fig. 5.16. This output comes from executing the Docker image.

Fig 5.16 Output for myMicroservice by executing Docker commands



5.9 Continuous Integration With Wercker

Wercker is a Docker based continuous delivery platform which is used by software developers to build and deploy their applications and Microservices. Using Wercker developers can create Docker containers on their desktop, automate their build and deploy processes, testing them on their desktop, and then deploy them to various cloud platforms like Heroku, AWS and Rackspace. The command-line interface of Wercker is open-sourced. The business behind Wercker, also called Wercker, was founded in 2012 and acquired by Oracle Corporation in 2017.

Installing the Wercker CLI

After installing Docker and Docker-machine, you have to create a new virtual machine that will run Docker by using the following command:

docker-machine create -- driver virtualbox dev

Once the VM is created, you have to export some variables to your environment:

eval "$(docker-machine env dev)"

The environment is now set up, you can install wercker using brew by using the following commands.

```
brew tap wercker/wercker
brew install wercker-cli
```

You can also install the CLI manually:

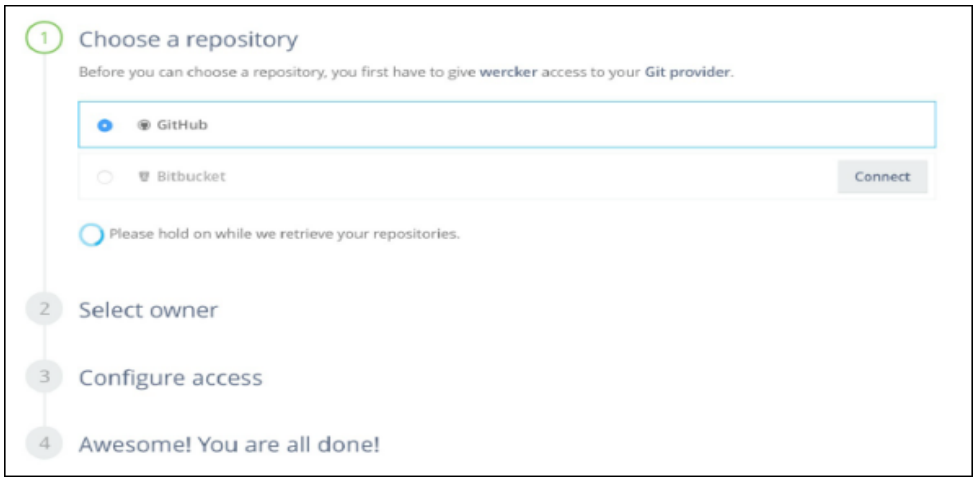
```
curl -L https://s3.amazonaws.com/downloads.wercker.com/cli/stable/darwin_amd64/wercke
/usr/local/bin/wercker
```

You can check the version of Wrecker installed by using the command wercker version.

There are three basic steps for using Wercker:-

1. **Create an application in Wercker using the website** - First sign up for an account by logging in with your existing GitHub account. Once you've got an account and you're logged in, click the Create link in the top menu. This will bring up a wizard as shown in Fig 5.17. The wizard will prompt you to choose a GitHub repository as the source for your build. It will then ask you whether you want the owner of this application build to be your personal account or an organization to which you belong.

Fig 5.17: Wizard for Wercker Application



2. **Add a wercker.yml file to your application's codebase** – Fig. 5.18 depicts the code of wercker.yml file:-

Fig 5.18: Code for wercker.yml

The box property indicates the base docker hub image that we're going to use as a starting point. Following commands are executed in this container - dotnet restore, dotnet build and dotnet publish.

3. **Choose how to package and where to deploy successful builds** - There is a script within the application as shown in Fig. 5.19 to invoke the Wercker build command.

Fig 5.19 Script for invoking Wercker build command

```
rm -rf _builds _steps _projects
wercker build --git-domain github.com \
  --git-owner microservices-aspnetcore \
  --git-repository hello-world
rm -rf _builds _steps _projects
```

This will execute the Wercker build exactly as it executes in the cloud, all within the confines of a container image. A bunch of messages will be seen from the Wercker pipeline, including the latest version of the .NET Core Docker image and running all of the steps in the pipeline.

5.10 Continuous Integration with Circleci

CircleCI is a modern platform used for continuous integration and continuous delivery i.e. CI/CD platform. The CircleCI Enterprise solution is installable inside a private cloud or a data center and is free to try for a limited time. CircleCI automates build, test, and deployment of software. CircleCI can also be configured to deploy code to various environments like AWS CodeDeploy, AWS EC2 Container Service (ECS), AWS S3, Google Container Engine (GKE), Heroku etc.

Check the website http://circleci.com and sign up with a new account for free or log in using your GitHub account. We first need to set up a configuration file circle.yml to tell CircleCI how to build the app. Fig. 5.20 depicts the code for circle.yml file.

Fig. 5.20: Code for circle.yml

```
machine:
  pre:
    - sudo sh -c 'echo "deb [arch=amd64] https://apt-
mo.trafficmanager.net/repos/
    dotnet-release/ trusty
main" > /etc/apt/sources.list.d/dotnetdev.list'
    - sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
recv-keys
    417A0893
    - sudo apt-get update
    - sudo apt-get install dotnet-dev-1.0.1

compile:
  override:
    - dotnet restore
    - dotnet build
    - dotnet publish -o publish

test:
  override:
    - echo "no tests"
```

Fig. 5.21 shows the CircleCI dashboard for the application.

Fig. 5.21: CircleCI dashboard for the application

5.11 Deploying to Docker Hub.

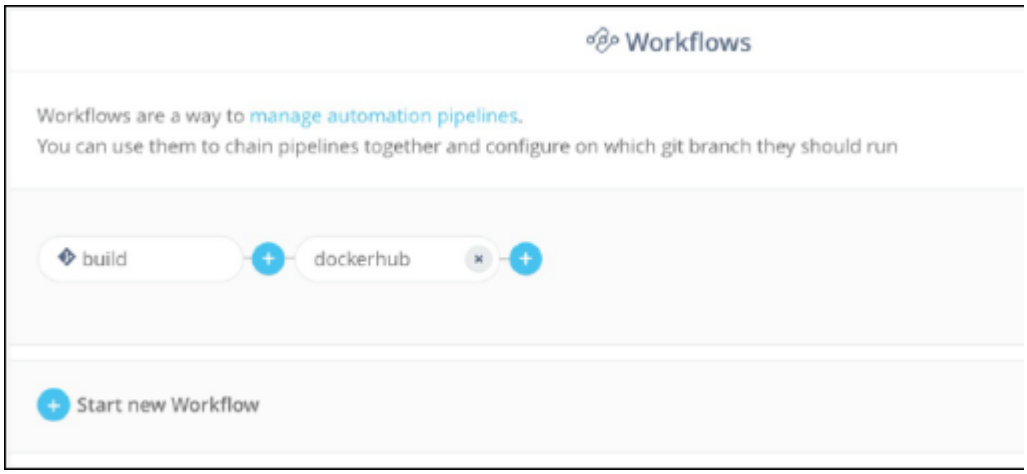
Once the Wercker or CircleCI build is ready that is producing a Docker image and all the tests are passing, it can be deployed to docker hub. Fig. 5.22 shows the deploy section of the wercker.yml file that when executed will deploy the build artifact as a docker hub image.

Fig. 5.22: Deploy section of the wercker.yml file

```
deploy:
  steps:
    - internal/docker-push:
        username: $USERNAME
        password: $PASSWORD
        repository: dotnetcoreservices/hello-world
        registry: https://registry.hub.docker.com
        entrypoint: "/pipeline/source/app/docker_entrypoint.sh"
```

Fig. 5.23 depicts a sample Wercker workflow. The docker hub section of this pipeline is easily created by clicking the "+" button in the GUI and giving the name of the YAML section for deployment.

Fig. 5.23: A sample Wercker workflow



5.12 Summary

1) This chapter introduces the concepts of .NET Core and how it is used for designing web applications. This chapter also introduced the technology of ASP.NET Core used along with C# language and how to build console and web applications

2) The concept of Docker was also introduced along with it uses and commands and how it is used with Wercker and CircleCI and finally how the web application is deployed on Docker Hub.

5.13 Review Questions

1) What is .NET Core? Explain its components?

2) What is ASP.NET Core? Explain the steps for installing .NET Core.

3) Explain the steps to build a console application using ASP.NET Core.

4) Explain the steps to build a console based web application using ASP.NET Core.

5) What is Docker? Explain the Docker terminology. Explain Docker installation process.

6) Explain the steps to build a console based web application using ASP.NET Core and Docker.

7) Explain the process of continuous integration using Wercker.

8) Explain the process of continuous integration using CircleCI.

5.14 Bibliography, References And Further Reading

1) Building Microservices with ASP.NET Core by Kevin Hoffman

2) LittleAspNetCoreBook by Nate Barbettini

3) Microservices for the Enterprise by Kasun Indrasiri and Prabath Siriwardena

Unit 3

6

INTRODUCTION TO MICROSERVICES

Chapter Structure:

6.0 Objectives

This chapter will make you understand the basic concepts of Microservices and how to develop a Microservice using ASP.NET Core.

6.1 Introduction to Microservices

Microservices is an architectural style pattern where an application is designed as a collection of small autonomous services which is modeled around a business domain. Each service is self-contained and implements a single business capability. The main idea behind Microservices is that applications become easier to build and maintain when they are broken down into smaller pieces which work together. Each component is continuously developed and separately maintained, and the application is then simply the sum of its constituent components. This is in stark

contrast to the traditional monolithic application which is developed all in one piece and tightly coupled in nature. Benefits of using Microservices include Developer independence, Isolation and resilience, Scalability, Lifecycle automation etc.

6.2 Introduction to Team Service

Every company requires teams whether it is sales teams, development teams, support etc. Companies need to keep track of those members - their locations, contact information, project assignments etc. The team service is used for solving this problem. This service allows clients to query team lists as well as team members and their details. It is also possible to add or remove teams and team members.

6.3 API First Development for Team Service

Table 6.1 describes the Team Service API.

Resource	Method	Description
`/teams`	GET	Gets a list of all teams
`/teams/{id}`	GET	Gets details for a single team
`/teams/{id}/members`	GET	Gets members of a team
`/teams`	POST	Creates a new team
`/teams/{id}/members`	POST	Adds a member to a team
`/teams/{id}`	PUT	Updates team properties
`/teams/{id}/members/{memberId}`	PUT	Updates member properties
`/teams/{id}/members/{memberId}`	DELETE	Removes a member from the team
`/teams/{id}`	DELETE	Deletes an entire team

In this chapter we are building applications for a fictitious company called the BajiRainaBanu Corporation. Thus the team service will be in a project called BajiRainaBanuCorp.TeamService and the tests will be in BajiRainaBanuCorp.TeamService.Tests. The main project will be in src/BajiRainaBanuCorp.TeamService and the test project will be in test/BajiRainaBanuCorp.TeamService.Tests.

Fig. 6.1 illustrates the XML code for the BajiRainaBanu.TeamService.Tests.csproj project file.

Fig. 6.1 XML code for BajiRainaBanu.TeamService.Tests.csproj

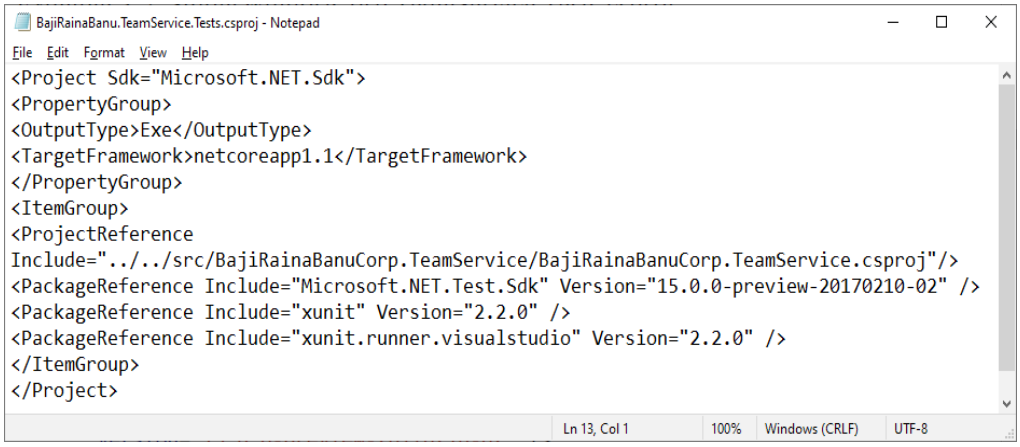
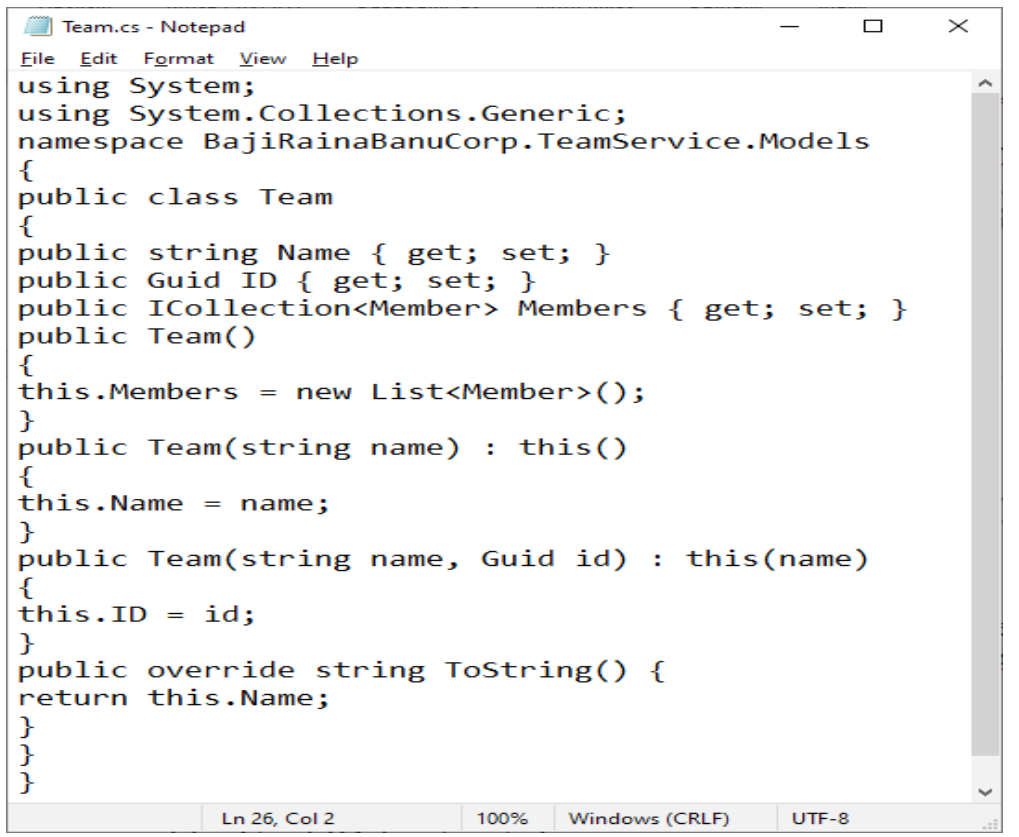


Fig. 6.2 illustrates the class code for src/BajiRainaBanuCorp.TeamService/Models/Team.cs Team model:-

Fig. 6.2 Class code for src/BajiRainaBanuCorp.TeamService/Models/Team.cs



Each team is going to need a collection of Member objects in order to Compile. Fig. 6.3 illustrates the code for src/BajiRainaBanuCorp.TeamService/Controllers/TeamsController.cs. This is the controller file of the project.

Fig. 6.3 Code for src/BajiRainaBanuCorp.TeamService/Models/Member.cs

Member.cs - Notepad

File Edit Format View Help

```
using System;
namespace BajiRainaBanuCorp.TeamService.Models
{
public class Member {
public Guid ID { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public Member() {
}
public Member(Guid id) : this() {
this.ID = id;
}
public Member(string firstName,
string lastName, Guid id) : this(id) {
this.FirstName = firstName;
this.LastName = lastName;
}
public override string ToString() {
return this.LastName;
}
}
}
```

Ln 22, Col 2 100% Windows (CRLF) UTF-8

Fig. 6.4 illustrates the code for src/BajiRainaBanuCorp.TeamService/Controllers/TeamsController.cs. This is the controller file of the project.

Fig. 6.4 Code for src/BajiRainaBanuCorp.TeamService/Controllers/TeamsController.cs

ITeamRepository.cs - Notepad

File Edit Format View Help

```
using System.Collections.Generic;
namespace BajiRainaBanuCorp.TeamService.Persistence
{
public interface ITeamRepository
{
IEnumerable<Team> GetTeams();
void AddTeam(Team team);
}
}
```

Ln 9, Col 2 100% Windows (CRLF) UTF-8

Next we will create an interface called ITeamRepository which is the interface that will be used by the project. Fig. 6.5 illustrates the code for src/BajiRainaBanuCorp.TeamService/Persistence/ITeamRepository.cs

Fig. 6.5 Code for src/BajiRainaBanuCorp.TeamService/Persistence/ITeamRepository.cs

ITeamRepository.cs - Notepad

File Edit Format View Help

```
using System.Collections.Generic;
namespace BajiRainaBanuCorp.TeamService.Persistence
{
public interface ITeamRepository
{
IEnumerable<Team> GetTeams();
void AddTeam(Team team);
}
}
```

Ln 9, Col 2 100% Windows (CRLF) UTF-8

Next we will create an inmemory implementation of this repository interface in the service project. Fig. 6.6 illustrates the code for src/BajiRainaBanuCorp. TeamService/Persistence/MemoryTeamRepository.cs

Fig. 6.6 Code for src/BajiRainaBanuCorp.TeamService/Persistence/ MemoryTeamRepository.cs

MemoryTeamRepository.cs - Notepad

File Edit Format View Help

```
using System.Collections.Generic;
namespace BajiRainaBanuCorp.TeamService.Persistence
{
public class MemoryTeamRepository : ITeamRepository {
protected static ICollection<Team> teams;
public MemoryTeamRepository() {
if(teams == null) {
teams = new List<Team>();
}
}
public MemoryTeamRepository(ICollection<Team> teams) {
teams = teams;
}
public IEnumerable<Team> GetTeams() {
return teams;
}
public void AddTeam(Team t)
{
teams.Add(t);
}
}
}
```

Ln 23, Col 1 100% Windows (CRLF) UTF-8

Using the concept of Dependency Injection we are going to add the repository as a service in the Startup class, as shown in the Fig. 6.7

Fig. 6.7 Concept of Dependency Injection used in the Startup class

Startup.cs - Notepad

File Edit Format View Help

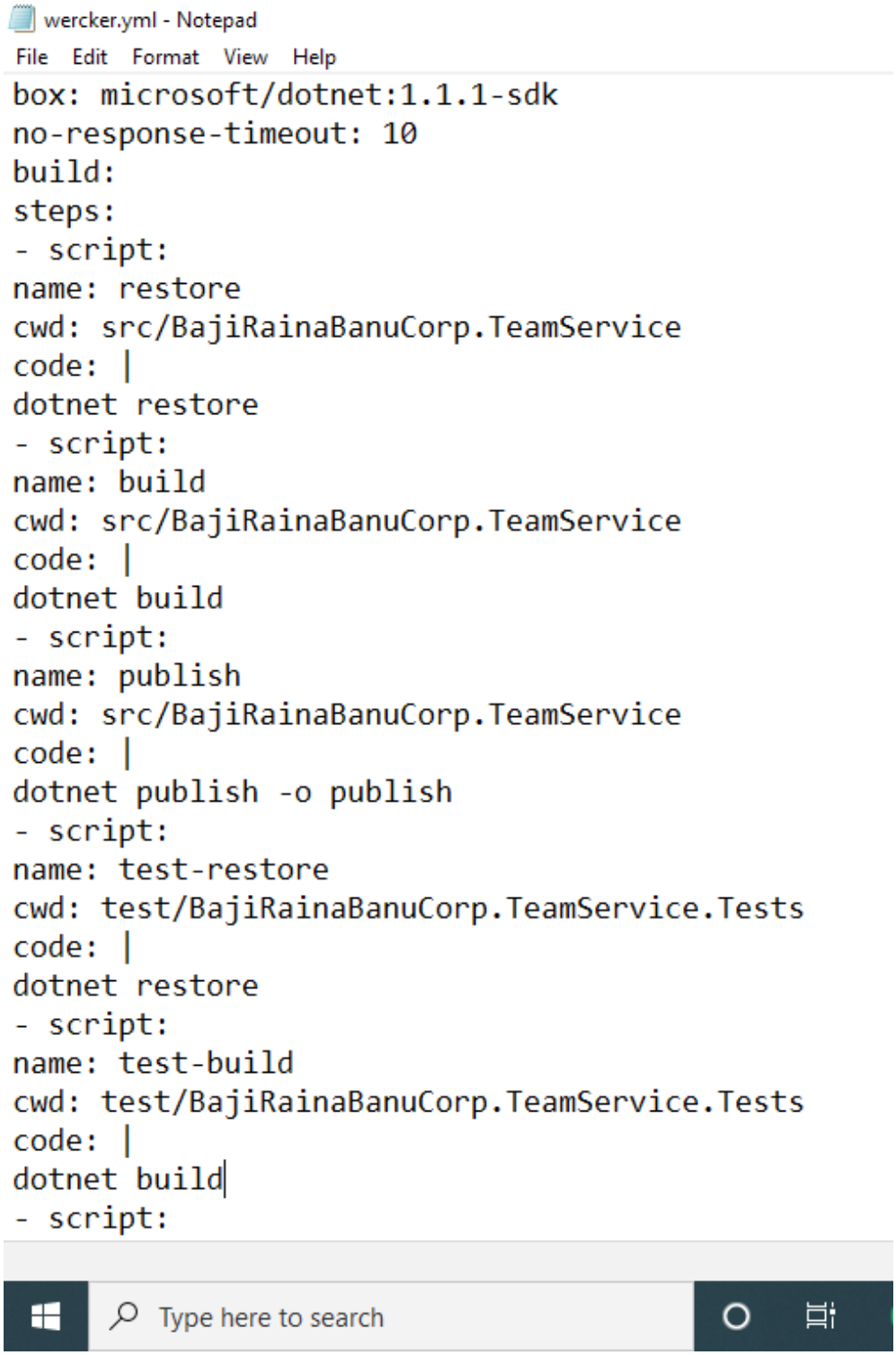
```
public void ConfigureServices(IServiceCollection services)
{
services.AddMvc();
services.AddScoped<ITeamRepository, MemoryTeamRepository>();
}
```

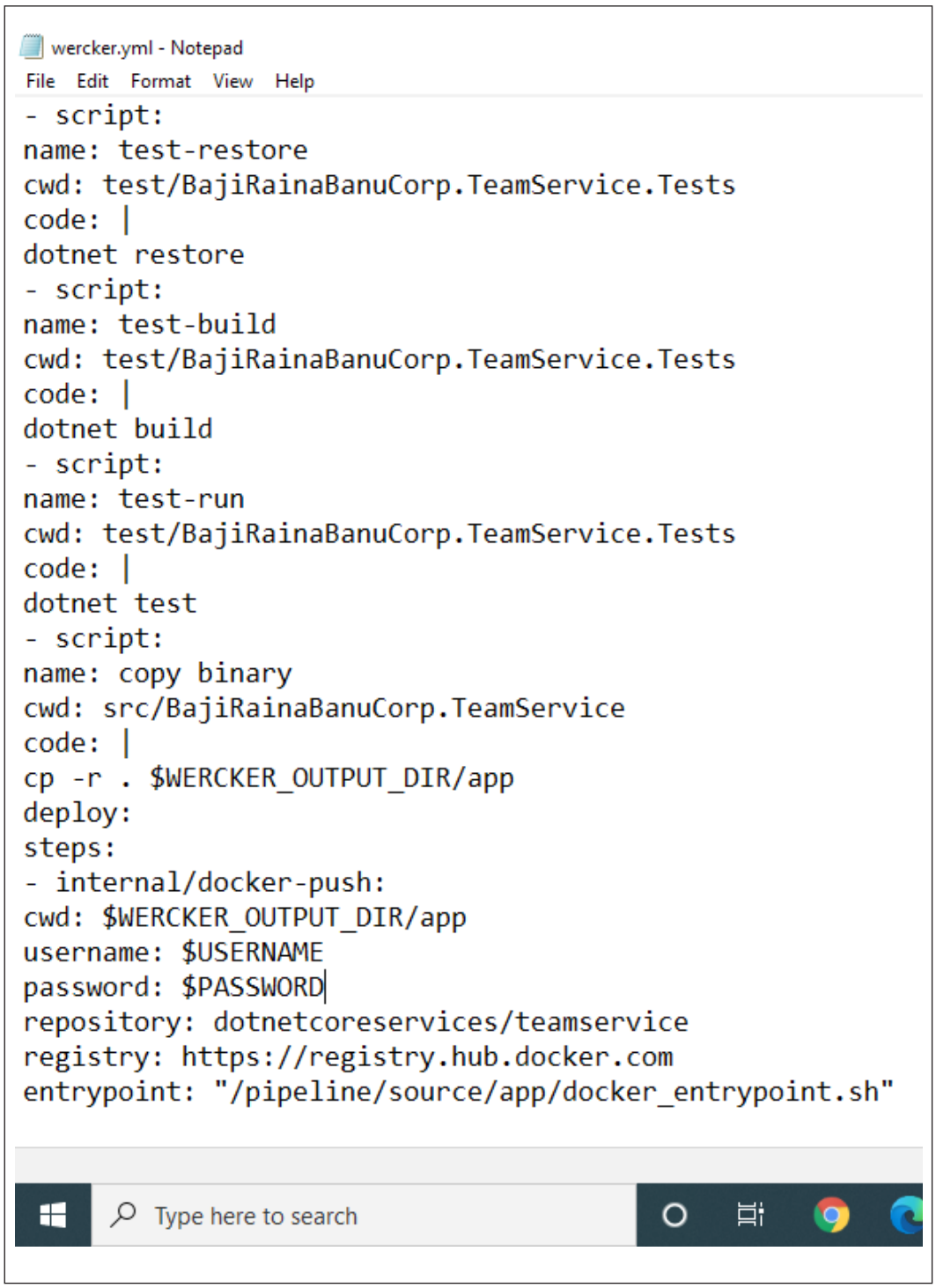
Ln 6, Col 1 100% Windows (CRLF) UTF-8

6.4 Creating a CI Pipeline

Fig. 6.8 illustrates the code for wercker.yml file for the team service.

Fig. 6.8 Code for wercker.yml file




wercker.yml - Notepad
File Edit Format View Help
- script:
name: test-restore
cwd: test/BajiRainaBanuCorp.TeamService.Tests
code: |
dotnet restore
- script:
name: test-build
cwd: test/BajiRainaBanuCorp.TeamService.Tests
code: |
dotnet build
- script:
name: test-run
cwd: test/BajiRainaBanuCorp.TeamService.Tests
code: |
dotnet test
- script:
name: copy binary
cwd: src/BajiRainaBanuCorp.TeamService
code: |
cp -r . $WERCKER_OUTPUT_DIR/app
deploy:
steps:
- internal/docker-push:
cwd: $WERCKER_OUTPUT_DIR/app
username: $USERNAME
password: $PASSWORD
repository: dotnetcoreservices/teamservice
registry: https://registry.hub.docker.com
entrypoint: "/pipeline/source/app/docker_entrypoint.sh"
Type here to search

6.5 Integration Testing

An integration test is used to verify that all of the components of the system are connected and you get suitable responses as expected from the system. Fig. 6.9 illustrates the code for

test/BajiRainaBanuCorp.TeamService.Tests.Integration/SimpleIntegrationTest.cs

Fig. 6.9 Code for test/BajiRainaBanuCorp.TeamService.Tests.Integration/SimpleIntegrationTest.cs

myFirstApp.csproj - Notepad

File Edit Format View Help

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
        <PackageReference Include="Microsoft.AspNetCore.Server.Kestrel" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1"/>
        <PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
Version="1.1.1"/>
  </ItemGroup>

</Project>
```

6.6 Running the Team Service Docker Image

Now that the CI pipeline is working for the team service, it should automatically be deploying a Docker image to docker hub. Execute the docker command as shown in Fig. 6.10 to execute the docker image.

Fig. 6.10 Docker run command for teamservice

Tools | Building Microservi... | Sign

```
$ docker run -p 8080:8080 dotnetcoreservices/teamservice
Unable to find image 'dotnetcoreservices/teamservice:latest' locally
latest: Pulling from dotnetcoreservices/teamservice
693502eb7dfb: Already exists
081cd4bfd521: Already exists
5d2dc01312f3: Already exists
36c0e9895097: Already exists
3a6b0262adbb: Already exists
79e416d3fe9d: Already exists
d96153ed695f: Pull complete
Digest:
sha256:fc3ea65afe84c33f5644bbec0976b4d2d9bc943ddba997103dd3fb731f56ca5b
Status: Downloaded newer image for
dotnetcoreservices/teamservice:latest
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:8080
Application started. Press Ctrl+C to shut down.
```

Fig. 6.11 illustrates the curl command in Ubuntu which issues a POST to the /teams resource of the service. This will return the following JSON payload containing the newly created team:

Fig. 6.11 The curl command in Ubuntu for teamservice

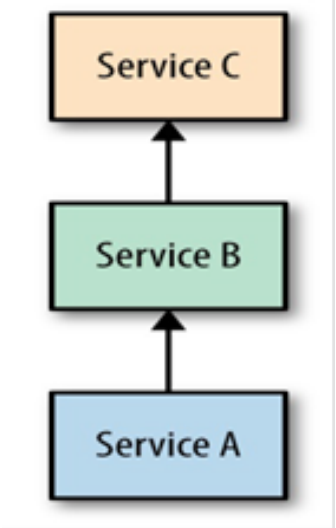
```
$ curl -H "Content-Type:application/json" \
   -X POST -d \
   '{"id":"e52baa63-d511-417e-9e54-7aab04286281", \
    "name":"Team Zombie"}' \
    http://localhost:8080/teams

{"name":"Team Zombie","id":"e52baa63-d511-417e-9e54-7aab04286281",
  "members":[]}
```

6.7 Microservices Ecosystems

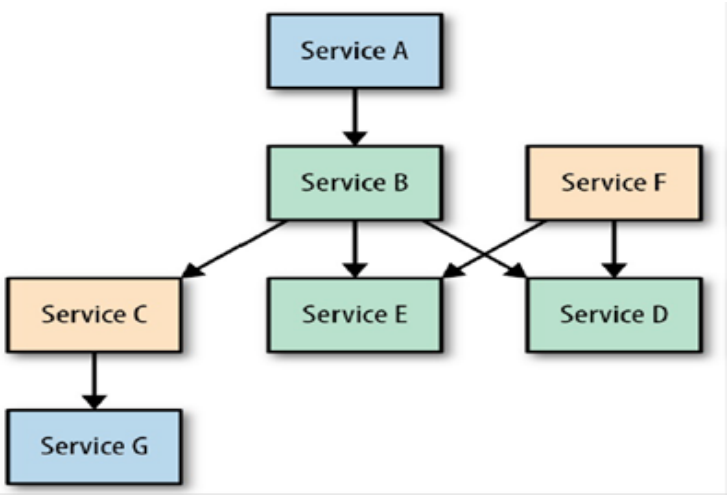
Fig 6.12 shows a simple Microservice ecosystem where service A depends on B which in turn depends on C. The hierarchy in this scenario is very clear and also unrealistic. Organizations must never assume that there is going to be a clear dependency chain or hierarchy of Microservices.

Fig 6.12 Simple Microservice ecosystem



Instead organizations must plan for a hierarchy which looks something like as shown in Fig. 6.13. In this ecosystem, we have a better representation of reality where no proper hierarchy is followed. Any Microservice can depend on other Microservice irrespective of any hierarchy.

Fig 6.13 Realistic microservice ecosystem



6.8 Building the Location Service

In section 6.2 and 6.3 we have seen how to build the team service. In this section we will see how to maintain and query the locations of all of the team members. Location management will be done through a separate service which is called as the location service that will manage the location history of the team members without regards for their team membership. Table 6.2 describes the LocationService API.

Table 6.2 Location Service API

Resource	Method	Description
`/locations/{memberID}/latest`	GET	Retrieves the most current location of a member
`/locations/{memberID}`	POST	Adds a location record to a member
`/locations/{memberID}`	GET	Retrieves the location history of a member

Fig. 6.14 illustrates the code for src/BajiRainaBanuCorp.LocationService/Models/LocationRecord.cs

Fig. 6.14 Code for src/BajiRainaBanuCorp.LocationService/Models/LocationRecord.cs

LocationRecord.cs - Notepad

File Edit Format View Help

```
public class LocationRecord
{
public Guid ID { get; set; }
public float Latitude { get; set; }
public float Longitude { get; set; }
public float Altitude { get; set; }
public long Timestamp { get; set; }
public Guid MemberID { get; set; }
}
```

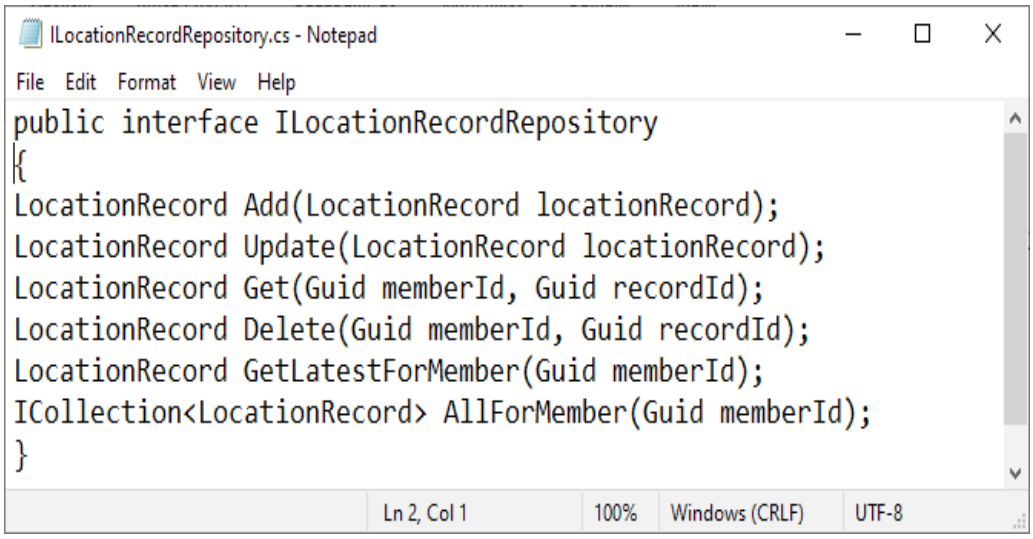
Ln 10, Col 1 100% Windows (CRLF) UTF-8

In Fig 6.14 each location record is uniquely identified by a GUID called ID. This record contains a set of coordinates for latitude, longitude, and altitude; the timestamp for when the location event took place; and the GUID of the individual member involved.

Next we require a interface which is a simple inmemory system representing the contract for a location repository. Fig. 6.15 illustrates the code for

src/BajiRainaBanuCorp.LocationService/Models/ILocationRecordRepository.cs

Fig. 6.15 Code for src/BajiRainaBanuCorp.LocationService/Models/ILocationRecordRepository.cs



Next we will create another controller that exposes this public API. Fig. 6.16 illustrates that the controller accepts an ILocationRecordRepository instance via constructor injection.

Fig. 6.16 Code for src/BajiRainaBanuCorp.LocationService/Controllers/ LocationRecordController.cs

LocationRecordController.cs - Notepad

File Edit Format View Help

```
[Route("locations/{memberId}")]
public class LocationRecordController : Controller {
private ILocationRecordRepository locationRepository;
public LocationRecordController(
ILocationRecordRepository repository) {
this.locationRepository = repository;
}
[HttpPost]
public IActionResult AddLocation(Guid memberId,
[FromBody]LocationRecord locationRecord) {
locationRepository.Add(locationRecord);
return this.Created(
$"/locations/{memberId}/{locationRecord.ID}",
locationRecord);
}
[HttpGet]
public IActionResult GetLocationsForMember(Guid memberId) {
return this.Ok(locationRepository.AllForMember(memberId));
}
[HttpGet("latest")]
public IActionResult GetLatestForMember(Guid memberId) {
return this.Ok(
locationRepository.GetLatestForMember(memberId));
}
}
```

Ln 15, Col 2 100% Windows (CRLF) UTF-8

Next make the repository available for dependency injection by adding it as a scoped service in the startup class. Fig. 6.17 illustrates the code for Startup.cs.

Fig. 6.17 Concept of Dependency Injection used in the Startup class

*Startup.cs - Notepad

File Edit Format View Help

```
services.AddScoped<ITeamRepository, MemoryTeamRepository>();
}


public void ConfigureServices(IServiceCollection services)
{
services.AddScoped<ILocationRecordRepository,
InMemoryLocationRecordRepository>();
services.AddMvc();
}
```

Next run the application as shown in Fig. 6.18.

Fig. 6.18 Illustration of dotnet run command for location service

```
$ dotnet run
Hosting environment: Production
Content root path: [...]
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Next we POST a new location record using the syntax shown in Fig. 6.19.

Fig. 6.19 Illustration of curl command for location service

```
$ curl -H "Content-Type: application/json" -X POST
   -d '{"id": "55bf35ba-deb7-4708-abc2-a21054dbfa13", \
        "latitude": 12.56, "longitude": 45.567, \
        "altitude": 1200, "timestamp" : 1476029596, \
        "memberId": "0edaf3d2-5f5f-4e13-ae27-a7fbea9fccfb" }'
 http://localhost:5000/locations/0edaf3d2-5f5f-4e13-ae27-a7fbea9fccfb

{"id":"55bf35ba-deb7-4708-abc2-a21054dbfa13",
  "latitude":12.56,"longitude":45.567,
  "altitude":1200.0,"timestamp":1476029596,
  "memberID":"0edaf3d2-5f5f-4e13-ae27-a7fbea9fccfb"}
```

We receive back the location record we submitted indicating that a new record was created. Next we query the location history for our member using the same memberId with the command as shown in Fig. 6.20.

Fig. 6.20 Illustration of curl command for location service

```
$ curl http://localhost:5000/locations/0edaf3d2-5f5f-4e13-ae27-
a7fbea9fccfb

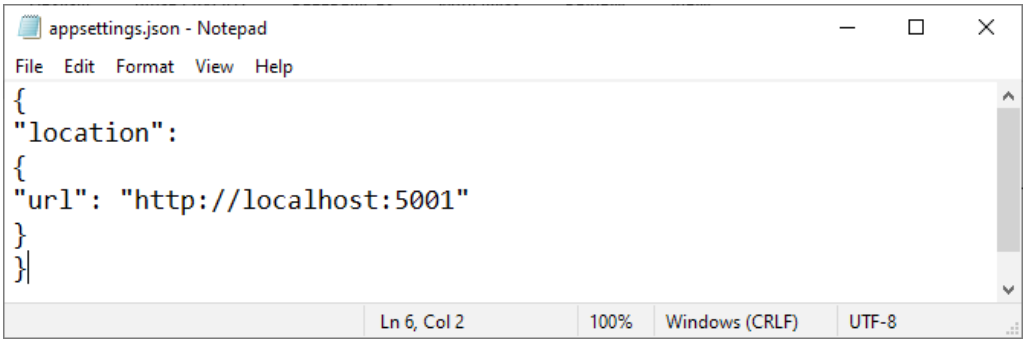
[
{"id":"55bf35ba-deb7-4708-abc2-a21054dbfa13",
   "latitude":12.56,"longitude":45.567,"altitude":1200.0,
   "timestamp":1476029596,
   "memberID":"0edaf3d2-5f5f-4e13-ae27-a7fbea9fccfb"}
]
```

6.9 Enhancing Team Service

In this section we will modify the team service created in section 6.4 so that when we query the details for a particular team member, their current location can also be included when they are checked into that location.

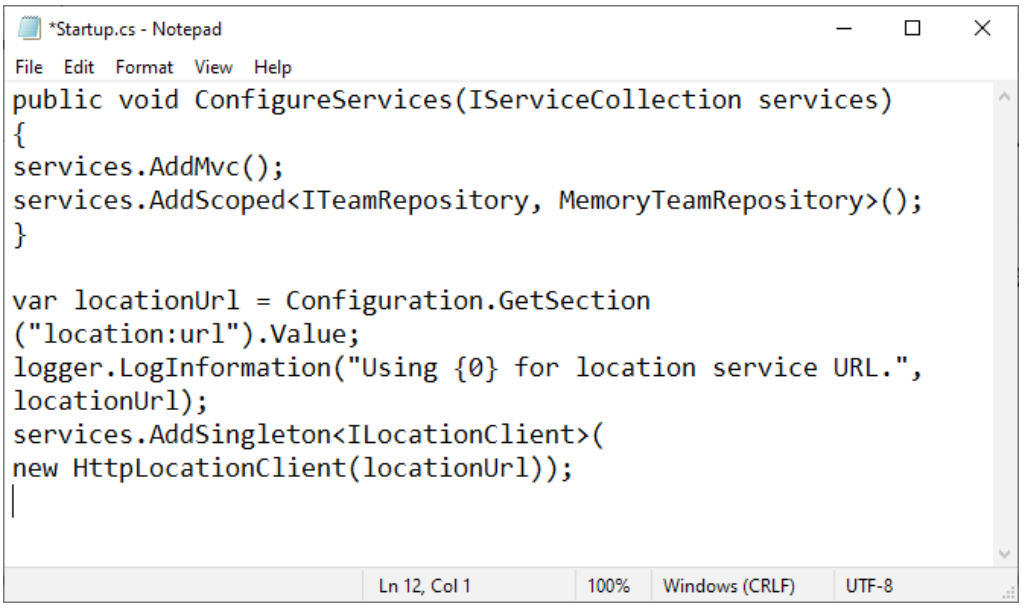
First we configure Service URLs with Environment Variables. For this open the appsettings.json file and set the default variables as shown in Fig. 6.21

Fig. 6.21 Code for appsettings.json file



Modify the startup file as shown in Fig. 6.22 so that we can register an HttpLocationClient instance with the appropriate URL.

Fig. 6.22 Code for Startup class



Next we create an interface for our location client as shown in Fig. 6.23.

Fig. 6.23 Code for src/BajiRainaBanuCorp.TeamService/LocationClient/ILocationClient.cs

ILocationClient.cs - Notepad

File Edit Format View Help

```
public interface ILocationClient
{
Task<LocationRecord> GetLatestForMember(Guid memberId);
}
```

Ln 4, Col 2 100% Windows (CRLF) UTF-8

Next we create an implementation of a location client, as shown in Fig. 6.24 that makes simple HTTP requests.

Fig. 6.24 Code for src/BajiRainaBanuCorp.TeamService/LocationClient/HttpLocationClient.cs

HttpLocationClient.cs - Notepad

File Edit Format View Help

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using BajiRainaBanuCorp.TeamService.Models;
using Newtonsoft.Json;
namespace BajiRainaBanuCorp.TeamService.LocationClient
{
public class HttpLocationClient : ILocationClient
{
public String URL {get; set;}
public HttpLocationClient(string url){  this.URL = url;   }
public async Task<LocationRecord>
GetLatestForMember(Guid memberId)
{
LocationRecord locationRecord = null;
using (var httpClient = new HttpClient()) {
httpClient.BaseAddress = new Uri(this.URL);
httpClient.DefaultRequestHeaders.Accept.Clear();
httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
HttpResponseMessage response = await httpClient.GetAsync(String.Format("/locations/{0}/latest", memberId));
if (response.IsSuccessStatusCode) {
string json = await response.Content.ReadAsStringAsync();
locationRecord = JsonConvert.DeserializeObject<LocationRecord>(json);
}
}
return locationRecord;
}
}
}
```

Next we will modify the controller as shown in Fig. 6.25 to invoke the location client so that the most recent location for the member to the response can be appended.

Fig. 6.25 Code for src/BajiRainaBanuCorp.TeamService/Controllers/MembersController.cs

MembersController.cs - Notepad

File Edit Format View Help

```
[HttpGet]
[Route("/teams/{teamId}/[controller]/{memberId}")]
public async virtual Task<IActionResult>
GetMember(Guid teamID, Guid memberId)
{
Team team = repository.GetTeam(teamID);
if(team == null) {
return this.NotFound();
} else {
var q = team.Members.Where(m => m.ID == memberId);
if(q.Count() < 1) {
return this.NotFound();
} else {
Member member = (Member)q.First();
return this.Ok(new LocatedMember {
ID = member.ID,
FirstName = member.FirstName,
LastName = member.LastName,
LastLocation =
await this.locationClient.GetLatestForMember(member.ID)
});
}
}
}
```

Next we execute the docker images of teamservice by executing the command as shown in Fig. 6.26.

Fig. 6.26 Docker commands for executing team service

```
$ docker run -p 5000:5000 -e PORT=5000 \
  -e LOCATION__URL=http://localhost:5001 \
  dotnetcoreservices/teamservice:location
...
info: Startup[0]
      Using http://localhost:5001 for location service URL.
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:5000
Application started. Press Ctrl+C to shut down.
```

After the successful execution of team service let us execute the location service by executing the command as shown in Fig. 6.27:

Fig. 6.27 Docker commands for executing location service

```
$ docker run -p 5001:5001 -e PORT=5001 \
   dotnetcoreservices/locationservice:nodb
...
Status: Downloaded newer image for
dotnetcoreservices/locationservice:nodb
starting
Hosting environment: Production
Content root path: /pipeline/source/app/publish
Now listening on: http://0.0.0.0:5001
Application started. Press Ctrl+C to shut down.
```

Next we will execute a series of commands to check if everything is working properly. First we will create a new team by executing the curl command as shown in Fig. 6.28.

Fig. 6.28 Curl command for creating a new team

```
$ curl -H "Content-Type:application/json" -X POST -d \
 '{"id":"e52baa63-d511-417e-9e54-7aab04286281", \
   "name":"Team Zombie"}' http://localhost:5000/teams
```

Next we will add a member to that team by posting to the /teams/{id}/members resource by executing the curl command as shown in Fig. 6.29.

Fig. 6.29 Curl command for adding new team member

```
$ curl -H "Content-Type:application/json" -X POST -d \
 '{"id":"63e7acf8-8fae-42ce-9349-3c8593ac8292", \
   "firstName":"Al", \
   "lastName":"Foo"}' \
 http://localhost:5000/teams/e52baa63-d511-417e-9e54-
7aab04286281/members
```

Next query the team details to check the member details from the teams/{id}/members/{id} resource by executing the curl command as shown in Fig. 6.30.

Fig. 6.30 Curl command for querying the team details

```
$ curl http://localhost:5000/teams/e52baa63-d511-417e-9e54-7aab04286281

{"name":"Team Zombie",
  "id":"e52baa63-d511-417e-9e54-7aab04286281",
  "members":[{"id":"63e7acf8-8fae-42ce-9349-3c8593ac8292",
    "firstName":"Al","lastName":"Foo"}]}
```

Next we will add a location to the member's location history by executing the curl command as shown in Fig. 6.31.

Fig. 6.31 Curl command for adding a location to the member's location history

```
$ curl -H "Content-Type:application/json" -X POST -d \
 '{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f", \
   "latitude":12.0,"longitude":12.0,"altitude":10.0, \
   "timestamp":0, \

   "memberId":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}' \
 http://localhost:5001/locations/63e7acf8-8fae-42ce-9349-3c8593ac8292

{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f",
  "latitude":12.0,"longitude":12.0,
  "altitude":10.0,"timestamp":0,
  "memberID":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}
```

Finally query the member's details from the team service to see their location added to the response. We will query for the member details from the teams/{id}/members/{id} resource by executing the curl command as shown in Fig. 6.32.

Fig. 6.32 Curl command for querying the member details

```
$ curl http://localhost:5000/teams/e52baa63-d511-417e-9e54-7aab04286281
\
/members/63e7acf8-8fae-42ce-9349-3c8593ac8292

{
  "lastLocation":
    {"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f",
      "latitude":12.0,"longitude":12.0,
      "altitude":10.0,"timestamp":0,
      "memberID":"63e7acf8-8fae-42ce-9349-3c8593ac8292"},
  "id":"63e7acf8-8fae-42ce-9349-3c8593ac8292",
  "firstName":"Al",
  "lastName":"Foo"
}
```

6.10 Summary

1) This chapter gives an introduction on Microservices and their functionality and how to build Microservices with ASP.NET Core using the language C#. Also the concept of API First and how it is essential to allow multiple teams to have independent release cadences is also discussed in this chapter.

2) This chapter also gives an introduction on how to build two services – team service and location service for an imaginary company. We also saw some of the complexities of building ecosystems of Microservices and challenges involved in allowing one service to communicate with another service.

6.11 Review Questions

1) What is Microservices? What is Microservices Ecosystem?

2) Explain the API first development for the team service discussed in this chapter.

3) Explain the steps for creating a CI pipeline for the team service.

4) Explain how to build the location service?

5) Explain how to build and execute the docker images for team service and location service?

6.12 Bibliography, References and Further Reading

1) Building Microservices with ASP.NET Core by Kevin Hoffman

2) LittleAspNetCoreBook by Nate Barbettini

3) Microservices for the Enterprise by Kasun Indrasiri and Prabath Siriwardena

Unit 4

7

CREATING DATA SERVICE AND EVENT SOURCING AND CQRS.

UNIT STRUCTURE

7.0 Objectives

- In this chapter we will learn how to store our microservice data in Data Store using Postgres.
- We will learn how to configure data service and to choose between transient and persistent storage.
- At the end we will learn how to run sample service using local or docker image.
- Using Entity Framework to trace changes.
- We will also learn about Event Sourcing and CQRS with example.
- We will learn how to use RabbitMQ and Redis cache.

7.1 Introduction

- Data can be stored In-Memory by setting transient as True.
- To store data permanently in data store we need to set transient as False.
- Using separate data service allows us to keep other services independently deployable.
- Here data service created as a backing service.
- Event sourcing allows us to send notifications immediately to the Queue (RabbitMQ).
- We have team service with Teams and Members details.
- And location report service which is responsible to generate event when member location changes.
- Proximity detector service will find if another team member is in given threshold range (proximity) and generates proximity detected event.
- Redis cache can be used to store latest location found and can be fetched using Reality service.
- In CQRS: Command Query Responsibility Segregation pattern we try to create two separate microservices one for executing commands (add, update and delete) and another for only query (select or reading operations).

7.2 Choosing a Data Store

- The ecosystem is generally immature, so support for your favorite things may be lacking or missing entirely.

- One of things we might run into when trying to pick a data store that is compatible with EF Core is a lack of available providers.
- The following providers were available for EF Core:
- SQL Server , SQLite, Postgres, IBM databases, MySQL, SQL Server Lite
- In-memory provider for testing
- Oracle (coming soon)
- For databases that aren't inherently compatible with the Entity Framework relational model, like MongoDB, Neo4j, Cassandra, etc., you should be able to find client libraries available that will work with .NET Core.
- Since most of these databases expose simple RESTful APIs, you should still be able to use them even if you have to write your own client.

7.3 Building a Postgres Repository

- We're going upgrade our location service to work with Postgres.
- To do this we're going to create a new repository implementation that encapsulates the PostgreSQL client communication.
- The location repository exposes standard CRUD functions like Add, Update, Get, and Delete.
- In addition, this repository exposes methods to obtain the latest location entry for a member as well as the entire location history for a member.
- The purpose of the location service is solely to track location data, so you'll notice that there is no reference to team membership at all in this interface.
- Example file : ILocationRecordRepository.cs

```
using System;
using System.Collections.Generic;

namespace StatlerWaldorfCorp.LocationService.Models {

    public interface ILocationRecordRepository {
        LocationRecord Add(LocationRecord locationRecord);
        LocationRecord Update(LocationRecord locationRecord);
        LocationRecord Get(Guid memberId, Guid recordId);
        LocationRecord Delete(Guid memberId, Guid recordId);

        LocationRecord GetLatestForMember(Guid memberId);

        ICollection<LocationRecord> AllForMember(Guid memberId);
    }
}
```

7.3.1 Creating a Database Context

- This class will serve as a wrapper around the base DbContext class we get from Entity Framework Core.
- Since we're dealing with locations, we'll call our context class LocationDbContext.
- The pattern for using a database context is to create a class that inherits from it that is specific to your model.
- In our case, since we're dealing with locations, we'll create a LocationDbContext class.

```
using Microsoft.EntityFrameworkCore;
using StatlerWaldorfCorp.LocationService.Models;
using Npgsql.EntityFrameworkCore.PostgreSQL;

namespace StatlerWaldorfCorp.LocationService.Persistence
{
    public class LocationDbContext : DbContext
    {
      public LocationDbContext(
        DbContextOptions<LocationDbContext> options) :
           base(options)
      {
      }

      protected override void OnModelCreating(
        ModelBuilder modelBuilder)
      {
          base.OnModelCreating(modelBuilder);
          modelBuilder.HasPostgresExtension("uuid-ossp");
      }

      public DbSet<LocationRecord> LocationRecords {get; set;}
    }
}
```

- In our case, we're ensuring that our model has the uuid-ossp Postgres extension to support the member ID field.

7.3.2 Implementing the Location Record Repository Interface

- We can create a real implementation of the ILocationRecordRepository interface.

- This real implementation will take an instance of LocationDbContext as a constructor parameter.

```
using System;
using System.Linq;
using System.Collections.Generic;
using StatlerWaldorfCorp.LocationService.Models;

namespace StatlerWaldorfCorp.LocationService.Persistence
{
  public class LocationRecordRepository :
    ILocationRecordRepository
  {
      private LocationDbContext context;

      public LocationRecordRepository(LocationDbContext context)
      {
          this.context = context;
      }

public LocationRecordRepository(LocationDbContext context)
{
    this.context = context;
}

public LocationRecord Add(LocationRecord locationRecord)
{
    this.context.Add(locationRecord);
    this.context.SaveChanges();
    return locationRecord;
}

public LocationRecord Update(LocationRecord locationRecord)
{
    this.context.Entry(locationRecord).State =
         EntityState.Modified;
    this.context.SaveChanges();
    return locationRecord;
}

public LocationRecord Get(Guid memberId, Guid recordId)
{
    return this.context.LocationRecords
     .Single(lr => lr.MemberID == memberId &&
                   lr.ID == recordId);
}

public LocationRecord Delete(Guid memberId, Guid recordId)
{
    LocationRecord locationRecord =
      this.Get(memberId, recordId);
    this.context.Remove(locationRecord);
    this.context.SaveChanges();
    return locationRecord;
```

```
        }

        public LocationRecord GetLatestForMember(Guid memberId)
        {
            LocationRecord locationRecord =
              this.context.LocationRecords.
                Where(lr => lr.MemberID == memberId).
                OrderBy(lr => lr.Timestamp).
                Last();
            return locationRecord;
        }

        public ICollection<LocationRecord> AllForMember(Guid memberId)
        {
            return this.context.LocationRecords.
                Where(lr => lr.MemberID == memberId).
                OrderBy(lr => lr.Timestamp).
                ToList();
        }
    }
}
```

- Any time we make a change to the database, we call SaveChanges on the context.
- If we need to query, we use the LINQ expression syntax where we can combine Where and OrderBy to filter and sort the results.
- When we do an update, we need to flag the entity we're updating as a modified entry so that Entity Framework Core knows how to generate an appropriate SQL UPDATE statement for that record.
- If we don't modify this entry state, EF Core won't know anything has changed and so a call to SaveChanges will do nothing.
- The next big trick in this repository is injecting the Postgres-specific database context.
- To make this happen, we need to add this repository to the dependency injection system in the ConfigureServices method of our Startup class.

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddEntityFrameworkNpgsql()
        .AddDbContext<LocationDbContext>(options =>
               options.UseNpgsql(Configuration));
     services.AddScoped<ILocationRecordRepository,
        LocationRecordRepository>();
     services.AddMvc();
}
```

- First we want to use the AddEntityFrameworkNpgsql extension method exposed by the Postgres EF Core provider.
- Next, we add our location repository as a scoped service.
- When we use the AddScoped method, we're indicating that every new request made to our service gets a newly created instance of this repository.

7.3.3 Testing with the Entity Framework Core In-Memory Provider

- The InMemory provider is *not* a relational database.
- This means that you can save data using this provider that might normally violate a real database's referential integrity and foreign key constraints.
- We have already built a repository that works against collection objects, so the only added value this provider gives us is a little bit of additional code coverage to ensure that our database context is actually invoked.
- You should *not* assume that the InMemory provider is going to give you confidence that your database operations will behave as planned.

7.4 Databases are Backing Services

- We need to treat everything that our application needs to function as a bound resource: files, databases, services, messaging, etc.
- Every backing service our application needs should be configurable externally.
- As such, we need to be able to get our database connection string from someplace *other* than our code.
- The means by which an application gets its external configuration vary from
- Ex: using environment variables that can override defaults supplied by a configuration file.
- This *appsettings.json* file looks like the one here

```
{
    "transient": false,
    "postgres": {
        "cstr": "Host=localhost;Port=5432;Database=locationservice;
Username=integrator;Password=inteword"
    }
}
```

7.4.1 Configuring a Postgres Database Context

- The repository we built earlier requires some kind of database context in order to function.
- The database context is the core primitive of Entity Framework Core.
- To create a database context for the location model, we just need to create a class that inherits from DbContext.
- I've also included a DbContextFactory because that can sometimes make running the Entity Framework Core command-line tools simpler:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using StatlerWaldorfCorp.LocationService.Models;
using Npgsql.EntityFrameworkCore.PostgreSQL;

namespace StatlerWaldorfCorp.LocationService.Persistence
{
    public class LocationDbContext : DbContext
    {
        public LocationDbContext(
          DbContextOptions<LocationDbContext> options) :base(options)
        {
        }

        protected override void OnModelCreating(
            ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.HasPostgresExtension("uuid-ossp");
        }

        public DbSet<LocationRecord> LocationRecords {get; set;}
    }


        public class LocationDbContextFactory :
         IDbContextFactory<LocationDbContext>
        {
            public LocationDbContext
              Create(DbContextFactoryOptions options)
            {
                var optionsBuilder =
                  new DbContextOptionsBuilder<LocationDbContext>();
                var connectionString =
                  Startup.Configuration
                    .GetSection("postgres:cstr").Value;
                optionsBuilder.UseNpgsql(connectionString);

                return new LocationDbContext(optionsBuilder.Options);
            }
        }
    }
```

With a new database context, we need to make it available for dependency injection so that the location repository can utilize it:

```
public void ConfigureServices(IServiceCollection services)
{
    var transient = true;
    if (Configuration.GetSection("transient") != null) {
        transient = Boolean.Parse(Configuration
          .GetSection("transient").Value);
    }
    if (transient) {
        logger.LogInformation(
          "Using transient location record repository.");
        services.AddScoped<ILocationRecordRepository,
                           InMemoryLocationRecordRepository>();
    } else {
        var connectionString =
          Configuration.GetSection("postgres:cstr").Value;

        services.AddEntityFrameworkNpgsql()
                .AddDbContext<LocationDbContext>(options =>
                           options.UseNpgsql(connectionString));
        logger.LogInformation(
          "Using '{0}' for DB connection string.",
          connectionString);
        services.AddScoped<ILocationRecordRepository,
          LocationRecordRepository>();
    }

    services.AddMvc();
}
```

- With a context configured for DI, our service should be ready to run, test, and accept EF Core command-line parameters like the ones we need to execute migrations.

7.5 Integration Testing Real Repositories

- We want our automated build pipeline to spin up a fresh, empty instance of Postgres *every time* we run the build.
- Then we want integration tests to run against this fresh instance, including running our migrations to set up the schema in the database.
- We want this to work locally, on our teammates' workstations, and in the cloud, all automatically after every commit.

- If we just add the following lines to the top of our *wercker.yml* file, the Wercker CLI will spin up a connected Postgres Docker image and create a bunch of environment variables that provide the host IP, port, and credentials for the database

```
services:
    - id: postgres
      env:
        POSTGRES_PASSWORD: inteword
        POSTGRES_USER: integrator
        POSTGRES_DB: locationservice
```

- Now we can set up some build steps that prepare for and execute integration tests.

```
# integration tests
    - script:
        name: integration-migrate
        cwd: src/StatlerWaldorfCorp.LocationService
        code: |
           export TRANSIENT=false
           export POSTGRES__CSTR=
"Host=$POSTGRES_PORT_5432_TCP_ADDR"
           export POSTGRES__CSTR=
"$POSTGRES__CSTR;Username=integrator;Password=inteword;"
           export POSTGRES__CSTR=
"$POSTGRES__CSTR;Port=$POSTGRES_PORT_5432_TCP_PORT;
  Database=locationservice"
           dotnet ef database update
    - script:
        name: integration-restore
        cwd: test/StatlerWaldorfCorp.LocationService.Integration
        code: |
           dotnet restore
    - script:
        name: integration-build
        cwd: test/StatlerWaldorfCorp.LocationService.Integration
        code: |
          dotnet build
    - script:
        name: integration-test
        cwd: test/StatlerWaldorfCorp.LocationService.Integration
        code: |
          dotnet test
```

- The following is the list of commands being executed by the integration suite:

- **dotnet ef database update**
- Ensures that the schema in the database matches what our EF Core model expects.
- This will actually instantiate the Startup class, call ConfigureServices, and attempt to pluck out the LocationDbContext class and then execute the migrations stored in the project.
- **dotnet restore**
- Verifies and collects dependencies for our integration test project.
- **dotnet build**
- Compiles our integration test project.
- **dotnet test**
- Runs the detected tests in our integration test project.

7.6 Exercise the Data Service.

- The first thing we're going to need to do is spin up a running instance of Postgres.
- If you were paying attention to the *wercker.yml* file for the location service that sets up the integration tests, then you might be able to guess at the docker run command to start Postgres with our preferred parameters:

```
$ docker run -p 5432:5432 --name some-postgres \
  -e POSTGRES_PASSWORD=inteword -e POSTGRES_USER=integrator \
  -e POSTGRES_DB=locationservice -d postgres
```

- This starts the Postgres Docker image with the name some-postgres (this
- will be important shortly). To verify that we can connect to Postgres, we can
- run the following Docker command to launch psql:

```
$ docker run -it --rm --link some-postgres:postgres postgres \
  psql -h postgres -U integrator -d locationservice
Password for user integrator:
psql (9.6.2)
Type "help" for help.

locationservice=# select 1;
 ?column?
----------
        1
(1 row)
```

- With the database up and running, we need a schema.
- The tables in which we expect to store the migration metadata and our location records don't yet exist.
- To put them in the database, we just need to run an EF Core command from the location service's project directory.
- Note that we're also setting environment variables that we'll need soon:

```
$ export TRANSIENT=false
$ export POSTGRES__CSTR="Host=localhost;Username=integrator; \
  Password=inteword;Database=locationservice;Port=5432"
$ dotnet ef database update

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.25
info: Startup[0]
      Using 'Host=localhost;Username=integrator;
Password=inteword;Port=5432;Database=locationservice' for DB
 connection string.
Executed DbCommand (13ms) [Parameters=[], CommandType='Text',
 CommandTimeout='30']
SELECT EXISTS (SELECT 1 FROM pg_catalog.pg_class c
JOIN pg_catalog.pg_namespace n ON n.oid=c.relnamespace WHERE
 c.relname='__EFMigrationsHistory');
Executed DbCommand (56ms) [Parameters=[], CommandType='Text',
 CommandTimeout='30']
CREATE TABLE "__EFMigrationsHistory" (
    "MigrationId" varchar(150) NOT NULL,
    "ProductVersion" varchar(32) NOT NULL,
    CONSTRAINT "PK___EFMigrationsHistory" PRIMARY KEY
("MigrationId")
);
```

```
Executed DbCommand (0ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
SELECT EXISTS (SELECT 1 FROM pg_catalog.pg_class c JOIN
 pg_catalog.pg_namespace n ON n.oid=c.relnamespace WHERE
 c.relname='__EFMigrationsHistory');
Executed DbCommand (2ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
SELECT "MigrationId", "ProductVersion"
FROM "__EFMigrationsHistory"
ORDER BY "MigrationId";
Applying migration '20160917140258_Initial'.
Executed DbCommand (19ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
Executed DbCommand (18ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
CREATE TABLE "LocationRecords" (
    "ID" uuid NOT NULL,
    "Altitude" float4 NOT NULL,
    "Latitude" float4 NOT NULL,
    "Longitude" float4 NOT NULL,
    "MemberID" uuid NOT NULL,
    "Timestamp" int8 NOT NULL,



    CONSTRAINT "PK_LocationRecords" PRIMARY KEY ("ID")
);
Executed DbCommand (0ms) [Parameters=[], CommandType='Text',
 CommandTimeout='30']
INSERT INTO "__EFMigrationsHistory" ("MigrationId",
"ProductVersion")
VALUES ('20160917140258_Initial', '1.1.1');
Done.
```

- At this point Postgres is running with a valid schema and it's ready to start accepting commands from the location service.
- Here's where it gets a *little* tricky. If we're going to run the location service from inside a Docker image, then referring to the Postgres server's host as localhost won't work — because that's the host *inside* the Docker image.
- What we need is for the location service to reach *out* of its container and then *into* the Postgres container.

- We can do this with a container link that creates a virtual hostname (we'll call it postgres), but we'll need to change our environment variable before launching the Docker image:

```
$ export POSTGRES__CSTR="Host=postgres;Username=integrator; \
Password=inteword;Database=locationservice;Port=5432"
$ docker run -p 5000:5000 --link some-postgres:postgres \
  -e TRANSIENT=false -e PORT=5000 \
  -e POSTGRES__CSTR dotnetcoreservices/locationservice:latest
```

- Now that we've linked the service's container to the Postgres container via the postgres hostname, the location service should have no trouble connecting to the database.
- To see this all in action, let's submit a location record (as usual, take the line feeds out of this command when you type it):

```
$ curl -H "Content-Type:application/json" -X POST -d \
  '{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f","latitude":12.0, \
    "longitude":10.0,"altitude":5.0,"timestamp":0, \
    "memberId":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}' \
  http://localhost:5000/locations/63e7acf8-8fae-42ce-9349-3c8593ac8292
```

- Let's ask the service for this fictitious member's location history:

```
$ curl http://localhost:5000/locations/63e7acf8-8fae-42ce-9349-
3c8593ac8292

[{"id":"64c3e69f-1580-4b2f-a9ff-2c5f3b8f0e1f",
  "latitude":12.0,"longitude":10.0,"altitude":5.0,
  "timestamp":0,"memberID":"63e7acf8-8fae-42ce-9349-3c8593ac8292"}]
```

The corresponding Entity Framework trace looks like this:

```
info: Microsoft.EntityFrameworkCore.Storage.
IRelationalCommandBuilderFactory[1]
      Executed DbCommand (23ms) [Parameters=[@__memberId_0='?'],
 CommandType='Text', CommandTimeout='30']
      SELECT "lr"."ID", "lr"."Altitude", "lr"."Latitude",
 "lr"."Longitude", "lr"."MemberID", "lr"."Timestamp"
      FROM "LocationRecords" AS "lr"
      WHERE "lr"."MemberID" = @__memberId_0
      ORDER BY "lr"."Timestamp"
```

- Use docker ps and docker kill to locate the Docker process for the location service and kill it.
- Restart it using the exact same command you used before.

7.7 Introducing Event Sourcing

- We're just making smaller monoliths, not taking full advantage of the cloud, or of truly robust distributed computing design patterns. we'll use an analogy: reality itself.

7.7.1 Reality Is Event Sourced

- Our brains are essentially event-sourced systems.
- We receive stimuli in the form of the five senses, and our brains are then responsible for properly sequencing each stimulus (an event).
- Every few hundred milliseconds or so, they perform some calculations against this never- ending stream of stimuli.
- The result of these calculations is what we call reality.
- Our minds process the incoming event stream and then compute state.
- This state is what we perceive as our reality; the world around us.
- When we watch someone dancing to music, we're receiving audio and visual events, ensuring they're in the proper order Event-sourced applications operate in a similar manner.
- They consume streams of incoming events, perform functions against the inbound streams, and compute results or state in response.

7.7.2 Event Sourcing Defined

- Number of requirements for an event-sourced system.

1. Ordered

- Event streams are ordered. Performing calculations against the same set of events but in a different sequence will produce different output.

2. Idempotent

- Any function that operates on an event stream must always return the exact same result for identical ordered event streams.

3. Isolated

- Any function that produces a result based on an event stream cannot make use of external information. All data required for calculations must be present in the events.

4. Past tense

- Events take place in the past.
- This should be reflected in your variable names, structure names, and architecture.
- Event processors run calculations against a chronologically ordered sequence of events that have already happened.
- Put mathematically, a function that operates on a stream of events will always produce the same state and output set of new events.
- For example:
- $f(event^1, event^2, ...) = state^1 + \{ \text{output event set} \}$
- Let's take a financial transaction processing system as a sample. We could have an inbound stream of transactions, the processing of which results in state changes such as modifications of account balances, credit limits, and so on.

7.7.3 Learning to Love Eventual Consistency

- In an event-sourced system, you don't get to perform simple CRUD (Create, Read, Update, Delete) operations in a synchronous fashion against one or more services.
- There is no immediate feedback from the system of record that gives you the concrete state of how things exist in a consistent manner.
- Instead, things in this new world are eventually consistent.
- Your banking system is eventually consistent: eventually the transaction where you just purchased that shiny new computer will show up in your bank account.
- Other eventually consistent applications with which we all interact daily are social networking apps.
- You've probably seen the scenario where a comment or post you make from one device takes a few minutes to show up in a friend's browser or device.
- This is because the application architects have decided on a trade-off: by giving up the immediate consistency of synchronous operation in favor of a tolerable delay in feedback.

7.8 The CQRS Pattern

- The separation of command inputs from queries in our system, otherwise known as the Command Query Responsibility Segregation pattern.
- Commands are responsible for submitting inputs into our system, which will likely result in the creation of events distributed to one or more streams.
- The response from submitting a command is not the newly altered (consistent) state, it is merely an acknowledgment of whether or not the command was successfully ingested by the system.
- Eventually, the state of the system will be altered to reflect the processing of this one command.
- The size of this time lapse depends entirely on the business process being performed and the criticality of the propagation of the data change.
- Traditional backend monolithic applications involve hitting a query endpoint with some parameters.
- Those parameters are then used to perform some amount of lengthy processing and querying, returning calculated results.
- In the world of massive scale, volume, and throughput we simply can't afford to tie up the resources of our microservices by making computationally expensive queries.
- We want the queries to be as dumb as possible.
- Knowing the usage pattern of the majority of our customers gives us the ability to take advantage of Event Sourcing and build a proper CQRS implementation.
- Our event processor can recompute cached meter aggregates every time it receives a new event.
- With this in place, we'll have the results portal users are expecting already sitting in a database or cache when the query happens.
- The event store (persistent storage of all meter events received since the system started) is still available if we need more complex calculations or auditing, but the eventually consistent state (reality) is made available for immediate, super-fast query to all consumers.

7.9 Event Sourcing and CQRS in Action—Team Proximity Sample (Event Sourcing and CQRS combined)

- While having near-real-time location data on all of the people using our application is a great feature on its own, the real power comes from what we can do by processing incoming events.
- In our case, we want to detect when two team members are close to each other.
- We will be detecting when member locations occur within some small distance of each other.
- The system will then support reacting to these proximity detections.
- For example, we might want to send push notifications to the mobile devices of the nearby team members to alert them to the possibility for catching up in person.
- we'll be splitting up the responsibilities of the system among four components, as follows:
- 1.The location reporter service (Command)
- 2.The event processor (Event Sourcing)
- 3.The reality service (Query)
- 4.The proximity monitor (Event Sourcing)

7.9.1 The Location Reporter Service

- In a CQRS system, the inputs and outputs of the system are decoupled entirely.
- For our sample, the inputs take the form of commands sent to the location reporter service.
- The client applications (mobile, web, IoT, etc.) in our system need to submit new location data on members on a regular basis.
- They will do so by sending updates to the location reporter.

- API for this service:

Resource	Method	Description
/api/members/{memberId}/locationreports	POST	Submits a new location report

- When we get a new location report, we'll perform the following tasks:
- 1. Validate the report object.
- 2. Convert the command into an event.
- 3. Emit the event on a message queue
- The command processor needs to create an event with an appropriate timestamp, and it's also going to need to fetch the team membership (which is a volatile quantity, subject to change at any time) to place that information on the event.
- This has the desirable effect of allowing our system to detect nearby team members only if they were on the same team at the time the events occurred.
- **Creating the location reports controller:**

```
using System;
using Microsoft.AspNetCore.Mvc;
using StatlerWaldorfCorp.LocationReporter.Events;
using StatlerWaldorfCorp.LocationReporter.Models;
using StatlerWaldorfCorp.LocationReporter.Services;

namespace StatlerWaldorfCorp.LocationReporter.Controllers
{
[Route("/api/members/{memberId}/locationreports")]
public class LocationReportsController : Controller
{
    private ICommandEventConverter converter;
```

```
private IEventEmitter eventEmitter;
private ITeamServiceClient teamServiceClient;


public LocationReportsController(
    ICommandEventConverter converter,
    IEventEmitter eventEmitter,
    ITeamServiceClient teamServiceClient) {

    this.converter = converter;
    this.eventEmitter = eventEmitter;
    this.teamServiceClient = teamServiceClient;
}
    [HttpPost]
    public ActionResult PostLocationReport(Guid memberId,
      [FromBody]LocationReport locationReport)
    {
      MemberLocationRecordedEvent locationRecordedEvent =
        converter.CommandToEvent(locationReport);
      locationRecordedEvent.TeamID =
        teamServiceClient.GetTeamForMember(
           locationReport.MemberID);
      eventEmitter.EmitLocationRecordedEvent(
         locationRecordedEvent);

      return this.Created(
        $"/api/members/{memberId}/locationreports/
        {locationReport.ReportID}",
        locationReport);
    }
}
}
```

- The command converter creates a basic event from an input command while the team service client allows us to fetch the ID of the team to which the member belongs (our system only allows people to belong to one team at a time).
- Finally, the event emitter is responsible for sending the event to the right place.

- **Building an AMQP event emitter**
- The location reporter service is actually pretty small, and other than the controller, the most interesting stuff is in the event emitter.
- Our sample emits events to an Advanced Message Queuing Protocol (AMQP) queue supported by RabbitMQ.

```
using System;
using System.Linq;
using System.Text;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using RabbitMQ.Client;
using StatlerWaldorfCorp.LocationReporter.Models;

namespace StatlerWaldorfCorp.LocationReporter.Events
{
    public class AMQPEventEmitter : IEventEmitter
    {
        private readonly ILogger logger;
        private AMQPOptions rabbitOptions;
        private ConnectionFactory connectionFactory;


public AMQPEventEmitter(ILogger<AMQPEventEmitter> logger,
    IOptions<AMQPOptions> amqpOptions)
{
    this.logger = logger;
    this.rabbitOptions = amqpOptions.Value;

    connectionFactory = new ConnectionFactory();

    connectionFactory.UserName = rabbitOptions.Username;
    connectionFactory.Password = rabbitOptions.Password;
    connectionFactory.VirtualHost =
      rabbitOptions.VirtualHost;
    connectionFactory.HostName = rabbitOptions.HostName;
    connectionFactory.Uri = rabbitOptions.Uri;

    logger.LogInformation(
      "AMQP Event Emitter configured with URI {0}",
       rabbitOptions.Uri);
```

```
    }
public const string QUEUE_LOCATIONRECORDED =
  "memberlocationrecorded";

public void EmitLocationRecordedEvent(
  MemberLocationRecordedEvent locationRecordedEvent)
{
    using (IConnection conn = connectionFactory.
       CreateConnection()) {
        using (IModel channel = conn.CreateModel()) {


                    channel.QueueDeclare(
                        queue: QUEUE_LOCATIONRECORDED,
                        durable: false,
                        exclusive: false,
                        autoDelete: false,
                        arguments: null
                    );
                    string jsonPayload =
                      locationRecordedEvent.toJson();
                    var body =
                       Encoding.UTF8.GetBytes(jsonPayload);
                    channel.BasicPublish(
                        exchange: "",
                        routingKey: QUEUE_LOCATIONRECORDED,
                        basicProperties: null,
                        body: body
                    );
                }
            }
        }
    }
}
```

- **Configuring and starting the service**
- The AMQP event emitter class gets the information needed to configure a RabbitMQ connection factory from an options instance.
- You can see how these options are configured by looking at the location reporter's Startup class.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System;
using Microsoft.Extensions.Logging;
using System.Linq;
using StatlerWaldorfCorp.LocationReporter.Models;
using StatlerWaldorfCorp.LocationReporter.Events;
using StatlerWaldorfCorp.LocationReporter.Services;

 namespace StatlerWaldorfCorp.LocationReporter
 {
     public class Startup
     {
         public Startup(IHostingEnvironment env,
           ILoggerFactory loggerFactory)
         {
             loggerFactory.AddConsole();
             loggerFactory.AddDebug();

   public void ConfigureServices(IServiceCollection services)
   {
       services.AddMvc();
       services.AddOptions();

       services
         .Configure<AMQPOptions>(
            Configuration.GetSection("amqp"));
       services
         .Configure<TeamServiceOptions>(
            Configuration.GetSection("teamservice"));

       services.AddSingleton(typeof(IEventEmitter),
         typeof(AMQPEventEmitter));
       services.AddSingleton(typeof(ICommandEventConverter),
         typeof(CommandEventConverter));
       services.AddSingleton(typeof(ITeamServiceClient),
         typeof(HttpTeamServiceClient));
   }

           public void Configure(IApplicationBuilder app,
                   IHostingEnvironment env,
                   ILoggerFactory loggerFactory,
                   ITeamServiceClient teamServiceClient,
                   IEventEmitter eventEmitter)
           {
               // Asked for instances of singletons during startup
               // to force initialization early.

               app.UseMvc();
           }
       }
   }
```

- It's important to remember that the order of precedence is defined by the order in which you add configuration sources, so make sure that you always add your local/default JSON settings first so they can be overridden. Here's what our appsettings.json file looks like:

```
{
  "amqp": {
    "username": "guest",
    "password": "guest",
    "hostname": "localhost",
    "uri": "amqp://localhost:5672/",
    "virtualhost": "/"
  },
  "teamservice": {
    "url": "http://localhost:5001"
  }
}
```

- **Consuming the team service**
- Before we get to running the location reporter, let's take a look at the HTTP implementation of the ITeamServiceClient

```
using System;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;
using StatlerWaldorfCorp.LocationReporter.Models;

namespace StatlerWaldorfCorp.LocationReporter.Services
{
    public class HttpTeamServiceClient : ITeamServiceClient
    {
        private readonly ILogger logger;

        private HttpClient httpClient;

        public HttpTeamServiceClient(
            IOptions<TeamServiceOptions> serviceOptions,
            ILogger<HttpTeamServiceClient> logger)
        {
            this.logger = logger;

            var url = serviceOptions.Value.Url;
```

```
        logger.LogInformation(
          "Team Service HTTP client using URL {0}",
          url);

        httpClient = new HttpClient();
        httpClient.BaseAddress = new Uri(url);
    }
    public Guid GetTeamForMember(Guid memberId)
    {
        httpClient.DefaultRequestHeaders.Accept.Clear();
        httpClient.DefaultRequestHeaders.Accept.Add(
          new MediaTypeWithQualityHeaderValue(
          "application/json"));

        HttpResponseMessage response =
          httpClient.GetAsync(
          String.Format("/members/{0}/team",
           memberId)).Result;

         TeamIDResponse teamIdResponse;
         if (response.IsSuccessStatusCode) {
             string json = response.Content
               .ReadAsStringAsync().Result;
             teamIdResponse =
               JsonConvert.DeserializeObject<TeamIDResponse>(
                  json);
             return teamIdResponse.TeamID;
         }
         else {
             return Guid.Empty;
         }
      }
    }

    public class TeamIDResponse
    {
        public Guid TeamID { get; set; }
    }
}
```

- Example we're using the .Result property to force a thread to block while we wait for a reply from the asynchronous method.
- To see the location reporter in action, we first need to set up a local copy of RabbitMQ.
- If you're on a Mac, it should be easy enough to either install RabbitMQ or just start up a Docker image running Rabbit with the management console plug-in enabled (make sure to map both the management console port and the regular port).
- On Windows, it's probably easiest to just install RabbitMQ locally.
- **Running the location reporter service**
- With that running, and our defaults set up to point to a local Rabbit instance, we can fire up the location reporter service as follows (make sure you're in the src/StatlerWaldorfCorp.LocationReporter subdirectory):

```
$ dotnet restore
...
$ dotnet build
...
$ dotnet run --server.urls=http://0.0.0.0:9090
...
```

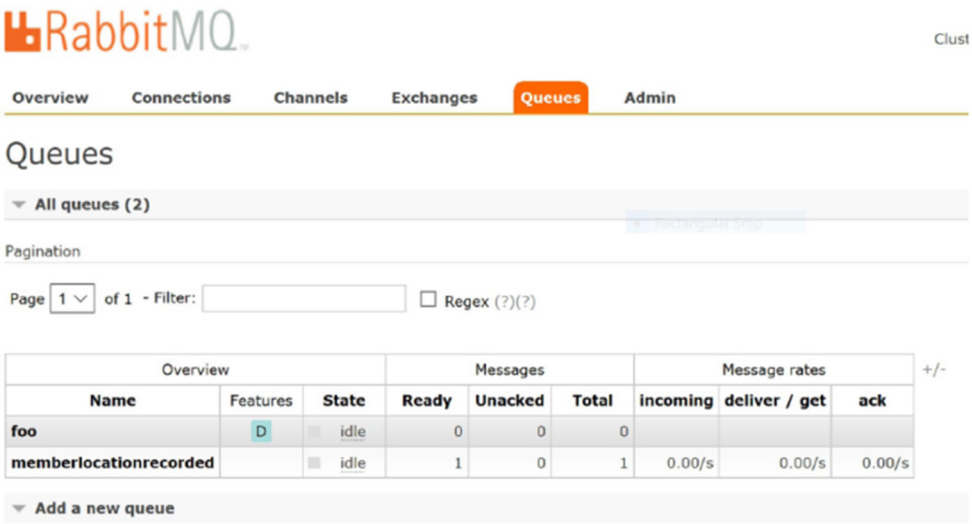
- Depending on your setup, you might not need to change the default port.
- With the service running, we just need to submit a request to the service.
- One of the easiest ways to do that is to install the Postman plug-in for Chrome, or we can use curl to submit a JSON payload like this one:

```
$ curl -X POST -d \
 '{"reportID": "...", \
   "origin": "...", "latitude": 10, "longitude": 20, \
   "memberID": "..."}' \
 http://...1e2 \
/locationreports
```

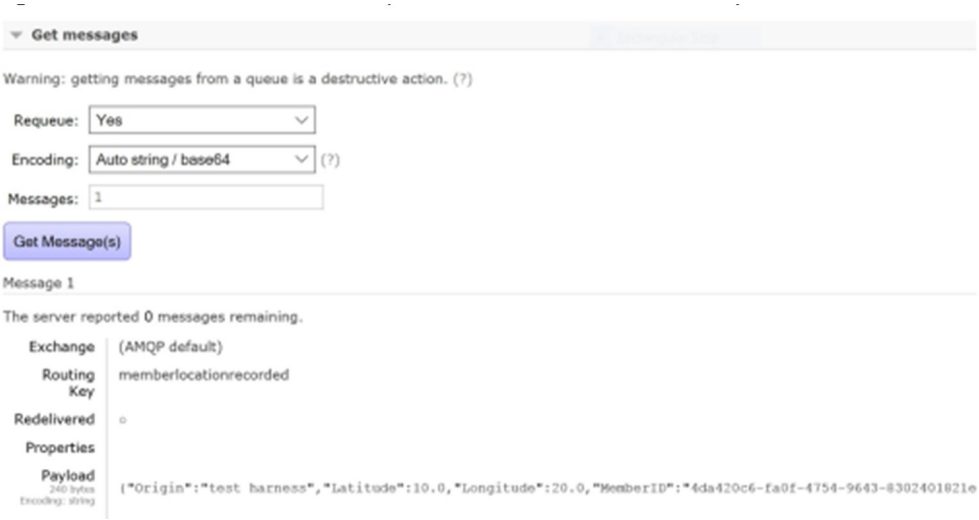
- When we submit this, we should get an HTTP 201 reply from the service, with the Location header set to something that looks:

```
like /api/members/4da420c6-fa0f-4754-9643-
8302401821e2/locationreports/f74be394-0d03-4a2f-bb55-
```

- If everything else is working properly, we should be able to use our RabbitMQ management console to see that there's a new message sitting in the memberlocationrecorded queue.



- And if we use this same management console to examine the contents of the message, we should see that it is a faithful JSON conversion of the event we created, including the augmentations of the timestamp and the team membership of the member.



7.9.2 The Event Processor

- The event processor is the part of the system that is as close to a pure function as we can get.
- It is responsible for consuming events from the stream and taking the appropriate actions.

- These actions could include emitting new events on new event streams or pushing state changes to the reality service While there are many important pieces to the event processor, the core of it is the ability to detect nearby teammates.
- To perform that detection, we need to know how to compute the distance between their GPS coordinates.
- In order to keep the code clean and testable, we want to separate the responsibilities of event processing into the following:
- Subscribing to a queue and obtaining new messages from the event stream.
- Writing messages to the event store.
- Processing the event stream (detecting proximity).
- Emitting messages to a queue as a result of stream processing.
- Submitting state changes to the reality server/cache as a result of stream processing.

```
[Fact]
public void ProducesAccurateDistanceMeasurements()
{
  GpsUtility gpsUtility = new GpsUtility();

  GpsCoordinate losAngeles = new GpsCoordinate() {
      Latitude = 34.0522222,
      Longitude = -118.2427778
  };

  GpsCoordinate newYorkCity = new GpsCoordinate() {
      Latitude = 40.7141667,
      Longitude = -74.0063889
  };

  double distance =
      gpsUtility.DistanceBetweenPoints(losAngeles, newYorkCity);
  Assert.Equal(3933, Math.Round(distance)); // 3,933 km
  Assert.Equal(0,
      gpsUtility.DistanceBetweenPoints(losAngeles, losAngeles));

}
```

- To detect proximity events, we write a proximity detector that makes use of the GPS utility class. It takes as input the event pulled from the stream, a list of teammates and their locations, and a radius threshold.

```
using System.Collections.Generic;
using StatlerWaldorfCorp.EventProcessor.Location;
using System.Linq;
using System;

namespace StatlerWaldorfCorp.EventProcessor.Events
{
    public class ProximityDetector
    {
      public ICollection<ProximityDetectedEvent>
          DetectProximityEvents(
             MemberLocationRecordedEvent memberLocationEvent,
             ICollection<MemberLocation> memberLocations,
             double distanceThreshold)
{
    GpsUtility gpsUtility = new GpsUtility();
    GpsCoordinate sourceCoordinate = new GpsCoordinate() {
        Latitude = memberLocationEvent.Latitude,
        Longitude = memberLocationEvent.Longitude
    };

    return memberLocations.Where(
      ml => ml.MemberID != memberLocationEvent.MemberID &&

            gpsUtility.DistanceBetweenPoints(
               sourceCoordinate, ml.Location) <
               distanceThreshold)
      .Select( ml => {
        return new ProximityDetectedEvent() {
          SourceMemberID = memberLocationEvent.MemberID,
          TargetMemberID = ml.MemberID,
          DetectionTime = DateTime.UtcNow.Ticks,
          SourceMemberLocation = sourceCoordinate,
          TargetMemberLocation = ml.Location,
          MemberDistance =
            gpsUtility.DistanceBetweenPoints(
               sourceCoordinate, ml.Location)
          };

               }).ToList();
          }
     }
}
```

- We can then take the results of this method and use them to create the appropriate side effects, including the optional dispatch of a ProximityDetectedEvent and the writing of an event to the event store.

```
using System;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using StatlerWaldorfCorp.EventProcessor.Location;
using StatlerWaldorfCorp.EventProcessor.Queues;

namespace StatlerWaldorfCorp.EventProcessor.Events
{
public class MemberLocationEventProcessor : IEventProcessor
{
    private ILogger logger;
    private IEventSubscriber subscriber;
    private IEventEmitter eventEmitter;
    private ProximityDetector proximityDetector;
    private ILocationCache locationCache;

 public MemberLocationEventProcessor(
     ILogger<MemberLocationEventProcessor> logger,
     IEventSubscriber eventSubscriber,
     IEventEmitter eventEmitter,
     ILocationCache locationCache)
 {
     this.logger = logger;
     this.subscriber = eventSubscriber;
     this.eventEmitter = eventEmitter;
     this.proximityDetector = new ProximityDetector();
     this.locationCache = locationCache;

     this.subscriber.
       MemberLocationRecordedEventReceived += (mlre) => {
         var memberLocations =
            locationCache.GetMemberLocations(mlre.TeamID);
         ICollection<ProximityDetectedEvent> proximityEvents =
             proximityDetector.DetectProximityEvents(mlre,

                 memberLocations, 30.0f);
           foreach (var proximityEvent in proximityEvents) {
             eventEmitter.
               EmitProximityDetectedEvent(proximityEvent);
           }

           locationCache.Put(mlre.TeamID,
             new MemberLocation {
               MemberID = mlre.MemberID,
               Location = new GpsCoordinate {
                 Latitude = mlre.Latitude,
                 Longitude = mlre.Longitude
               }
            });
       };
  }
```

```
    public void Start()
    {
        this.subscriber.Subscribe();
    }

    public void Stop()
    {
        this.subscriber.Unsubscribe();
    }
}
}
```

- The dependencies of this class are not only evident, but made mandatory through the use of the constructor parameters. They are:
- An instance of a logger appropriate for this class.
- An event subscriber (responsible for telling the processor when new MemberLocationRecordedEvents arrive).
- An event emitter, allowing the processor to emit ProximityDetectedEvents.
- A location cache, allowing us to quickly store and retrieve current locations of team members as discovered by the event processor.
- Depending on how you design your "reality" service, this cache can be shared by the reality service or a duplication of it.
- The only other responsibility of the event processing service is that it should store every event it receives in the event store.
- **The Redis location cache**
- The location cache interface has the following methods defined on it:
- GetMemberLocations(Guid teamId)
- Put(Guid teamId, MemberLocation location)
- Redis is also quite a bit more than just a cache.
- We're creating a Redis hash for each of the teams in our service.
- The JSON payload for a serialized member location is then added as a field (keyed on member ID) to this hash.
- This makes it easy to update multiple member locations simultaneously without creating a data-overwrite situation and makes it easy to query the list of locations for any given team, since the team is a hash.

7.9.3 The Reality Service

- Reality is subjective. The reality service is responsible for maintaining the location of each team member, but that location will only be the most recently received location from some application.
- We will never know exactly where someone is; we can only tell where they were when they last submitted a command that produced a successfully processed event.
- Again, this reinforces the notion that reality is really a function of stimuli received in the past.
- Let's take a look at the API we want to expose from the reality service.

Resource	Method	Description
`/api/reality/members`	GET	Retrieves the last known location of all members that are known to the reality service
`/api/reality/members/{memberId}`	GET	Retrieves the last known location of a single member
`/api/reality/members/{memberId}`	PUT	Sets the last known location of a member

- There are two important things to remember about a reality service like this:
- Reality is not the event store.
- Reality is merely a representation of the state you expect your consumers to need, a prebuilt set of data designed to support the query operations in a CQRS pattern.
- Reality is disposable.
- The reality cache that supports the query operations of the system is disposable.
- We should be able to destroy all instances of reality and reconstitute them simply by running our event processing algorithm against either an entire event stream, or the events occurring since the last snapshot.
- The code for the reality service is made up of:
- Basic microservice scaffolding (middleware, routing, configuration, bootstrapped web server)
- Reliance upon dependency injection to provide configuration options and implementation instances

- A class that talks to the Redis cache to query the current locations
- A consumer of the team service to query the list of teams

7.9.4 The Proximity Monitor

- The output of the event processor is a stream of proximity detected events.
- In a real-world, production system, we would have some kind of application or service sitting on the end of this stream.
- It would await the arrival of these events and then notify appropriate downstream components that the events have occurred.
- This could be a notification to a website to let a single-page app update its UI, or it could be a notification that gets sent to the mobile devices of both the source and target team members that are part of the event.
- The code for a proximity monitor would include:
- Basic microservice scaffolding (this should be old hat to you by now)
- A queue consumer subscribed to the arrival of ProximityDetectedEvent messages
- Consumption of some third-party or cloud provider to deal with push notifications

7.10 Running the Samples

- The following are the prerequisites for running the samples in this chapter:
- **A RabbitMQ server**
- You can install this locally on your machine, you can run a copy of the Docker image available on docker hub or you can point to a cloud-hosted RabbitMQ server.
- **A Redis server:**
- As with Rabbit, you can install this locally, run the Docker image, or point to a cloud-hosted Redis server. The appsettings.json files for the services are checked into GitHub such that the default operating mode is to assume the prerequisites are running locally either through direct install or through ports exposed and mapped from running Docker images.
- **Starting the Services**
- Once you've got your prerequisites up and running, check out the code for the services eslocationreporter and es-eventprocessor from GitHub.

- You'll also need to grab a copy of teamservice.
- Make sure you grab the master branch since you just want an in-memory repository for testing (the location branch requires a Postgres database).
- As per usual procedure, make sure you do a dotnet restore and a dotnet build on the main service application for each of them from inside their respective src/<project> directories.
- To start the team service, issue the following command in a terminal from the src/StatlerWaldorfCorp.TeamService directory:
- **$ dotnet run --server.urls=http://0.0.0.0:5001**
- To start the location reporter, issue the following command at your terminal from the src/StatlerWaldorfCorp.LocationReporter directory:
- **$ dotnet run --server.urls=http://0.0.0.0:5002**
- Now start the event processor (from the src/StatlerWaldorfCorp.EventProcessor directory):
- **$ dotnet run --server.urls=http://0.0.0.0:5003**
- The event processor has a number of dependencies, and you'll see a bunch of diagnostic information during startup that lets you know where it is attempting to find those dependencies.

Service	Docker image	Port
RabbitMQ	`rabbitmq:3.6.6`	5672
Redis Cache	`redis:3.2.6`	6379
Team service	`dotnetcoreservices/teamservice`	5001
Location reporter	`dotnetcoreservices/locationreporter`	5002
Event processor	`dotnetcoreservices/es-eventprocessor`	5003
Reality service (optional)	`dotnetcoreservices/es-reality`	5004

- **Submitting Sample Data**
- Use the following steps to exercise the entire Event Sourcing/CQRS system from end to end:

  1. Issue a POST to http://localhost:5001/teams to create a new team.
  2. Issue a POST to http://localhost:5001/teams/<new guid>/members to add a member to the team. Make sure you keep the GUID for the new member handy.

3. Issue a POST to http://localhost:5002/api/members/<memberguid>/locationreports.

   A location report requires the following fields: ReportID, Latitude, Longitude, Origin, ReportID, and MemberID.

4. Watch the location report being converted to a MemberLocationReportedEvent and placed on the appropriate queue (the default is memberlocationrecorded). If you need some reference coordinates for latitude and longitude, you can find several of them in the GpsUtilityTest class in the event processor unit test project.

5. Repeat step 3 a few times for locations that are far away from each other that will not trigger a proximity detected event.

- 6. Repeat step 2 to create a new member that belongs to the same team as your first test member.

7. Now repeat step 3 for this second team member at a location within a few kilometers of the most recently supplied location for the first team member.

8. You should now be able to see a new message in the proximitydetected queue (you can use the RabbitMQ management plug-in to view the queues without having to write code).

9. Either query the Redis cache directly or talk to the reality service to see the most up-to- date locations for members.

7.11 Summary

- We learned how to configure Postgres as data service in our microservice.
- The database context is the core primitive of Entity Framework Core.
- When transient is false we add scoped instance of In-Memory location record and if it is true we add Location Record in data service.
- We can use environment variable to pass values for variables such as transient, host name, user name , password , database name and connection string details of postgres.
- We saw how to perform Event sourcing using RabbitMQ and Redis cache.
- RabbitMQ stores messages in queues.
- Reality services reads data from Redis cache to find latest location of member.

7.12 Reference for further reading

- Building Microservices with ASP.NET Core by Kevin Hoffman (chapters 5 and 6)
- https://microservices.io/patterns/data/database-per-service.html

Unit 4

8

BUILDING AN ASP.NET CORE WEB APPLICATION AND SERVICE DISCOVERY UNIT STRUCTURE

Unit Structure:

8.0 Objectives

- First we will learn to write ASP.Net core wen application.
- We will learn MVC – Model, View and Controller.
- How to use Javascript and Ajax for invoking our application.

- Later in this chapter we will learn how to register service so that it can be discovered dynamically by other services.
- As example we study Netflix Eureka.
- We will learn example of Catalog service and Inventory service.
- Inventory service is registered and discovered or consumed by Catalog service.

8.1 Introduction

- We can create web application using ASP.Net core that is in console (without using visual studio).
- It creates default template for us, which we can modify (MVC).
- For service discovery we use following to variables:
- "shouldRegisterWithEureka" if set to true the service can be discovered.
- "shouldFetchRegistry" if set to true it will consume service.
- We use Steeltoe project for this.
- We can run two services on two port numbers to check the working.
- The full sample code for the catalog service and the inventory service can be downloaded from Github.

8.2 ASP.NET Core Basics

- To create MVC application type :
- dotnet new mvc --auth none
- We can simply indicate that we want to use the Web SDK (Microsoft.NET. Sdk.Web) at the opening of the project file, and that saves us from having to explicitly declare certain dependencies:
-

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
```

- ```
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore"
      Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc"
      Version="1.1.2" />
  ```

8.2.1 Adding ASP.NET MVC Middleware

- To do this, we simply replace the app.Use middleware configuration with the UseMvc extension in the Startup class in Startup.cs file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;

namespace StatlerWaldorfCorp.WebApp
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {

        }

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env, ILoggerFactory loggerFactory)
        {
           app.UseMvc(routes =>
           {
               routes.MapRoute("default",
                template:
                  "{controller=Home}/{action=Index}/{id?}");
```

- ```
                });
            }
        }
    }
```

- For this to work, we'll also need to add a dependency on the NuGet package Microsoft.AspNetCore.Mvc.
- Go ahead and run this application with the usual command-line tools (dotnet restore, dotnet run) and see what happens. You should simply get 404s on every possible route because we have no controllers.

8.2.2 Adding a Controller

- Controllers should do the following and nothing more:
- 1. Accept input from HTTP requests.
- 2. Delegate the input to service classes that are written without regard for
- HTTP transport or JSON parsing.
- 3. Return an appropriate response code and body.
- In other words, our controllers should be very, very small.
- To add a controller to our project, let's create a new folder called Controllers and put a class in it called HomeController.
-
```
using Microsoft.AspNetCore.Mvc;

namespace StatlerWaldorfCorp.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Hello World";
        }
    }
}
```

- If you run the app from the command line and hit the home URL (e.g., http://localhost:5000 or whatever port you're running on) you'll see the text "Hello World" in your browser.

8.2.3 Adding a Model

- The role of the model is, as you might have guessed, to represent the data required by the controller and the view to present some form of interaction between the user and an application.
- Create a model representing a simple stock quote (created in a new Models folder).
- ```
  namespace StatlerWaldorfCorp.WebApp.Models
  {
      public class StockQuote
      {
          public string Symbol { get; set; }
          public int Price { get; set; }
      }
  }
  ```

8.2.4 Adding a View

- Just like with the controller and the model, there is a default convention for locating the views that correspond to controllers.
- For example, if we wanted to create a view for the HomeController's Index method, we would store that view as Index.cshtml in the Views/Home directory.

```
<html>
<head>
    <title>Hello world</title>
</head>
<body>
    <h1>Hello World</h1>

        <h2>Stock Quote</h2>

            Symbol: @Model.Symbol<br/>
            Price: $@Model.Price<br/>

</body>
</html>
```
```

- Now we can modify our home controller to render a view instead of returning simple text:

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using webapp.Models;

namespace webapp.Controllers
{
    public class HomeController : Controller
    {
        public async Task<IActionResult> Index()
        {
            var model = new StockQuote { Symbol = "HLLO",
                                         Price = 3200 };
            return View(model);
        }
    }
}
```

- UseDeveloperExceptionPage method to our Startup class, in the Configure method.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace StatlerWaldorfCorp.WebApp
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddEnvironmentVariables();

            Configuration = builder.Build();
        }
```

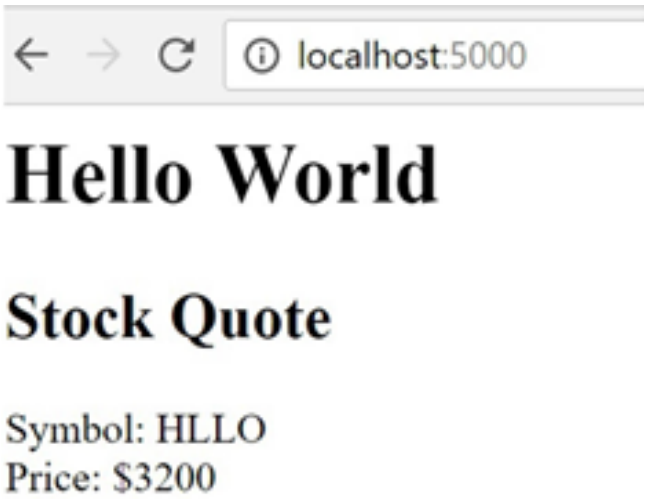
```
        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
           IHostingEnvironment env, ILoggerFactory loggerFactory)
        {
           loggerFactory.AddConsole();
           loggerFactory.AddDebug();

           app.UseDeveloperExceptionPage();
           app.UseMvc(routes =>
           {
            routes.MapRoute("default",
              template: "{controller=Home}/{action=Index}/{id?}");
           });
           app.UseStaticFiles();
        }
    }
}
```

- With this new Startup class, we should be able to do a dotnet restore and a dotnet run to start our application.
- Hitting the home page should combine our controller, our view, and our model to produce a rendered HTML page for the browser



8.2.5 Invoking REST APIs from JavaScript

- Today, the most common type of web application is a single-page application (SPA) that just loads up in the browser and communicates with one or more APIs—no server-side templating is involved.
- In a single-page app, the server renders an HTML page along with links to include a mountain of JavaScript.
- The JavaScript loads in the client browser and then interacts with a RESTful API exposed by the web application in order to provide the end users with the type of experience they've come to expect from modern web and mobile applications.
- First, let's create an API endpoint to use by adding a new controller, the ApiController.

```
using Microsoft.AspNetCore.Mvc;
using webapp.Models;

namespace webapp.Controllers
{
    [Route("api/test")]
    public class ApiController : Controller
    {
        [HttpGet]
        public IActionResult GetTest()
        {
            return this.Ok(new StockQuote
             {
               Symbol = "API",
               Price = 9999
             });
        }
    }
}
```

- If you run the application again right now, then you can hit http://localhost:5000/api/test with your favorite browser and you'll see a JSON payload (with lowercased property names by default) that looks like this:
- {
- "symbol" : "API",
- "price" : 9999
- }
- Now that we've got an API to consume, let's modify our single view so that it grabs some JavaScript to consume it.

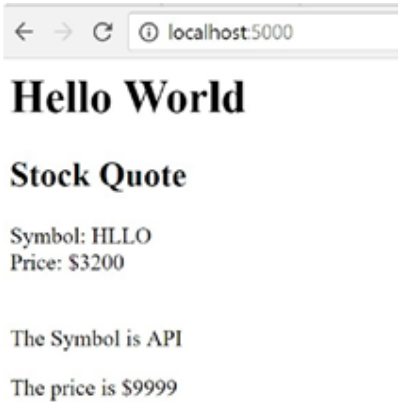
```
<html>
<head>
    <title>Hello world</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
    </script>
    <script src="/wwwroot/Scripts/hello.js"></script>
</head>

   <body>
        <h1>Hello World</h1>
        <div>
            <h2>Stock Quote</h2>
            <div>
                Symbol: @Model.Symbol<br/>
                Price: $@Model.Price<br/>
            </div>
        </div>
        <br/>
        <div>
            <p class="quote-symbol">The Symbol is </p>
            <p class="quote-price">The price is $</p>
        </div>
   </body>
   </html>
```

- We include JQuery here as well as a new script, hello.js.
- We're going to add this to a new directory called wwwroot by convention.
- Our hello.js script just waits for the page to be ready and then consumes our API and appends the result of the data call to the new paragraph elements we've added to the page.

```
$(document).ready(function () {
    $.ajax({
        url: "/api/test"
    }).then(function (data) {
        $('.quote-symbol').append(data.symbol);
        $('.quote-price').append(data.price);
    });
});
```

- This is just some straightforward jQuery that makes an Ajax call to our API endpoint.
- The object that comes back will have the symbol and price properties, which we'll use to append to the new paragraph elements.
- The static files, like our image assets, stylesheets, and JavaScript files, are all made available to browsers through the use of the UseStaticFiles extension method we used in our Startup class.
- We've established the content root (you can see it show up in the debug output when you run the application) with the call to SetBasePath in the Startup class's constructor, which allows our static files to default to the wwwroot directory.
- Now when we launch the application, we should get what we're expecting when we try and load http://localhost:5000.



8.3 Building Cloud-Native Web Applications

- **API First**
- When building applications that consume services, you cannot build that application unless you know the public contract for that service's API.
- Fixed public APIs will do your organization a ton of good and save you a mountain of headaches in the long run.
- Configuration
- When we move toward real production pipelines for continuously delivering microservices in multiple environments as well as resilient green/blue delivery of applications visible to the outside world, the need to store our configuration outside the application becomes a mandate for all teams.
- Logging
- The file system on which your application is running in the cloud needs to be treated as though it is ephemeral (temporary).
- At any time, the disk supporting your application can be taken away, and that instance of your application torn down and restarted on another coast, country, or even continent.
- Developers should log all informational messages to stdout (or, as .NET developers view it, the console).
- We don't need complex logging systems or file rollover and purging logic embedded in our applications. We just write to stdout or stderr.
- **Session State**
- We must use an out-of-process provider.
- Whether it's the easy-to-use and favorite wizard target of the SQL Server session state or a different technology like Redis or Gemfire, the requirement remains the same: if we're deploying our application to the cloud, it cannot use inmemory session state.
- We should never store anything in memory that will last beyond the lifetime of an individual HTTP request.
- If something needs to live longer than that, it should probably be the responsibility of a backing service or an out-of-process cache.
- Data Protection

- Middleware that we use will encrypt and decrypt data for us without ever really getting in our way.
- If we're going to involve data protection, we need to apply the same out-of-process mentality to the storage of keys.
- We need to use an off-the-shelf key vault, a cloud-based key vault, or roll our own solution with storage like Redis or another database.
- Backing Services
- The location and nature of our backing services should be exposed to us through the environment, and never through configuration or code
- Environment Parity
- The only configuration values that should be checked into source control with your main application code are values that never change across environments; values that, when they change, actually warrant the release of a new version of your application.
- In most real-world cases, applying this rule reduces the size of an application's configuration file to either nothing, or some really tiny artifact.
- It means that directly or indirectly, your application must invoke both the AddCommandLine method and the AddEnvironmentVariables method during startup.
- Port Binding
- The PaaS() [platform as a service] environment needs to tell your application which port has been reserved for it within the isolated container currently hosting your application.
- This port can (and almost always does) change from one startup to another for your application.
- Legacy code like Windows Communication Foundation port bindings that specifically try and grab actual ports on a virtual or physical machine are incompatible with the idea of container-based port mappings.
- To support container-assigned ports in any cloud environment, your application needs to allow for command-line override of the server URLs, with the server.urls property, as shown here:
- dotnet run --server.urls=http://0.0.0.0:90210
- dotnet run --server.urls=http://0.0.0.0:90210

- PaaS platforms often make available the port to which your application must bind as an environment variable called PORT.
- This means your application will need to ingest environment variables and make them available inside the IConfiguration instance injected into your app.
- To do this, you'll need to make sure your app invokes both AddCommandLine and
- AddEnvironmentVariables.
- **Telemetry**
- Monitoring your application in the cloud is vastly different from how you monitor it when it is up close and you can attach all kinds of debuggers and diagnostic equipment to it.
- This applies to ASP.NET legacy applications as well as .NET Core services.
- Authentication and Authorization
- Securing applications and services in the cloud shouldn't be all that different from securing legacy web applications running in your own data centers.
- The easiest shortcut for intranet applications is to simply embrace Windows authentication and pull the user's information from a Kerberos-based browser identity challenge.
- This isn't going to work when our services are running on an ephemeral operating system that is probably not Windows (since we're using .NET Core) and, even if it was Windows, isn't joined to a particular workgroup or domain that would allow for normal Windows authentication to work.

8.4 Cloud Native Factors.

8.4.1 External Configuration

- With following code
- using (var httpClient = new HttpClient())
- {

  httpClient.BaseAddress = new Uri("http://foo.bar/baz");
- ...
- }

- The address of the backing service is hardcoded in your application code.
- When you commit this to your version-control system, the URL is sitting there, unaltered.
- This is even more problematic if you've embedded credentials in the URL. This value can't easily change from one environment to the next, and you have to recompile every time you decide to change hostnames.
- Alternate option 1: move the URL out of the C# code and into a web.config file.
- But this is also as hard-coded.
- You should consider any values sitting in a configuration file (web.config, appsettings.json, whatever) as part of your code.
- Alternative 2: Move the URLs and credentials out of configuration files, out of C# files, and into environment variables.
- Written this way, our code makes it obvious what configuration parameters it needs in order to function, but it leaves the responsibility of supplying those values up to the environment.

8.4.2 Backing Services

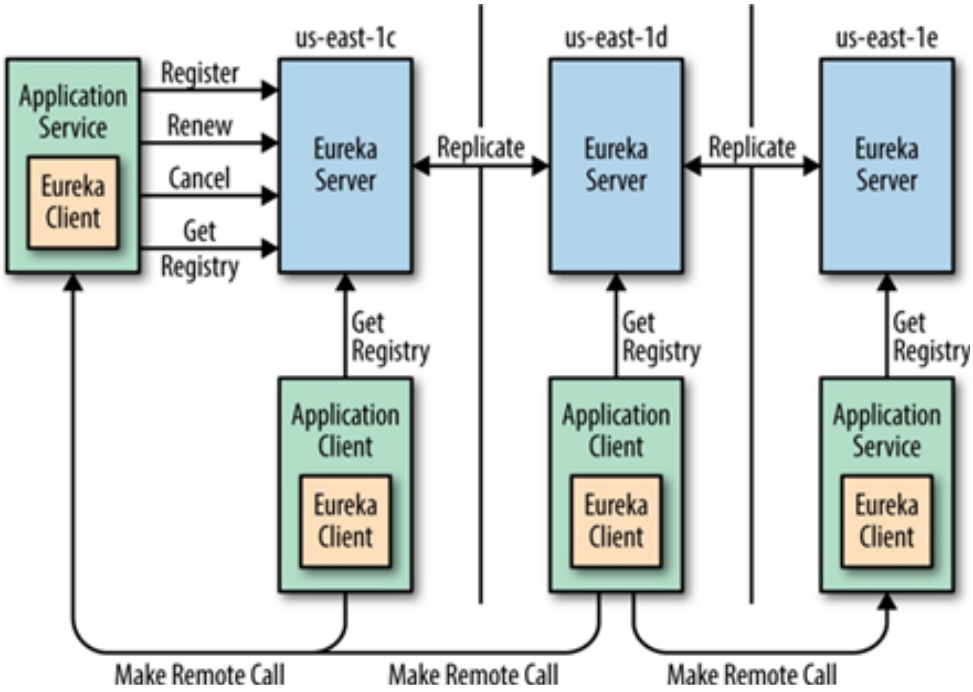
- Everything your application needs must be treated as though it is a backing service.
- Whether you need binary storage for files, a database, another web service, a queue service, or anything else, the thing you need should be loosely coupled, and configured from the environment.
- There are two ways to bind a resource that is a backing service: static binding and dynamic (runtime) binding.
- **Statically bound resources:**
- Need to allow for environment based replacement of default values to connect to databases, web services, and queuing services, this binding is fixed by the environment.
- Whether defined by automation tools or DevOps personnel, the binding between the service and its resource is persistent and made available to the application at start time, and it does not change.

- **Dynamically bound resources:**
- This binding is not fixed and can actually change at runtime between requests to the application.
- This dynamic, loose runtime coupling between apps and bound resources facilitates more advanced functionality like failover, load balancing, fault tolerance—all with no visible impact to the application code.

8.5 Introducing Netflix Eureka.

- In order to discover services at runtime, we need something that serves as a service registry; a central catalog of services.
- This registry and the features it offers can vary from product to product.
- Most of the service registries provide at the basic level a list of services, metadata about the services, and their endpoint(s).
- Netflix's infrastructure is predominantly run on top of Amazon Web Services.
- AWS has plenty of functionality available for load balancing at the edge, and it has a naming service (Route 53) that can function as a full DNS service, neither of these services are entirely appropriate for mid-tier service naming, registry, and load balancing.
- With Eureka, Netflix built its own internal product to manage the service registry that allowed for failover and load balancing.
- From a developer's perspective, your service code interacts with a Eureka server by registering itself when it starts up.
- If you need to discover and consume other backing services, then you can ask the Eureka server for some or all of the service registry.
- Your service also emits a heartbeat to the Eureka service at some interval (usually 30 seconds).
- If your service fails to send a heartbeat to Eureka after some number of intervals, it will be taken out of the registry.
- If there are multiple instances of your service running, and consumers of your service are talking to Eureka to find your service, then they will stop getting the URL of the service that was taken out of the registry due to heartbeat failure and will simply refer to the service instances that are up and running.
- Eureka is also not the only player in the service registry and discovery game.

- There are a number of other companies and products available that provide everything from the bare-bones service registry to more full-featured registry, discovery, and fault- tolerance functionality.



- If you want to try out Eureka without the commitment of having to build it from source or install a full copy on a server, you can just run it from a docker hub image, as shown in the following command:
- **$ docker run -p 8080:8080 -d --name eureka -d netflixoss/eureka:1.3.1**
- This will run a default, nonproduction copy of the server in your Docker virtual machine and map port 8080 from inside the container to your local machine.
- This means, assuming port 8080 is available, that when you statically bind an application to this Eureka server, you'll use the URL http://localhost:8080/eureka.

8.6 Discovering and Advertising ASP.NET Core Services.

- We're going to be building a suite of services that support an ecommerce application.
- The edge service is responsible for exposing a product catalog.
- This catalog has standard API endpoints for exposing a list of products as well as product details.

- There is also an inventory service that is responsible for exposing the real-time status of physical inventory.
- The product service will need to discover the inventory service and make calls to it in order to return enriched data when asked for product details.
- To keep things simple, our sample has just these two services.

8.6.1 Registering a Service

- The first part of our sample is the inventory service, a service that needs to be dynamically discovered at runtime to provide real-time inventory status.
- In our case, the Steeltoe project (Steeltoe discovery client library) maintains a number of client libraries for Netflix OSS projects, including Eureka.
- The Steeltoe library allows us to supply some configuration information using the standard .NET Core configuration system.
- The key things that we need to declare are the name of our application (this is how it will be identified in the registry) and the URL pointing to the Eureka server, as shown here:

```
{
 "spring": {
   "application": {
     "name":  "inventory"
   }
 },
 "eureka": {
   "client": {
     "serviceUrl": "http://localhost:8080/eureka/",
     "shouldRegisterWithEureka": true,
     "shouldFetchRegistry": false,
     "validate_certificates": false
   },
   "instance": {
       "port": 5000
     }
   }
 }
```

- Another key part of this configuration is the value shouldRegisterWithEureka.
- If we want our service to be discoverable then we must choose true here.
- The next setting, shouldFetchRegistry, indicates whether we want to discover other services.
- Put another way, we need to indicate whether we're consuming registry information or producing it—or both.
- Our inventory service wants to be discovered and does not need to discover anything else; therefore it will not fetch the registry, but it will register itself.
- We load the appsettings.json file with our discovery client configuration:
-

```
var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: false,
                reloadOnChange: true)
                .AddEnvironmentVariables();

Configuration = builder.Build();
```

- Then we'll use Steeltoe's AddDiscoveryClient extension method in our Startup class's ConfigureServices method:
- services.AddDiscoveryClient(Configuration);
- Finally, we just need to add a call to UseDiscoveryClient in our Configure method:
- app.UseDiscoveryClient();

8.6.2 Discovering and Consuming Services

- We're going to build: the catalog.
- This service exposes a product catalog and then augments product detail requests by querying the inventory service.
- The key difference between this service and the others we've built so far is that this one will dynamically discover the URL of the catalog service at runtime.

- We'll configure the client almost the same way we configured the inventory service:

```
"spring": {
   "application": {
     "name": "catalog"
   }
 },
 "eureka": {
   "client": {
     "serviceUrl": "http://localhost:8080/eureka/",
     "shouldRegisterWithEureka": false,
     "shouldFetchRegistry": true,
     "validate_certificates":  false
   }
 }
```

- The difference is that the catalog service doesn't need to register (since it does not need to be discovered), and it should fetch the registry so it can discover the inventory service.
- Take a look at the code for the HttpInventoryClient class, the class responsible for consuming the inventory service:

```
using StatlerWaldorfCorp.EcommerceCatalog.Models;
using Steeltoe.Discovery.Client;
using System.Threading.Tasks;
using System.Net.Http;
using Newtonsoft.Json;

namespace StatlerWaldorfCorp.EcommerceCatalog.InventoryClient
{
    public class HttpInventoryClient : IInventoryClient
    {
        private DiscoveryHttpClientHandler handler;
        private const string INVENTORYSERVICE_URL_BASE =
              "http://inventory/api/skustatus/";

        public HttpInventoryClient(IDiscoveryClient client)
        {
            this.handler = new DiscoveryHttpClientHandler(client);
        }

        private HttpClient CreateHttpClient()
        {
```

```
            return new HttpClient(this.handler, false);
        }

        public async Task<StockStatus> GetStockStatusAsync(int sku) {
            StockStatus stockStatus = null;

            using (HttpClient client = this.CreateHttpClient())
            {
                var result =
                      await client.GetStringAsync(
                        INVENTORYSERVICE_URL_BASE + sku.ToString());
                stockStatus =
                      JsonConvert.DeserializeObject<StockStatus>(
                            result);
            }

            return stockStatus;
        }
    }
}
```

- The .NET Core HttpClient class has a variant of its constructor that lets you pass in an instance of your own HttpHandler.
- The DiscoveryHttpClientHandler provided by Steeltoe is responsible for swapping the service name in your URL with the actual, runtime-discovered URL.

  This is what allows our code to rely on a URL like http://inventory/api/skustatus, which can then be converted by Steeltoe and Eureka to something like

  http://inventory.myapps.mydomain.com/api/skustatus.
- To run the inventory service, the catalog service, and Eureka all at the same time on your computer, use the following steps.
- First, start the Eureka server:
- $ docker run -p 8080:8080 -d --name eureka \
- -d netflixoss/eureka:1.3.1
- Then start the inventory service on port 5001:
- $ cd <inventory service>
- $ dotnet run --server.urls=http://0.0.0.0:5001
- To run the service in Docker, use the following docker run command:

- $ docker run -p 5001:5001 -e PORT=5001 \
- -e EUREKA___CLIENT_____SERVICEURL=http://192.168.0.33:8080/eureka/ \
- dotnetcoreservices/ecommerce-inventory
- When you provide the configuration overrides here, make sure you use your actual machine's IP address.
- When running inside the Docker image, referring to localhost doesn't do anyone any good.
- And finally, start the catalog service on port 5001:
- $ cd <catalog service>
- $ dotnet run --server.urls=http://0.0.0.0:5002
- Now you can issue the appropriate GET requests to the product API for the product list and product details:
- GET http://localhost:5002/api/products
- Retrieve product list
- GET http://localhost:5002/api/products/{id}
- Retrieve product details, which will invoke the inventory service, whose URL is dynamically discovered via Eureka.

8.7 DNS and Platform Supported Discovery.

- Using Eureka (including the helper classes that come with Steeltoe), there's still code that has to manually replace logical service descriptors (e.g., inventory) with IP addresses or fully qualified domain names like inventory.mycluster.mycorp.com.
- Service discovery, registration, and fault detection are all things should be nonfunctional requirements.
- As such, application code shouldn't contain anything that tightly couples it to some implementation of service discovery.
- Platforms like Kubernetes have plug-ins like SkyDNS that will automatically synchronize information about deployed and running services with a network-local DNS table.

This means that without any client or server dependencies, you can simply consume

a service at a URL like http://inventory and your client code will automatically resolve to an appropriate IP address.

- When evaluating how you're going to do discovery, you should see if there might be a way to accomplish it without creating a tight coupling or dependency in your application code.

8.8 Summary

- Using ASP.Net core we can create a web application including MVC.
- We can also use Javascript and jQuery that makes an Ajax call to our API endpoint.
- Service discovery allows one service to be aware of the URLs and status of all the services on which it depends.
- We can even use DNS service to map URLs.
- One service can consume other services.
- Making service discovery dynamic by using registry.

8.9 Reference for further reading

- Building Microservices with ASP.NET Core by Kevin Hoffman (chapters 7 and 8)
- https://docs.microsoft.com/en-us/aspnet/core/getting-started/?view=aspnetcore- 3.1&tabs=windows

Unit V

9

CONFIGURING MICROSERVICE ECOSYSTEMS AND SECURING APPLICATIONS AND MICROSERVICES

Unit Structure

9.0 Objective

This chapter would make you understand the following concepts:

Configuring Microservice Ecosystems

- Using Environment
- Variables with Docker
- Using Spring Cloud Config Server
- Configuring
- Microservices with etcd

Securing Applications and Microservices

- Security in the Cloud
- Securing ASP.NET Core Web Apps
- Securing
- ASP.NET Core Microservices.

9.1 Configuring Microservice Ecosystems

In the Microservice Ecosystems Configuration is one of the main parts of architecture and application that are regularly passed over by product teams. A various teams just consider that the bequest examples for placing applications will effort well in the cloud. It is very easy to assume that we will "just" add all configuration using environment variables.

Now we see the configuration in a microservice ecosystem needs attention to lots of other matters, including:

- *The main source of configuration information is Flexibility and reliability.*
- Sustenance for big and complex configuration information likely too burdensome to cram into a handful of environment variables
- Defining whether your application wants to reply to live updates or real-time changes in configuration values, and if so, how to provision for that
- Capability to support things like feature flags and complex ladders of settings
- Perhaps supporting the storage and recovery of secure (encrypted) information or the encryption keys themselves.

9.2 Using Environment Variables with Docker

This very is essentially honestly cool to graft with environment variables and Docker. Environment variables are used by cloud application when they need. We are going to receive various vigorous configuration mechanisms.

We can also have set of configuration available, we should also figure out which settings can be overridden by environment variables at application startup.

We can also explicitly set configuration values using name-value pairs as shown in the following command:

$ sudo docker run -e SOME_VAR='foo' \

-e PASSWORD='foo' \

-e USER='bar' \

-e DB_NAME='mydb' \

-p 3000:3000 \

--name container_name microservices-aspnetcore/image:tag

Or we can also avoid passing explicit values on the command line or we can also forward environment variables from the beginning environment into the vessel by only not passing values or using the equals sign, as shown here:

$ docker run -e PORT -e CLIENTSECRET -e CLIENTKEY [...]

It will take the PORT, CLIENTSECRET, and CLIENTKEY environment variables from the shell in which the command was run and pass their values into the Docker container without exposing their values on the command line, avoiding a potential safety susceptibility or dripping of private information.

When we have a huge number of environment variables to pass into your container, we can also give the docker command the name of a file that holds name-value pairs: $ docker run --env-file ./myenv.file [...]

When we are running higher-level container transposition tool like Kubernetes, then we will have to contact to more graceful ways to manage our environment variables and how they get injected into our containers. Using Kubernetes, we can use a notion called ConfigMap to create external configuration values accessible to our containers without having to create complex launch commands or manage stuffed start scripts.

9.3 Using Spring Cloud Config Server

The more difficult thing is to surrounding configuration management for services dishonesties in the mechanics of inserting values into environment variables, but in every day maintenance of the values themselves.

What is the process to see the huge source of truth for the configuration values has changed? What is processor to change them, and how do we implement security controls to avoid these values from existence by illegal people and keep the values hidden from those avoid of appropriate access?

When values are change , how we can go back and see what are the previous values were? when we are thinking that we apply the processor like a Git source to manage configuration values, then we are not alone.

Same idea of developer of Spring Cloud Config Server (SCCS), why redevelop safety, version, auditing , etc…when Git already solved the issues? In its place they develop a service that exposes the values enclosed in Git source through a RESTful A PI. This API exposes URLs in the below format:

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

Our application name is soo, then all of the {application} sections in the previous template it is replaced with soo. When we have to see the values of configuration contain in the development profile , then we have to issue a DET request to the / soo/development URL.

When we have to know to more about Spring Cloud Config Server then we have to start with the documentation.

Java developers are targeted towards documentation and code, there are various other clients who can speak about SCCS, containing a .NET core application. we have to just add a information to the Steeltoe.Extensions.Configuration.ConfigServer NuGet package.

Then we have to include configure our application so it will get the correct settings from the correct location. That means we have to define a Spring application name and provide the URL to the configuration server in our appsettings.json file.

```
{
“spring”: {
“application”: {
“name”: “foo”
},
“cloud”: {
“config”: {
“uri”: “http://localhost:8888”
}
}
},
“Logging”: {
“IncludeScopes”: false,
“LogLevel”: {
“Default”: “Debug”,
“System”: “Information”,
“Microsoft”: “Information”
}
}
}
```

Using this configuration setting, our initial method looks nearly to it, as per all other applications.

```
public Startup(IHostingEnvironment env)
{
var builder = new ConfigurationBuilder()
.SetBasePath(env.ContentRootPath)
.AddJsonFile(“appsettings.json”, optional: false,
reloadOnChange: false)
.AddEnvironmentVariables()
.AddConfigServer(env);
Configuration = builder.Build();
}
```

Futher changes need to include support to the configuration server come in the ConfigureServices method. Initially we have to call AddConfigServer, which allows us to get the settings of applications as recovered from the server in an IOptionsSnapshot, who are available to inclusion to our supervisors and other code:

```
public void ConfigureServices(IServiceCollection services)
{
services.AddConfigServer(Configuration);
services.AddMvc();
services.Configure<ConfigServerData>(Configuration);
}
```

To hold the data using the class from the config server is showed after the sample configuration which can found in Spring Cloud server sample repository:

```
public class ConfigServerData
{
public string Bar { get; set; }
public string Foo { get; set; }
public Info Info { get; set; }
}
public class Info
{
public string Description { get; set; }
public string Url { get; set; }
}
```

If we have the situation where we inject our C# clas and the configuration server client setting when we required them:

```
public class MyController : Controller
{
private IOptionsSnapshot<ConfigServerData>
MyConfiguration { get; set; }
private ConfigServerClientSettingsOptions
ConfigServerClientSettingsOptions { get; set; }
public MyController(IOptionsSnapShot<ConfigServerData> opts,
```

```
IOptions<ConfigServerClientSettingsOptions>
clientOpts)
{
...
}
....
}
```

Using this setup and running configuration server, the opts variable in the constructor it cover all of the related configuration for our application.

When we execute the config server, we can build and present the code from GitHub. when we want, but not all of us have a functioning Java/Maven development atmosphere up and running.

Using Docker image very easy to start a configuration server:

```
$ docker run -p 8888:8888 \
-e SPRING_CLOUD_CONFIG_SERVER_GIT_URI=https://github.com/spring-cloudsamples/ \
config-repohyness/spring-cloud-config-server
```

It will begin with start the server and point it at the sample GitHub repo declared previous to get the “soo” application’s configuration properties.

Using the following command we get the below command when server run in proper way

```
$ curl http://localhost:8888/foo/development
```

Using a config server Docker image running locally and the C# code shown in the chapter we will able to play with revealing outside configuration data to our .NET Core microservices.

9.4 Configuring Microservices with etcd

Spring Cloud Config Server can have various replacement but on of the very popular replacement is etcd. It can have the lightweight, circulated key value stock.

Here we put most complicated information need to support a distributed system. Raft consensus algorithm can use etcd product to communicate with nobels. More then 500 projects on GitHub depends on etcd.

We have to install local version of etcd to check the documentation or we have to run it from Docker image. Also we have etcdctl command –line utility to work with key-value grading in etcd that look like a simple folder structure. It will come free when we install etcd.with a Mac, we can just beverage to install etcd and you will need access to the tool. but we have to check the documentation for Windows and Linux instruction.

To pass the addresses of the cluster peers as well as the username and password and other options every time we have to run etcdctl command. To do easy we have to create as alias as follows:

```
$ alias e=’etcdctl --no-sync \
--peers https://portal1934-21.euphoric-etcd-
31.host.host.composedb.com:17174,\
https://portal2016-22.euphoric-etcd-31.host.host.composedb.com:17174
\
-u root:password’
```

We want to change the root:password which will used our installation. weather we can running locally or cloud hosted. etcd can have some basic command when we have got alias configuration and you have access to running.

mk
Creates a key and can optionally create directories if you define a deep path for the key.

set
Sets a key’s value.

rm
Removes a key.

ls

Queries for a list of subkeys below the parent. In keeping with the filesystem analogy, this works like listing the files in a directory.

update
Updates a key value.

watch
Watches a key for changes to its value.

Armed with a command-line utility, let's issue a few commands:

```
$ e ls /
$ e set myapp/hello world
world
$ e set myapp/rate 12.5
12.5
$ e ls
/myapp
$ e ls /myapp
/myapp/hello
/myapp/rate
$ e get /myapp/rate
12.5
```

Above session first inspected the root and saying that there was nothing there. Then, the myapp/hello key was produced with the value world and the myapp/rate key was produced with the value 12.5. This indirectly created /myapp as a parent key/ directory. Because of its status as a parent, it didn't have a value.

When executing these commands, we refreshed our elaborate dashboard on compose.Newly created values and keys of io's website as shown in Figure 9.1.

Figure 9.1.: Compose.io's etcd dashboard

After a configuration server and it has data ready for us to consume—but how are we going to consume it? For that we create custom ASP.NET configuration provider.

9.5 Creating an etcd Configuration Provider

Now we need to get how add multiple different configuration sources with the AddJsonFile and AddEnvironmentVariables methods. Now our aim to add an AddEtcdConfiguration method that plugs into a executing etcd server and takes values that seem as though they are a natural part of the ASP.NET configuration system.

9.5.1. Creating a configuration source

First we have to add a configuration source. The main funcation of a configuration source is to generate an instance of a configuration builder. Appreciatively these are very easy edges and there's already a starter ConfigurationBuilder class for us to shape upon.

Here's the new configuration source**:**

```
using System;
using Microsoft.Extensions.Configuration;
namespace ConfigClient
{
public class EtcdConfigurationSource : IConfigurationSource
{
public EtcdConnectionOptions Options { get; set; }
public EtcdConfigurationSource(
EtcdConnectionOptions options)
{
this.Options = options;
}
public IConfigurationProvider Build(
IConfigurationBuilder builder)
{
return new EtcdConfigurationProvider(this);
}
}
}
```

When we communicate with ectd then we have to know some information.so we will identify the same values we give to CLI earlier:

```
public class EtcdConnectionOptions
{
public string[] Urls { get; set; }
public string Username { get; set; }
public string Password { get; set; }
public string RootKey { get; set; }
}
```

7.5.2. Creating a configuration builder

Now we have to create a configuration builder. The Data that stores simple keyvalue pairs is known as the class from which will inherit maintains a protected dictionary. it is convenient for a sample, so Now it very easy to use. The configuration of etcd would need configuration section, so so /myapp/rate would become myapp:rate (nested sections) rather than a single section named /myapp/rate:

```
using System;
using System.Collections.Generic;
using EtcdNet;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Primitives;
namespace ConfigClient
{
public class EtcdConfigurationProvider : ConfigurationProvider
{
private EtcdConfigurationSource source;
public EtcdConfigurationProvider(
EtcdConfigurationSource source)
{
this.source = source;
}
public override void Load()
{
EtcdClientOpitions options = new EtcdClientOpitions()
{
Urls = source.Options.Urls,
Username = source.Options.Username,
```

```
Password = source.Options.Password,
UseProxy = false,
IgnoreCertificateError = true
};
EtcdClient etcdClient = new EtcdClient(options);
try
{
EtcdResponse resp =
etcdClient.GetNodeAsync(source.Options.RootKey,
recursive: true, sorted: true).Result;
if (resp.Node.Nodes != null)
{
foreach (var node in resp.Node.Nodes)
{
// child node
Data[node.Key] = node.Value;
}
}
}
catch (EtcdCommonException.KeyNotFound)
{
// key does not
Console.WriteLine("key not found exception");
}
}
}
}
```

A production-grade library might recursively sift over whole tree until it had got all values. Separately key-value pair recovered from etcd is only additional to the protected Data member.

That code uses an open source module available on NuGet called EtcdNet and this was stable and reliable. we will find it were compatible with .NET Core.

With a simple extension method like this:

```
public static class EtcdStaticExtensions
{
public static IConfigurationBuilder AddEtcdConfiguration(
this IConfigurationBuilder builder,
EtcdConnectionOptions connectionOptions)
{
return builder.Add(
new EtcdConfigurationSource(connectionOptions));
}
}
```

We can add etcd as a configuration source in our Startup class:

```
public Startup(IHostingEnvironment env)
{
var builder = new ConfigurationBuilder()
.SetBasePath(env.ContentRootPath)
.AddJsonFile("appsettings.json", optional: false,
reloadOnChange: true)
.AddEtcdConfiguration(new EtcdConnectionOptions
{
Urls = new string[] {
"https://(host1):17174",
"https://(host2):17174"
},
Username = "root",
Password = "(hidden)",
RootKey = "/myapp"
})
.AddEnvironmentVariables();
Configuration = builder.Build();
}
```

We have snipped the root password for the instance. We will vary reliant on how we installed etcd or where we are hosting it. When we end through this route, we will maybe essential to “bootstrap” the connection information to the config server through environment variables containing the noble URLs username and password.

9.6 Securing Applications and Microservices

Some of the organizations safety has list generate after application has been developed after that when we build application for the cloud the application build around the idea they cloud not run on organization we own –safety could be postscript or some tedious checkbox on a to-do list. Safety is first main problem when we develop any application and services identical.

9.6.1 Security in the Cloud

Protecting application that path where rule in cloud will not as straightforward when we organize applications to a local data warehouse we have full control on the operating system and the installation.

In above section we cover main issue of designers they will trying to acquaint their current ASP.NET skills or bequest codebases to running strongly in the cloud.

9.6.2 Intranet Applications

When consumers face complexity using internet application allover. To solve that problem organizations build these applications executing on a PaaS on top of accessible, cloud infrastructure, some of our old designs.

Main important point is incapability to do authentication of Windows. Application developers are have been damaged by a long history of built-in support for securing web applications with windows authorizations. Web applications are browser based issues answers with details of recently logged-in users, then server knows how face with information and the users are logged in. It is very effective and easy to build apps safe against a organization internal Active Directory.

PaaS is platform that controls very differently then traditional physical or virtual machine windows developments for running in a public cloud or our own on-premise.

Operating systems that supports our application needs to be measured temporary. We can not undertake that it will have capability to contain a domain. In various cases ,operating system support our cloud application are continuously and

Purposely damaged. that's why organizations prepare safety policies where all virtual machines are damaged and reconstructed through progressing informs to decrease the possible surface area visible for tenacious attacks.

9.6.3 Cookie and Forms Authentication

Every developer who are using ASP.NET web Application should be aware with form authentications. The way of authentication is where an web application shows a custom form to ready to users for their authorizations. Authorizations are converted directly to the application and verification by the application. The cookies marks as authenticated for some period of time, when users successfully log in

No matter where running our application on PaaS that is essentially good or bad for cookie authentication. Conversely it produce extra load on our application.

The main aim of forms authentication need our application to maintain and check authentication. That is we have to deal with safety, encoding, and storing private information.

9.6.4 Encryption for Apps in the Cloud

When we think about encryption we worry about our application because some services do encryption or some of the application do not. So Now a days ASP. NET application find the common use of encryption when developing a safe authentication and session cookies.

When we do this type of encryption will use the machine key in order to encrypt the cookies. Encryption use same machine key to decrypt cookies sent to web application form the web browser. When we use cloud then will not depend on appropriate or an appropriate file setting on those computers. We can start our application inside any container whenever we want, held by any number of virtual machines more number container. We have to take care that a one encryption key will be circulated across all machine on which our application is runs.

Now we take a example where token are regularly cryptographically signed, needing the use of irregular keys for authentication. Where we store our keys if we cannot trust on the existence of a tenacious filesystem, nor we trust on those keys being accessible in the storage of all running instance of our service? Solution to face the storage and maintenance of cryptographic keys as a support facility. This facility from outside to our application in the same way that state the filesystem, database and additional micro services are.

To encrypt the cookies we can use this form of encryption. Then use the similar machine key to decrypt cookies sent to the web application from the web browser.

We con not trust on specific machines or on specific files sitting on those machines. Our application can run inside any container at any time, held by various virtual machines on several of regions. We cannot easily trust on the fact that a single encryption key will be circulated over every machine where our application is runs.

9.6.5 Bearer Tokens

Where application does not have an central authority on the individual users, so we need to develop the code in such a way that they can required evidence of identity and proof of authentication. Here we have various standards that define number of methods for accepting proof of identity, containing OAuth and OpenID Connect.

Very general way to transfer the proof of identity in an HTTP-friendly, comfortable manner is through the bearer tokens. Preferably when an application takes a bearer token it does so over the Authorization header. Bellow example shows:

```
POST /api/service HTTP/1.1
Host: world-domination.io
Authorization: Bearer ABC123HIJABC123HIJABC123HIJ
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (X11; Linux x86_64) etc...etc...etc...
```

To authentication for header takes the value form of a one word that indicates the type of Authorization, tracked by order of characters that has the value of the credentials. We are very aware with commonly used authorization types: Digest and Basic.

In a typical help stream that uses carrier tokens, the administration will separate the token from the Authorization header. Numerous symbolic organizations, as OAuth 2.0 (JWT), are generally encoded in a Base64, URL-accommodating arrangement, so the initial move toward approving those tokens is disentangling them to get at the first substance. On the off chance that a token was encoded with a private key, an assistance will at that point utilize an open key to approve that the token was created by the suitable power.

For a point by point conversation of the JSON Web Token (JWT) arrangement and particulars, don’t hesitate to look at the first RFC. The code tests we will be taking a gander at in this section will utilize the JWT design.

9.7 Securing ASP.NET Core Web Apps

Making sure about an ASP.NET Core web application includes choosing confirmation and approval components and afterward utilizing the fitting middleware. Confirmation middleware looks at approaching HTTP demands, decides whether the client is verified, and, if not, issues the suitable test and diverts.

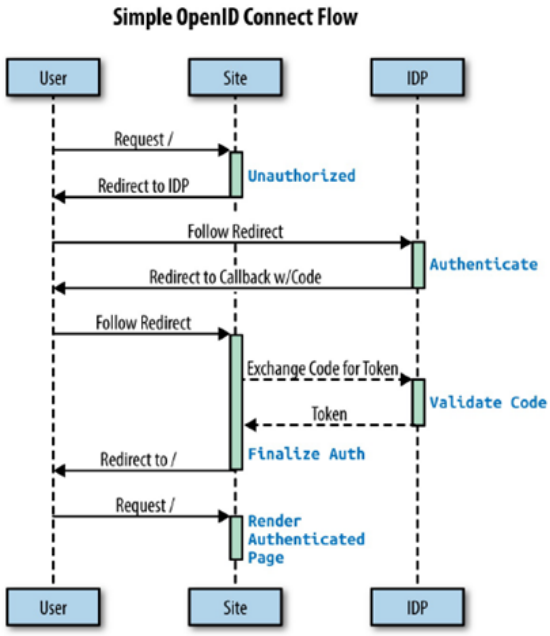
One of the most dependable approaches to perform confirmation in the cloud and keep your applications as concentrated on business rationale as conceivable is using carrier tokens.

For the examples in this part, we'll be concentrating on OpenID Connect and carrier tokens utilizing the JWT standard.

9.7.1 OpenID Connect Primer

Contingent upon the kind of utilization we're building and the security necessities of that application, we have a wide assortment of confirmation streams that we can use. OpenID Connect (we'll simply allude to it as OIDC starting now and into the foreseeable future) is a superset of the OAuth2 standard and contains determinations and guidelines for the ways character suppliers (IDPs), clients, and applications impart safely.

There are approval streams that are planned explicitly for single-page web applications, for versatile applications, and for customary web applications. One of the more straightforward streams accessible for web applications is the one appeared in the succession chart in Figure 9.2.



In this stream, an unauthenticated client asks for a secured asset on a site. The site at that point diverts the client to the personality supplier, giving it directions on the best way to get back to the site after verification. In the event that all works out in a good way, the personality supplier will flexibly to the site a short string with an extremely short termination period called a code. The site (likewise alluded to as the ensured asset in this situation) will at that point promptly make a HTTP POST call to the character supplier that incorporates a customer ID, a customer mystery, and the code.

Consequently, the IDP gives back an OIDC token (in JWT position). When the site has gotten and approved the token, it realizes that the client is appropriately confirmed. The site would then be able to compose a verification treat and divert to a landing page or the first secured asset. The nearness of the treat is currently utilized so the site can sidestep the full circle to the IDP.

There are increasingly mind boggling streams that incorporate the idea of assets and acquiring access tokens with progressively effusive divert circles, however for our example we will utilize the least complex stream. This stream is likewise one of the more secure in light of the fact that the cases bearing token is never uncovered on a URL, just as a fleeting string that is quickly traded over a safe association for a token.

9.7.2 Securing a Service with Client Credentials

The customer qualifications design is perhaps the least difficult approaches to make sure about an assistance. Initially, you speak with the administration just by means of SSL, and second, the code expending the administration is liable for transmitting certifications. These accreditations are typically just called a username and secret key, or, increasingly suitable for situations that don't include human collaboration, a customer key and a customer mystery. Whenever you're taking a gander at an open API facilitated in the cloud that expects you to flexibly a customer key and mystery, you're taking a gander at an execution of the customer accreditations design.

It is likewise genuinely regular to see the customer key and furtively transmitted as custom HTTP headers that start with the X-prefix; e.g., X-MyApp-ClientSecret and X-MyApp-ClientKey.

The code to execute this sort of security is in reality truly straightforward, so we'll avoid the example here. There are, notwithstanding, various drawbacks to this arrangement that originates from its effortlessness.

For instance, what do you do if a specific customer begins manhandling the framework? Would you be able to handicap its entrance? Imagine a scenario where a lot of customers have all the earmarks of being endeavoring a forswearing of administration assault. Would you be able to hinder every one of them? Maybe the most startling situation is this: what occurs if a customer mystery and key are undermined and the purchaser accesses secret data without setting off any conduct cautions that may get them prohibited? What we need is something that consolidates the straightforwardness of versatile qualifications that don't require correspondence with an outsider for approval with a portion of the more reasonable security highlights of OpenID

Associate, similar to the approval of backers, approval of crowd (target), terminating tokens, and the sky is the limit from there.

9.7.3 Securing a Service with Bearer Tokens

Through our investigation of OpenID Connect, we've just observed that the capacity to transmit convenient, autonomously certain tokens is the key innovation supporting the entirety of its validation streams. Conveyor tokens, explicitly those holding fast to the JSON Web Token a determination can likewise be utilized freely of OIDC to make sure about administrations without including any program diverts or the verifiable suspicion of human shoppers.

The OIDC middleware we utilized before in the part expands on the head of JWT middleware that we get from the Microsoft.AspNetCore.Authentication.JwtBearer NuGet bundle. To utilize this middleware to make sure about our administration, we would first be able to make an unfilled administration utilizing any of the past models in this book as reference material or platform. Next, we add a reference to the JWT carrier confirmation NuGet bundle.

In our administration's Startup class, in the Configure technique, we can empower and design JWT conveyor verification, as appeared in Example

Example . Startup.cs

```
app.UseJwtBearerAuthentication(new JwtBearerOptions
{
AutomaticAuthenticate = true,
AutomaticChallenge = true,
TokenValidationParameters = new TokenValidationParameters
{
```

```
ValidateIssuerSigningKey = true,
IssuerSigningKey = signingKey,
ValidateIssuer = false,
ValidIssuer = “https://fake.issuer.com”,
ValidateAudience = false,
ValidAudience = “https://sampleservice.example.com”,
ValidateLifetime = true,
}
});
```

We can control the entirety of the various sorts of things we approve while accepting carrier tokens, including the guarantor’s marking key, the backer, the crowd, and the symbolic lifetime. Approval of things like a symbolic lifetime as a rule additionally expects us to set up choices like permitting some range to oblige clock slant between a symbolic backer and the made sure about help.

In the previous code, we have approval killed for guarantor and crowd, yet these are both genuinely straightforward string examination checks. At the point when we approve these, the backer and the crowd must match precisely the guarantor and crowd contained in the token.

Suppose our administration is a stock administration running on the side of a store called alienshoesfrommars.com. We may see a backer estimation of https://idp.alienshoesfrommars.com and the crowd would be simply the administration, https://stockservice.fulfillment.alienshoesfrommars.com. While show directs that the guarantor and crowd are the two URLs, they shouldn’t be live sites that react to demands so as to approve the token.

You may have seen that we are approving the guarantor marking key. This is fundamentally the main way that we can guarantee that a conveyor token was given by a known and confided in backer. To make a marking key that we need to coordinate the one that was utilized to sign the token, we take a mystery key (some string) and make a SymmetricSecurityKey from it, as appeared here:

```
string SecretKey = “seriouslyneverleavethissittinginyourcode”;
SymmetricSecurityKey signingKey =
new SymmetricSecurityKey(
Encoding.ASCII.GetBytes(SecretKey));
```

As the string shows, you ought to never store the mystery key legitimately in your code. This should originate from a situation variable or some other outer item like Vault or a conveyed store like Redis. An assailant who can acquire this mystery key will have the option to create conveyor tokens freely and have liberated access to your not, at this point secure microservices.

That is basically all we have to begin protecting our administrations with conveyor tokens. We should simply drop the [Authorize] property on controller techniques that need this, and the JWT approval middleware will be conjured for those strategies. Undecorated techniques will permit unauthenticated access as a matter of course (however you can change this conduct too).

To expend our made sure about assistance, we can make a straightforward support application that makes a JwtSecurityToken occasion from a variety of Claim objects, at that point sends those as an Authorization header carrier token:

```
var claims = new[]
{
new Claim(JwtRegisteredClaimNames.Sub, “AppUser_Bob”),
new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString()),
new Claim(JwtRegisteredClaimNames.Iat,
ToUnixEpochDate(DateTime.Now).ToString(),
ClaimValueTypes.Integer64),
};
var jwt = new JwtSecurityToken(
issuer: “issuer”,
audience: “audience”,
claims: claims,
notBefore: DateTime.UtcNow,
expires: DateTime.UtcNow.Add(TimeSpan.FromMinutes(20)),
signingCredentials: creds);
var encodedJwt = new JwtSecurityTokenHandler().WriteToken(jwt);
httpClient.DefaultRequestHeaders.Authorization =
new AuthenticationHeaderValue(“Bearer”, encodedJwt);
var result =
httpClient.GetAsync(“http://localhost:5000/api/secured”).Result;
Console.WriteLine(result.StatusCode);
Console.WriteLine(result.Content.ToString());
```

Here's a made sure about controller technique in our administration that specifies the cases sent by the customer. Note that this code could never at any point be executed if the conveyor token hadn't just been approved by our middleware arrangement:

```
[Authorize]
[HttpGet]
public string Get()
{
foreach (var claim in HttpContext.User.Claims) {
Console.WriteLine($"{claim.Type}:{claim.Value}");
}
return "This is from the super secret area";
}
```

Since the JWT approval middleware has just been composed for us, there's almost no work for us to do so as to help carrier token security for our administrations. On the off chance that we need to have more command over which customers can call which controller strategies, we can exploit the idea of an arrangement. Strategies are simply custom bits of code that are executed as predicates while deciding approval.

For instance, we could characterize an approach called CheeseburgerPolicy and make a made sure about controller technique that requires a legitimate carrier token, yet it likewise necessitates that said symbolic meets the models characterized by the strategy:

```
[Authorize( Policy = "CheeseburgerPolicy")]
[HttpGet("policy")]
public string GetWithPolicy()
{
return "This is from the super secret area w/policy enforcement.";
}
```

Configuring this policy is done easily in the ConfigureServices method. In the following sample, we create CheeseburgerPolicy as a policy that requires a specific claim

(icanhazcheeseburger) and value (true):

```
public void ConfigureServices(IServiceCollection services)
{
services.AddMvc();
services.AddOptions();
services.AddAuthorization( options => {
options.AddPolicy("CheeseburgerPolicy",
policy =>
policy.RequireClaim("icanhazcheeseburger", "true"));
});
}
```

Presently on the off chance that we adjust our comfort application to include another case of this sort with an estimation of valid, we will have the option to conjure normal made sure about controller strategies just as controller techniques made sure about with the CheeseburgerPolicy.

The arrangement can require explicit cases, usernames, attestations, or jobs. You can likewise characterize your own prerequisite that executes IAuthorizationRequirement with the goal that you can include your own approval rationale while not contaminating your individual controllers.

Unit 5

10

BUILDING REAL-TIME APPS AND SERVICES AND PUTTING IT ALL TOGETHER

Unit Structure

10.0 Objective

This chapter would make you understand the following concepts

Building Real-Time Apps and Services

- Real-Time Applications Define
- Websockets in the Cloud
- Using a Cloud Messaging Provider
- Building the Proximity Monitor.

Putting It All Together

- Identifying and Fixing Anti-Patterns
- Continuing the Debate over Composite Microservices
- The Future.

10.1 Building Real-Time Apps and Services

All through the book we have been taking a gander at different ways microservicesaccept info and produce yield. We've seen the customary RESTful offices and we've seen offices that expend and produce messages in lines.

Clients of current web and versatile applications regularly request more than the possible consistency .They need to think about things that are critical to them, and they need to think about them right away. This carries us to the subject of this section: constant administrations.

This part will talk about what the expression "constant" signifies and the sorts of uses most purchasers consider to be inside that class. At that point we'll see at websockets and how conventional websocket programming models drop short in the cloud, and shape an example continuous application that shows the intensity of adding constant informing to an occasion sourced framework.

10.2 Real-Time Applications Defined

Before we can characterize a continuous application we have to characterize ongoing. Target like the period microservices, ongoing is over-burden, abused, and as a rule has in any event two different faculties for each one in the room pondering it.

Definithing.com characterizes it as:
A period used to characterize PC frameworks that update data at a similar rate as they acquire information. Different implications of ongoing suggest that something

is constant on the off chance that it would procedure be able to info and produce yield inside a couple of milliseconds. To me this seems like a really arbitrary worth. Around frameworks with ultra-low potential supplies may reflect ongoing to be a preparing time of two or three hundred microseconds, not milliseconds.

The occasion processor we made is more than fit for preparing input (part area occasions), recognizing closeness, and transmitting nearness distinguished occasions inside a couple of milliseconds. By both of the definitions we've secured up until now, our area occasion processor can be viewed as a continuous framework.

I figure a somewhat more extensive meaning of continuous may be to state that occasions happen with practically zero deferral among receipt and handling. The meaning of "pretty much nothing" here must be one that is settled upon by the advancement group dependent on the framework prerequisites and application space and can't be some self-assertive estimation of some arbitrarily picked unit of measure.

One source says that instances of constant applications may be a rocket direction framework or a carrier booking application. I can totally comprehend the ongoing idea of a rocket direction framework—an implanted processor performing a large number of figurings every second dependent on contribution from handfuls or several sensors so as to control the trip of a shot and report criticism to the ground. Other ongoing applications that fall into a comparative classification may be self-governing vehicles, leisure activity and business drone autopilot programming, and example acknowledgment programming applied to live video takes care of.

Be that as it may, shouldn't something be said about a carrier booking framework? I think this is out of line. A large portion of us have encountered the possible consistency (or once in a while steady) nature of these frameworks. You can book a ticket, and your cell phone may take 24 hours to get your ticket. You may get warnings of a flight postponement or entryway change an hour after that data may have been applicable to you. There are special cases obviously, however for countless cases, these are bunch mode frameworks and time-surveying frameworks that once in a while show the qualities of an ongoing application.

This raises another enemy of example of ongoing frameworks. Now a rare qualities of utilizations that preclude them from the continuous class:

- Your application gathers info and holds up before delivering yield.
- Your application just delivers yield on planned spans or upon outside upgrades that happen on any sort of timetable or are arbitrary in nature.

Main extremely normal attribute or normal for continuous frameworks is that invested individuals are informed of occasions concerning them by means of message pop-up, as opposed to the invested individual playing out a survey or planned inquiry to check for refreshes. We will be talking about push declarations of various sorts all over rest of the share.

10.3 Websockets in the Cloud

We've just secured one type of informing broadly all through this book—the utilization of message lines by means of an informing server like RabbitMQ. At the point when engineers consider constant applications, one thing that frequently strikes a chord is the utilization of websockets to push information and warnings to a live (ongoing) online UI.

Only a couple of years prior, utilizing a site that would refresh and respond to you powerfully would have appeared to be noteworthy and been named as "what's to come." Nowadays we underestimate this sort of thing.

At the point when we go to a site that sells items, we accept it as given that we ought to have the option to have a live talk with help individuals. It's no huge amazement when Facebook springs up notices telling us that somebody's remarked on our post or we see the site change and respond powerfully when somebody retweets one of our tweets.

While not the entirety of this usefulness is upheld expressly through websockets, its greater part was a couple of years back and a lot of it is as yet bolstered through either websockets or something intended to seem like a websocket to designers.

10.3.1 The WebSocket Protocol

The WebSocket convention appeared around 2008 and characterizes the methods by which a diligent, bidirectional attachment association can be made between a program and a server. This permits information to be sent to a server from a web application running in the program, and it permits a server to send information down without requiring the application to "survey" (occasionally check for refreshes, ordinarily on a sliding/exponential tumble off scale).

At a low level, the program demands an association overhaul of the server. When the handshake completes, the program and server change to a different, parallel TCP association for bidirectional correspondence. From the particular, and the relating Wikipedia page, a HTTP a solicitation requesting an association update looks something like this:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13

Origin: http://example.com

The websockets are then used to do everything from pushing spring up warnings via web-based networking media sites to refreshing live, gushing dashboards and observing consoles and in any event, playing intelligent multiplayer games with minimal more than HTML and smart utilization of illustrations and CSS.

10.3.2 Arrangement Models

What does any of this have to do with the cloud? In a conventional organization model, you turn up a server (physical or virtual), you introduce your facilitating item (an IIS web server or some J2EE compartment like WebSphere, for instance) and afterward, you send your application. In the event that your application is versatile and chips away at a ranch, you at that point rehash this procedure over for very server in your homestead or bunch.

At the point when a client associates with a page on your webpage that opens a WebSocket association, that association remains open with whatever server was picked to deal with the underlying solicitation. Until the client hits invigorate or clicks another connection, that WebSocket should work fine and dandy, however, there are different issues that may concoct intermediaries and firewalls.

Suppose since the entirety of your servers are running on EC2 occasions in AWS. At the point when a cloud-based framework is facilitating your virtual machines, they are dependent upon movement, obliteration, and remaking at any second. This is something to be thankful for, and intended to permit your application to scale for

wholly intentions and purposes unbounded. Lamentably, this implies these “live” websocket associations can be broken or get stale and inert without notice.

Moreover, the utilization of consistently up TCP associations with singular servers can affect your own application’s capacity to scale.

Contingent upon the volume of solicitations and information your application code is adjusting, additionally dealing with these associations and the information trade for them can turn into a problematic weight. The arrangement here is as a rule to externalize the utilization of websockets—to offload the administration of websocket associations and information move to something that exists (and scales) outside your application code. Another arrangement that assists with scaling is staying away from websockets altogether and utilizing HTTP-based informing frameworks.

To put it plainly, as opposed to your application overseeing WebSockets all alone, you should let the specialists oversee WebSockets and utilize a cloud informing supplier. It merits recalling that you’re fabricating an application for your business; you’re not (normally) anticipating spend significant time in the specialty of websocket the board.

Regardless of whether you have your own cloud informing server inside your framework or your utilization a different informing supplier facilitated somewhere else in the cloud is up to you and will rely upon your prerequisites and business area.

10.4 Utilizing a Cloud Messaging Provider

We realize that we need our application to have constant capacities. We need our microservices to have the option to push information down to customers, realizing that those customers won’t have a live TCP association with the microservice. We additionally need applications to have the option to utilize the equivalent or a comparative message pipeline to send messages into our back-end.

All together for our microservices to remain cloud-local and hold the capacity to scale and move around uninhibitedly in the cloud, we have to pick an informing supplier to deal with a portion of our constant abilities out of the procedure. The rundown of organizations that give informing administrations is tremendous and developing each day.

- **Apigee** (API gateway and real-time messaging)
- **PubNub** (real-time messaging and presence)
- **Pusher** (real-time messaging and presence)
- **Kaazing** (real-time messaging)
- **Mashery** (API gateway and real-time messaging)
- **Google** (Google Cloud Messaging)
- **ASP.NET SignalR** (real-time messaging hosted in Azure)
- **Amazon** (Simple Notification Service)

The standards you use to choose your informing supplier will be founded altogether on your requirements, the sort of utilization you're building, financial plan, anticipated volume, regardless of whether you're consolidating cell phones or IoT parts, etc.

Notwithstanding which component you pick, you ought to put a brief period in protecting your code from the specific supplier so you can change this without having an over the top sweeping effect. An enemy of defilement layer (ACL) would be a truly decent proposal here to protect your application from usage models from explicit suppliers seeping into your codebase.

In this section, we're going to utilize PubNub. I picked it to some degree subjectively, yet in addition as a result of the basic SDK, incredible documentation, prepared accessibility of open examples, and the way that we can utilize it for exhibit purposes without giving over a charge card number.

Approaching up ensuing are only a couple of the numerous organizations that offer cloud informing either as an independent item or as a feature of a bigger set-up of administrations:

10.5 Building the Proximity Monitor

Over the span of our conversation of Event Sourcing and the Command Query Responsibility Segregation design, we developed an application made of various microservices that identified at whatever point colleagues moved inside scope of one another.

At the point when this framework identifies two close by partners, it radiates a ProximityDetectedEvent onto a line—yet that is the place we quit structuring and coding. What we want to do currently is fabricate a screen that refreshes

progressively at whatever point the backend framework recognizes one of these closeness occasions.

For the reasons for our model, we'll be keeping the UI to something basic, yet it shouldn't take a lot of creative mind to imagine a portion of the potential continuous UIs that may be conceivable here. We could make a guide reconciliation where the current places of the entirety of the colleagues are plotted, and we may bob or enliven colleagues' symbols when the framework has distinguished that they are inside scope of one another. These colleagues may likewise get warnings on their cell phones simultaneously.

Example 10-1. ProximityDetectedEventProcessor.cs

```
using System;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using StatlerWaldorfCorp.ProximityMonitor.Queues;
using StatlerWaldorfCorp.ProximityMonitor.Realtime;
using StatlerWaldorfCorp.ProximityMonitor.TeamService;
namespace StatlerWaldorfCorp.ProximityMonitor.Events
{
public class ProximityDetectedEventProcessor : IEventProcessor
{
private ILogger logger;
private IRealtimePublisher publisher;
private IEventSubscriber subscriber;
private PubnubOptions pubnubOptions;
public ProximityDetectedEventProcessor(
ILogger<ProximityDetectedEventProcessor> logger,
IRealtimePublisher publisher,
IEventSubscriber subscriber,
ITeamServiceClient teamClient,
IOptions<PubnubOptions> pubnubOptions)
{
```

```
this.logger = logger;
this.pubnubOptions = pubnubOptions.Value;
this.publisher = publisher;
this.subscriber = subscriber;
logger.LogInformation("Created Proximity Event Processor.");
subscriber.ProximityDetectedEventReceived += (pde) => {
Team t = teamClient.GetTeam(pde.TeamID);
Member sourceMember =
teamClient.GetMember(pde.TeamID, pde.SourceMemberID);
Member targetMember =
teamClient.GetMember(pde.TeamID, pde.TargetMemberID);
ProximityDetectedRealtimeEvent outEvent =
new ProximityDetectedRealtimeEvent
{
TargetMemberID = pde.TargetMemberID,
SourceMemberID = pde.SourceMemberID,
DetectionTime = pde.DetectionTime,
SourceMemberLocation = pde.SourceMemberLocation,
TargetMemberLocation = pde.TargetMemberLocation,
MemberDistance = pde.MemberDistance,
TeamID = pde.TeamID,
TeamName = t.Name,
SourceMemberName =
$"{sourceMember.FirstName} {sourceMember.LastName}",
TargetMemberName =
$"{targetMember.FirstName} {targetMember.LastName}"
};
publisher.Publish(
```

```
this.pubnubOptions.ProximityEventChannel,
outEvent.toJson());
};
}
public void Start()
{
subscriber.Subscribe();
}
public void Stop()
{
subscriber.Unsubscribe();
}
}
}
```

The primary thing to see in this code posting is the not insignificant rundown of conditions that we'll be infusing into the constructor from the DI specialist co-op:

- Logger l
- Real-time occasion distributer
- Event endorser (line based)
- Team administration customer
- PubNub alternatives

The lumberjack is simple. The continuous occasion distributer, a class that adjusts to\.the IRealtimePublisher interface permits us to distribute a strong message on a given channel (likewise indicated by a string). We will distribute occasions of type ProximityDetectedRealtimeEvent on this channel, serializing the information into JSON.

The occasion endorser tunes in to our line (RabbitMQ), anticipating messages of type ProximityDetectedEvent. At the point when we start and stop our occasion processor, we buy in and withdraw the occasion supporter in like manner. The group administration customer is utilized to inquiry about the group administration for group and part subtleties. We utilize the group and part administration subtleties

to populate the part properties (first and last name) and the group name property on the continuous occasion.

At long last, the PubNub alternatives class holds data like the name of the channel on which the message will be distributed. While our fundamental execution is PubNub, most by far from cloud informing suppliers have some idea of a channel for message distributing, so we ought to be moderately protected trading PubNub out for an alternate supplier on the off chance that we pick.

10.5.2 Creating a real-time publisher class

A good refactor for the future might be to create another small class that is responsible for creating a new instance of a ProximityDetectedRealtimeEvent class from every ProximityDetectedEvent received. This is not just an anti-corruption function, but also augmentation that grabs the team member's name and other user-friendly information.

From a functional purist's perspective, this code doesn't really belong in the high-level processor, but rather should be delegated to a supporting tool that's been tested in isolation.

Moving on from the high-level processor, let's yield a see at the implementation of our IRealtimePublisher interface in Example 10-2, one that uses the PubNub API.

Example 10-2. PubnubRealtimePublisher.cs

```
using Microsoft.Extensions.Logging;

using PubnubApi;

namespace StatlerWaldorfCorp.ProximityMonitor.Realtime

{

public class PubnubRealtimePublisher : IRealtimePublisher

{

private ILogger logger;

private Pubnub pubnubClient;

public PubnubRealtimePublisher(

ILogger<PubnubRealtimePublisher> logger,

Pubnub pubnubClient)

{
```

```
logger.LogInformation(
“Realtime Publisher (Pubnub) Created.”);
this.logger = logger;
this.pubnubClient = pubnubClient;
}
public void Validate()
{
pubnubClient.Time()
.Async(new PNTimeResultExt(
(result, status) => {
if (status.Error) {
logger.LogError(
$”Unable to connect to Pubnub
{status.ErrorData.Information}”);
throw status.ErrorData.Throwable;
} else {
logger.LogInformation(“Pubnub connection established.”);
}
}
));
}
public void Publish(string channelName, string message)
{
pubnubClient.Publish()
.Channel(channelName)
.Message(message)
.Async(new PNPublishResultExt(
(result, status) => {
```

```
if (status.Error) {
logger.LogError(
$"Failed to publish on channel {channelName}:
{status.ErrorData.Information}");
} else {
logger.LogInformation(
$"Published message on channel {channelName},
{status.AffectedChannels.Count}
affected channels, code: {status.StatusCode}");
}
}
));
}
}
}
```

The code here is truly direct. It is only a basic covering around the PubNub SDK. The occurrence of the Pubnub class from the SDK is designed through certain expansions I composed that register a plant with ASP.NET Core.

10.5.3 Infusing the continuous classes

You can perceive how the Pubnub customer and different classes are empowered through DI in the Startup class in Example 10-3.

Example 10-3. Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using StatlerWaldorfCorp.ProximityMonitor.Queues;
using StatlerWaldorfCorp.ProximityMonitor.Realtime;
```

```
using RabbitMQ.Client.Events;
using StatlerWaldorfCorp.ProximityMonitor.Events;
using Microsoft.Extensions.Options;
using RabbitMQ.Client;
using StatlerWaldorfCorp.ProximityMonitor.TeamService;
namespace StatlerWaldorfCorp.ProximityMonitor
{
public class Startup
{
public Startup(IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
loggerFactory.AddConsole();
loggerFactory.AddDebug();
var builder = new ConfigurationBuilder()
.SetBasePath(env.ContentRootPath)
.AddJsonFile("appsettings.json",
optional: false, reloadOnChange: false)
.AddEnvironmentVariables();
Configuration = builder.Build();
}
public IConfigurationRoot Configuration { get; }
public void ConfigureServices(
IServiceCollection services)
{
services.AddMvc();
services.AddOptions();
services.Configure<QueueOptions>(
Configuration.GetSection("QueueOptions"));
```

```
services.Configure<PubnubOptions>(
Configuration.GetSection("PubnubOptions"));
services.Configure<TeamServiceOptions>(
Configuration.GetSection("teamservice"));
services.Configure<AMQPOptions>(
Configuration.GetSection("amqp"));
services.AddTransient(typeof(IConnectionFactory),
typeof(AMQPConnectionFactory));
services.AddTransient(typeof(EventingBasicConsumer),
typeof(RabbitMQEventingConsumer));
services.AddSingleton(typeof(IEventSubscriber),
typeof(RabbitMQEventSubscriber));
services.AddSingleton(typeof(IEventProcessor),
typeof(ProximityDetectedEventProcessor));
services.AddTransient(typeof(ITeamServiceClient),
typeof(HttpTeamServiceClient));
services.AddRealtimeService();
services.AddSingleton(typeof(IRealtimePublisher),
typeof(PubnubRealtimePublisher));
}
public void Configure(IApplicationBuilder app,
IHostingEnvironment env,
ILoggerFactory loggerFactory,
IEventProcessor eventProcessor,
IOptions<PubnubOptions> pubnubOptions,
IRealtimePublisher realtimePublisher)
{
realtimePublisher.Validate();
realtimePublisher.Publish(
```

```
pubnubOptions.Value.StartupChannel,
"{'hello': 'world'}");
eventProcessor.Start();
app.UseMvc();
}
}
}
```

The AddRealtimeService strategy is a static augmentation technique that I made to disentangle the infusion of the execution of an IRealtimePublisher by the administration provider.So far in the book we've been utilizing just the least difficult and most essential highlights of ASP.NET Core's reliance infusion. What we're attempting to do now is ensure that we can make a class (like the PubnubRealtimePublisher) that can have an instant occurrence of the PubNub API infused into it.

To do this neatly and still permit the entirety of our setup to be infused, including the mystery API keys, we have to enlist a processing plant. The production line will be a class that apportions designed cases of the Pubnub class from the PubNub SDK. shows the moderately basic industrial facility class.

Example 10-4. PubnubFactory.cs

```
using Microsoft.Extensions.Options;
using PubnubApi;
using Microsoft.Extensions.Logging;
namespace StatlerWaldorfCorp.ProximityMonitor.Realtime
{
public class PubnubFactory
{
private PNConfiguration pnConfiguration;
private ILogger logger;
public PubnubFactory(IOptions<PubnubOptions> pubnubOptions,
ILogger<PubnubFactory> logger)
{
```

```
this.logger = logger;
pnConfiguration = new PNConfiguration();
pnConfiguration.PublishKey = pubnubOptions.Value.PublishKey;
pnConfiguration.SubscribeKey = pubnubOptions.Value.SubscribeKey;
pnConfiguration.Secure = false;
}
public Pubnub CreateInstance()
{
return new Pubnub(pnConfiguration);
}
}
}
```

Given PubNub choices (we have these in our appsettings.json document, which can be abrogated through condition factors), this class makes another case of the Pubnub class. The genuine stunt, and the code that will probably prove to be useful in your improvement ventures, is the static augmentation technique to plug this manufacturing plant into the DI component, appeared in Example 10-5.

RealtimeServiceCollectionExtensions.cs

```
using System;
using Microsoft.Extensions.DependencyInjection;
using PubnubApi;
namespace StatlerWaldorfCorp.ProximityMonitor.Realtime
{
public static class RealtimeServiceCollectionExtensions
{
public static IServiceCollection AddRealtimeService(
this IServiceCollection services)
{
services.AddTransient<PubnubFactory>();
return AddInternal(services,p => p.GetRequiredService<PubnubFactory>(),
ServiceLifetime.Singleton);
}
private static IServiceCollection AddInternal(this IServiceCollection collection,
```

```
Func<IServiceProvider, PubnubFactory> factoryProvider,ServiceLifetime lifetime)
{
Func<IServiceProvider, object> factoryFunc = provider =>
{
var factory = factoryProvider(provider);
return factory.CreateInstance();
};
var descriptor = new ServiceDescriptor(typeof(Pubnub),factoryFunc, lifetime);
collection.Add(descriptor);
return collection;
}
}
}
```

A decent dependable guideline when attempting to work inside the limitations of a DI framework is to ask yourself what your class needs to work. On the off chance that it needs a thing that you can't yet infuse, make something (a covering, maybe) that permits it to be injectable, and afterward, reexamine. This procedure as a rule brings about a few little covering classes yet an entirely spotless and simple-to-follow infusion way.

The key piece of work in the first code was making a lambda work that acknowledges an IServiceProvider as information and returns an article as yield. This is the industrial facility work that we go into the administration descriptor when we register the manufacturing plant. From this point forward, whenever any item requires an occasion of a Pubnub object, it will be administered through the manufacturing plant we enlisted in the line:

```
var descriptor = new ServiceDescriptor(typeof(Pubnub),factoryFunc, lifetime);
```

This descriptor shows that a solicitation for Pubnub will be fulfilled by summoning the processing plant work in the variable factoryFunc, with the given article lifetime.

10.5.4 Assembling everything

To see this in real life and ensure everything is working, we can counterfeit the yield of the administrations from Chapter 6 by physically dropping a ProximityDetectedEvent JSON string into the proximitydetected line, as appeared in the accompanying screen capture from the RabbitMQ support (Figure 10-1).

On the off chance that our nearness screen administration is running and bought in to the line when this occurs, and our group administration is running and has all the proper information in it (there are some example shell contents in the GitHub store that tell you the best way to seed the group administration with test information), at that point the closeness screen will get the occasion, enlarge it, and afterward dispatch a continuous occasion through PubNub. Utilizing the PubNub troubleshoot comfort, we can see the yield of this procedure show up quickly (actually it's practically prompt) in the wake of handling (Figure 10-2).

You can duplicate and adjust the content record in the GitHub vault that populates the group administration with test information just as an example JSON document containing a test closeness occasion so you can go through this yourself without beginning any of the code.

10.5.5 Making a Real-Time Proximity Monitor UI

Having a microservice that gets nearness occasions, expands them, and afterward sends them out for dispatch to our constant informing framework is extraordinary, however now, we haven't done anything important with the continuous message.

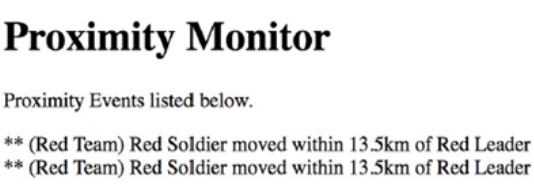
As referenced before, we could utilize this message to move pushpins on a guide UI, we could powerfully refresh tables or outlines, or we could essentially make little toasts or spring up warnings in a web UI. Contingent upon our informing supplier, we could likewise have these messages consequently changed over into pop-up messages and sent straightforwardly to colleagues' cell phones.

Example 10-6. realtimetest.html

```
<html>
<head>
<title>RT page sample</title>
<script src="https://cdn.pubnub.com/sdk/javascript/pubnub.4.4.0.js">
</script>
<script>
var pubnub = new PubNub({
subscribeKey: "yoursubkey",
publishKey: "yourprivatekey",
ssl: true
});
```

```
pubnub.addListener({
message: function(m) {
// handle message
var channelName = m.channel;
var channelGroup = m.subscription;
var pubTT = m.timetoken;
var msg = JSON.parse(m.message);
console.log(“New Message!!”, msg);
var newDiv = document.createElement(‘div’)
var newStr = “** (“ + msg.TeamName + “) “ +
msg.SourceMemberName + “ moved within “ +
msg.MemberDistance + “km of “ + msg.TargetMemberName;
newDiv.innerHTML = newStr var oldDiv = document.getElementById(‘chatLog’)
oldDiv.appendChild(newDiv)
},
presence: function(p) {
// handle presence
},
status: function(s) {
// handle status
}
});
console.log(“Subscribing..”);
pubnub.subscribe({channels: [‘proximityevents’]});
</script>
</head>
<body>
<h1>Proximity Monitor</h1>
<p>Proximity Events listed below.</p>
<div id=”chatLog”>
</div>
</body>
</html>
```

Figure 10-3. Receiving real-time messages via JavaScript



Here we have a HTML div called chatLog. Each time we get a message from the PubNub channel proximityevents we make another div and add it as a kid. This new div has the name of the group just as the names of the source and target individuals, as appeared in Figure 10-3.

It merits bringing up that you don't have to have this document on a server; you can open it in any program and the JavaScript just runs. At the point when you take a gander at the documentation for PubNub's different SDKs (counting versatile), you'll perceive that it is so natural to accomplish continuous correspondence between backend administrations, end clients utilizing internet browsers, cell phones, and other incorporation focuses. This convenience isn't restricted to simply PubNub, either; most cloud informing suppliers (counting Amazon, Azure, and Google) have exceptionally simple to utilize SDKs and their documentation is normally rich and brimming with genuine models.

10.6 Assembling It All

We see tool to manufacture a comfort application ("Hello, world!") in Microsoft's new cross-stage advancement structure, .NET Core. From that point, you just included bundle references and technique calls to routinely advancement from a comfort application to a completely working web server fit for facilitating RESTful endpoints with the Model–View–

Controller design completely bolstered.

While I don't have to disparage the criticalness of learning sentence structure and the subtleties of which lines of code to compose and when, there is a significant exercise to learn here: code won't tackle every one of your issues. Building microservices isn't tied in with learning C#, or Java, or Go—it's about figuring out how to construct applications that flourish in flexibly scaling conditions, that don't have liking, and that can begin and stop immediately. As it were, it's tied in with building cloud native applications.

As we've advanced from section to part, we've conceded some significant conversations in administration of clarifying the subtleties. Since we're finished with the subtleties, I'd prefer to utilize this part to return to certain examples, talk about zones where we may have compromised, and even present a couple of philosophical thoughts liable to fuel discusses that may cause riots inside your improvement group.

10.7 Recognizing and Fixing Anti-Patterns

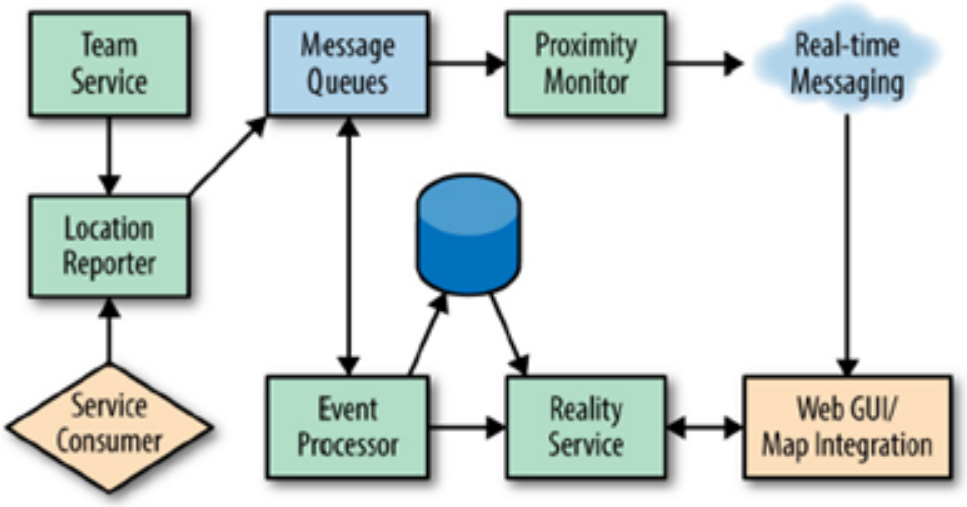
Each writer needs to walk the scarcely discernible difference between giving true examples and giving examples that are little and straightforward enough to process in the generally short vehicle of a solitary book or part.

This is the reason there are such huge numbers of "hi world" examples in books: on the grounds that else you'd have 30 pages of exposition and 1,000 pages of code postings. Parity must be struck, and bargains must be made so as to concentrate the peruser's consideration on taking care of each issue in turn.

All through the book, we've made a few trade-offs so as to keep up this equalization, yet I need to return now and return to certain thoughts and ways of thinking to assist better with advising your dynamic procedure since you've got an opportunity to construct, run, and tinker with the entirety of the code tests.

In this increasingly included situation, we start with cell phones presenting the GPS directions of colleagues to the area journalist administration. From that point, these orders are changed over into occasions with enlarged information from the

Figure10.4. Team monitoring solution with anti-patterns



group administration. The data at that point moves through the framework, in the end causing notices of vicinity occasions (colleagues who move inside the scope of one another) to show up at some purchaser confronting interfaces like a website page or cell phone.

From the outset, this looks decent, and it effectively demonstrated the code we needed to appear. Yet, on the off chance that we look somewhat nearer, we'll see that the occasion processor and the truth administration are really sharing an information store. For our example, this was a Redis reserve.

One of the standards of microservices regularly cited during the engineering and plan gatherings is "never utilize a database as a mix layer." It is a branch of the offer nothing rule. We frequently talk about this standard yet we infrequently invest enough energy examining the reasons why it's a standard.

One normal symptom of utilizing a database as a reconciliation level is that you end up with (at least two!) benefits that require a specific information structure or pattern to exist so as to work. This implies you can no longer change the fundamental information store freely, and these administrations frequently end up in a lockstep discharge rhythm as opposed to taking into account autonomous discharges as they should.

While this probably won't be an issue for Redis, numerous administrations perusing and composing similar information can regularly mess execution up because of locking or, more awful, can even reason information defilement.

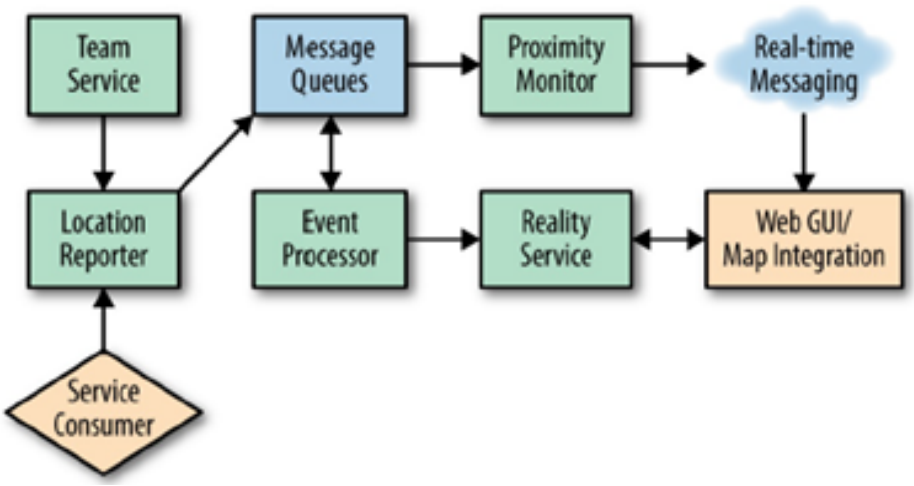
Obviously, sentiments on this fluctuate fiercely, so it's completely dependent upon you to choose whether you think this sort of sharing is suitable. For a microservices idealist such as myself, I would attempt to keep away from any engineering that firmly couples two administrations to one another, including the sharing of an information store that makes tight coupling to a real steadiness construction.

To address this issue, we can update our design as appeared in Figure 10-5.

In this new plan, the occasion processor and reality administration are not utilizing similar information store. In the old plan, the occasion processor composed the area information straightforwardly to the "truth reserve" (our Redis server). In the new plan, the occasion processor conjures the truth administration, requesting that it compose the current area.

In this engineering, the truth administration is the sole proprietor of the truth store information. This lets loose the support of progress its basic constancy component and construction at whatever point the group needs, and permits both the truth

Figure 10-5. Team monitoring solution corrected

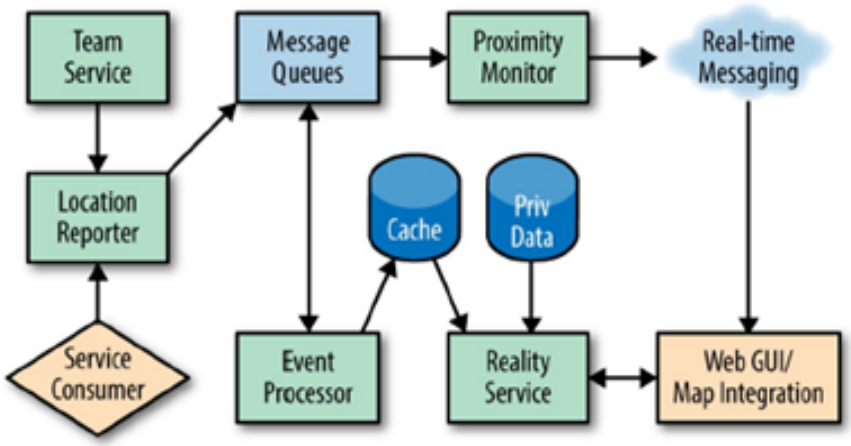


administration and the occasion processor to stay on free discharge rhythms insofar as they hold fast to best practices with regards to semantic forming of open APIs.

Another streamlining is to permit the truth administration to keep up its own private information, however to likewise keep up an outside store. The outside reserve would fit in with a notable determination that ought to be dealt with like an open API (e.g., breaking changes have downstream results). A representation of this is appeared in Figure 10-6.

We probably won't need this improvement, yet it is only one of numerous ways around the issue of utilizing an information store as an incorporation layer between administrations. There's nothing amiss with utilizing a store to give a subset of

Figure 10-6. Treating a cache as a versioned, public API



usefulness or as a streamlining, inasmuch as the mutual reserve doesn't turn into motivation to constrain distinctive improvement groups into a lockstep discharge rhythm or make discharge conditions.

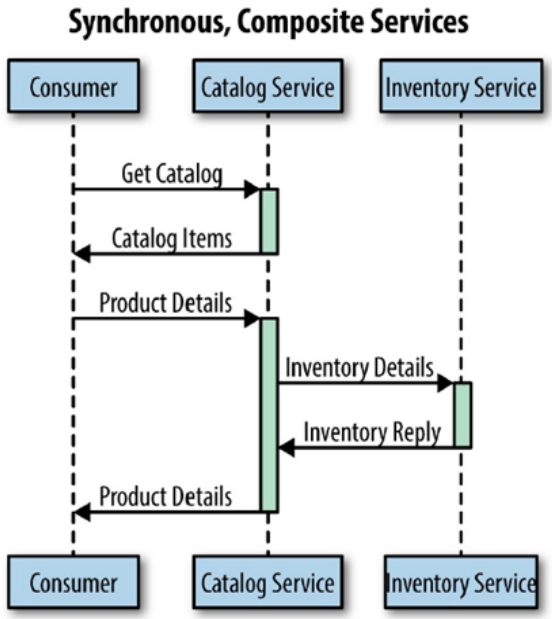
10.8 Proceeding with the Debate over Composite Microservices

Before we talk about the advantages and disadvantages of composite administrations, we ought to most likely characterize that state. A composite assistance is any help that relies upon summoning another assistance so as to fulfill a solicitation. This is quite often a coordinated call, which hinders the first call until at least one settled calls total.

We've seen this example a couple of times in this book while exhibiting different parts of ASP.NET Core. To start with, we saw the example when an early form of the group administration conjured the area administration when a guest requested subtleties on a particular colleague.

Afterward, we saw this equivalent example when talking about help disclosure and enrollment. we constructed an answer that has an information stream like the one outlined in Figure 10-7.

Figure 10-7. Synchronous, composite service usage



In this situation, a customer that demands item subtleties must pause while the index administration makes a coordinated call to the stock help to bring the stock status of a specific thing.

This is a generally basic situation, however how about we envision that this example spreads all through a venture. Assume the stock help gets adjusted a couple of months after discharge to rely upon some new assistance. That new help at that point gets split since individuals believe it's "too huge." The group that constructed the first item administration may be willfully ignorant that their one apparently innocuous, simultaneous call to stock is presently a chain six profound of coordinated calls. Without the item group busy, their normal reaction time could have gone from two or three hundred milliseconds to longer than an entire second.

More terrible, in this theoretical situation, the disappointment pace of the item administration has soar. It used to work constantly, and now customers are detailing breaks and weird server blunders. This happens in light of the fact that some place in the profoundly settled heap of coordinated calls, something fizzles and that low-level disappointment makes a course that air pockets back up to the customer.

There are microservice plan idealists who solidly accept that a genuine microservice ought to never call another assistance simultaneously. While I don't think this standard applies to constantly, we should be definitely mindful of the dangers engaged with making coordinated gets out of our administrations.

10.8.1 Alleviating Risk with Circuit Breakers

One possible approach to manage the settling of coordinated calls is to thought of a fallback instrument; an approach to manage disappointments anyplace in the call chain. The example of giving a fallback rather than either slamming or blocking inconclusively within the sight of a bombing backing administration is normally called executing an electrical switch.

A full exposition on circuit breakers and normal executions might take up a book all alone. Microsoft has an OK starting article, and you can peruse progressively about the first driving way of thinking in Martin Fowler's post. As per Fowler, the electrical switch design was initially promoted in Michael Nygard's book Release It! (Commonsense Press).

At the point when we make calls to different administrations, those calls can come up short. The explanations behind these disappointments are almost vast. The administration could return sudden information, making our procedure crash. The administration couldn't react inside a suitable time, obstructing our guests. The system could do a wide range of horrible things to our solicitation, keeping it from being taken care of.

Instead of letting these disappointments occur again and again and cause untold decimation, after some edge is crossed, the electrical switch is stumbled. When it's stumbled, we no longer endeavor to speak with the wrecked assistance, and we rather return some fitting fallback esteem. Much the same as in our homes, if a circuit comes up short out of the blue (a short, to an extreme

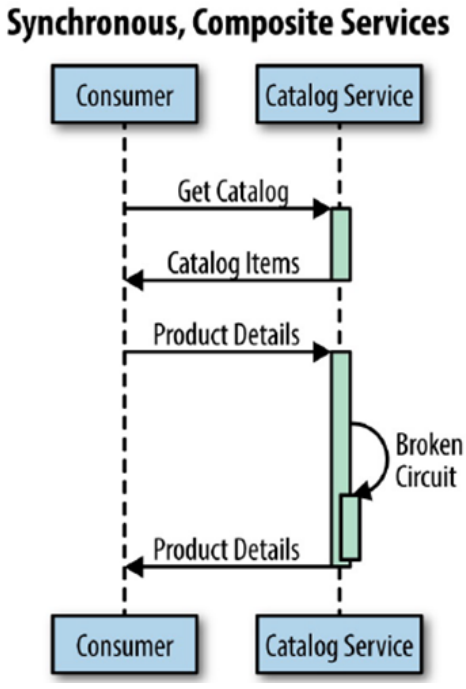
current draw, and so on.), the electrical switch outings and force is no longer provided to the bombing circuit inspired by a paranoid fear of the potential harm that could be caused.

The arrangement chart in Figure 10-8 shows the simultaneous stream when the circuit is stumbled between the index and stock administrations.

In this situation, we never endeavor to call the stock help. Instead of returning live stock information in the item subtleties, we could just return "N/A" for stock data, or some other metadata to show the disappointment.

Without this electrical switch, the stock help falling flat could totally bring down the list administration, despite the fact that the inventory administration is working appropriately. On the off chance that we recall Chapter 6 and our objective of grasping possible consistency, constructing our frameworks around the possibility that the stumbled electrical switch will in the long run return to ordinary shouldn't alarm us.

Figure 10-8. Composite service calls with a broken circuit



As Fowler delineates in his pseudocode, we typically set up an electrical switch as a covering around the customer that speaks with the sponsorship administration. It is inside this covering the condition of the circuit (open/terrible, shut/great) is kept up, just as the metadata distinguishing the conditions under which the circuit ought to be stumbled. You can see this showed in his meaning of the state variable:

def state

(@failure_count >= @failure_threshold) ? :open : :closed

End

As with such a large number of the other basic issues we experience when building microservice environments, there is a Netflix OSS answer for this. This item is called Hystrix. You can discover a review of the Hystrix item on Netflix's GitHub wiki.

Netflix's usage is just for Java, however there are a lot of libraries accessible for you to assess on the off chance that you think circuit breakers are what you need. A library worth taking a gander at is Polly. Polly gives an exquisite, familiar punctuation for proclaiming strategies for retries, breaks, circuit breakers, and the sky is the limit from there. Here's an example of Polly's explanatory electrical switch linguistic structure taken from its documentation:

Policy

.Handle<DivideByZeroException>()

.CircuitBreaker(2, TimeSpan.FromMinutes(1));

Action<Exception, TimeSpan> onBreak = (exception, timespan) =>

{ ... };

Action onReset = () => { ... };

CircuitBreakerPolicy breaker = Policy

.Handle<DivideByZeroException>()

.CircuitBreaker(2, TimeSpan.FromMinutes(1), onBreak, onReset);

In the event that Polly appears to be a little awkward or appears to take care of an excessive number of issues for you, there are some other lightweight options accessible on GitHub that can be found with a snappy quest for "C# electrical switch." The one suggestion I need to give here is that you ought to invest the main part of your energy making sense of on the off chance that you need circuit breakers, not which execution you need. Circuit breakers accompany their own additional

unpredictability and upkeep costs, and can frequently expand the measure of settled coordinated brings in a structure on the grounds that their quality can hush designers and engineers into a misguided sensation that all is well and good.

10.8.2 Disposing of the Synchronous Composite Pattern

The most significant choice to make about circuit breakers or composite administrations isn't the means by which to actualize them, yet whether we need them by any stretch of the imagination. Clearly, we don't in every case live in the place that is known for unicorns, rainbows, and perfect assistance structures. Be that as it may, on the off chance that we invest a little energy examining our issues and expected arrangements, searching for ways around normal entanglements, we may have the option to maintain a strategic distance from administration creation.

How about we investigate the model we've been utilizing: the index and stock administrations. Do we truly need to know the specific, ongoing stock status of any item whatsoever occasions? On the off chance that we investigate how every now and again that information may change, at that point we understand that we likely don't have to create these administrations the manner in which we have.

Imagine a scenario in which the stock assistance refreshed a store each time a huge change happened in the status of a thing. In this situation, the list administration doesn't have to make a simultaneous call to the stock help; it can simply question the reserve keyed side-effect ID. On the off chance that there's no store information, at that point we can attempt to call the stock help. In the event that the stock help bombs briefly, at that point the most dire outcome imaginable is the list administration will report the last known stock status. At the point when the stock help recoups, it can invigorate the store appropriately.

With this example, we don't have to actualize any retry rationale, and we don't have to work in exponential back-off surveying or utilize a heavyweight electrical switch system. Rather, we exploit the way that for this situation, the desires for buyers can be met with a less difficult, offbeat arrangement.

This won't generally be the situation, and intricacy is continually prowling around the bend. The lesson of the story here is to consistently address multifaceted nature. Each time something looks entangled, or appears like it includes a shortcoming or a basic purpose of inability to your engineering, rethink the requirements that drove you to that plan and check whether there's something more straightforward and less firmly coupled that can take care of a similar issue.