(2½ Hours)

[Total Marks: 60]

N. B.: (1) **All** questions are **compulsory**.
(2) Make **suitable assumptions** wherever necessary and **state the assumptions** made.
(3) Answers to the **same question** must be **written together**.
(4) Numbers to the **right** indicate **marks**.
(5) Draw **neat labelled diagrams** wherever **necessary**.
(6) Use of **Non-programmable** calculators is **allowed**.

I	**Choose the correct alternative and rewrite the entire sentence with the correct alternative. (30)**			
1.	Which architecture is goal-oriented?			
	a.	**Microservices**	**b.**	Microprocessor
	c.	IoT	**d.**	Azure
2.	What makes the microservices approach special?			
	a.	Maintaining existing components	**b.**	Microservice System
	c.	**Replacement of components**	**d.**	Microservice implementation
3.	Microservice applications does not share which important characteristics?			
	a.	Independently deployable	**b.**	Autonomously developed
	c.	**Centralized**	**d.**	Consistency
4.	Two microservices characteristics that might find especially concerning are______________________________.			
	a.	**decentralization and autonomy**	**b.**	Centralization and autonomy
	c.	Bounded by contexts and Centralized	**d.**	Bounded by contexts and autonomy
5.	What means that the bulk of the work done within system will no longer be managed?			
	a.	Messaging enabled	**b.**	Autonomously developed
	c.	Bounded by contexts	**d.**	**Decentralized**
6.	What reduces development time?			
	a.	Polyglotism	**b.**	**Composability**
	c.	Agility	**d.**	Comprehensibility
7.	Eliminating costly cross-team coordination challenges is indeed a significant motivation for _____________ .			
	a.	**Microservice adopters**	**b.**	Microservice architecture's
	c.	Transformation	**d.**	API client

8.	Currently the most widely deployed container toolset is _______________.			
	a.	**Docker**	**b.**	Orchestrator
	c.	Kubernete	**d.**	GitHub
9.	APIs provided by Microservices may call each other, may be called by _________________.			
	a.	networking	**b.**	internal
	c.	public-facing	**d.**	**frontend**
10.	Your organization's _________ is important because it shapes all of the atomic decisions that people within the system will make.			
	a.	Autonomy	**b.**	**Culture**
	c.	System observability	**d.**	Toolmaking
11.	The microservice designer should acknowledge that a ______________ of cultures occurs whenever outsourcing is conducted			
	a.	pollibation	**b.**	reverse data
	c.	loss of data integrity	**d.**	**cross-pollination**
12.	Bounded context should be as big as it needs to be in order to fully express its complete _______________ language			
	a.	Methodologies	**b.**	Continuous Delivery
	c.	MySQL	**d.**	**Ubiquitous**
13.	Which of the following is not the feature of .NET Core?			
	a.	**It is used for developing library projects only.**	**b.**	NET Core is the cross-platform and open-source implementation of .NET
	c.	It can run applications on multiple platforms.	**d.**	It supports modern application frameworks.
14.	A Docker registry is a place to store and distribute Docker ___________.			
	a.	Codes	**b.**	Files
	c.	Application	**d.**	**Images**
15.	________is a cross-platform web server which is included and enabled by default in ASP.NET Core project templates.			
	a.	IIS	**b.**	**Kestrel**
	c.	Apache	**d.**	Nginx
16.	Which of the Following is not an advantage of a docker?			
	a.	Simple configuration and interaction with docker composed as possible.	**b.**	Documentation provides information in detail.
	c.	**Docker provides a difficult set-up initially**	**d.**	The application lifecycle can be described in detail in Docker

17.	Select the docker image command,to list images				
	a.	docker image list	**b.**	docker ls	
	c.	**docker image ls**	**d.**	docker list	
18.	Which of the following is not true for Microservice Ecosystems?				
	a.	It is a large community of interconnected services.	**b.**	In Microservice Ecosystems, each service can have its own release cadence.	
	c.	In Microservice Ecosystems,each service can scale horizontally on demand.	**d.**	**In Microservice Ecosystems, each service cannot be deployed on its own**.	
19.	CRUD operations related to ______________________________				
	a.	Creativity, Read, Update, Delete	**b.**	Create, Read, Update, Deduce	
	c.	Create, Run, Update, Delete	**d.**	**Create, Read, Update, Delete**	
20.	Queries never __________ the database in CQRS.				
	a.	join	**b.**	**modify**	
	c.	restore	**d.**	update	
21.	The event processor is related to ________________				
	a.	Event	**b.**	Query	
	c.	Command	**d.**	**Event Sourcing**	
22.	API stands for ____________				
	a.	**Application Programming Interface**	**b.**	Analysis of Programming Interface	
	c.	Application Programming Institute	**d.**	Application Public Interface	
23.	In EF Core EF denotes to ____________________				
	a.	Entity Frame	**b.**	End Framework	
	c.	**Entity Framework**	**d.**	Empty Framework	
24.	Software built as _____________ can, by definition, be broken down into multiple component services ?				
	a.	Monolithic	**b.**	**Microservices**	
	c.	Controller	**d.**	Model	
25.	What is the meaning of OAuth?				
	a.	Original Authorization	**b.**	**Open Authorization**	
	c.	Old Authorization	**d.**	Other Authorization	

26.	Securing a microservice with no UI is called as______			
	a.	**headless**	**b.**	handfree
	c.	handheld	**d.**	handon
27.	Which of the following is also referred to as Short Message Service?			
	a.	Mini-blogs	**b.**	**Micro-blogs**
	c.	Nano-blogs	**d.**	big-blogs
28.	Which of the following about Spring Cloud is incorrect?			
	a.	Cloud-native based development	**b.**	Microservice-based architecture
	c.	Inter-service communication	**d.**	**Service integration**
29.	Which of the following is not the event of WebSocket API?			
	a.	Close	**b.**	Message
	c.	**Send**	**d.**	Error
30.	In Microservice architecture , STS stands for _______________			
	a.	service token security	**b.**	**security token service**
	c.	security to service	**d.**	service to security

II	**Attempt *any one* of the following:**		**6**
	a)	**In the Microservices Value Proposition, How Microservices can be used for deriving business value?** Explanation of the features of Miceroservice such as, **Delivery speed Benifiets:** *Agility, Composability, Comprehensibility, Independent deployability, Organizational alignment, Polyglotism* ***IThe Safety Aligned Benifets:*** Greater *efficiency, Independent manageability, Replaceability,* Stronger *resilience*, Better *runtime scalability, testability*	
	b)	**Define the concepts in regards with Microservice Design: *Organization, Culture and Embracing Change*.** **Organization** : From a microservice system perspective, organizational design includes the structure, direction of authority, granularity, and composition of teams. Many of the companies that have had success with microservice architecture point to their organizational design as a key ingredient.(Explanation and example) **Culture**: Of all the microservice system domains, culture is perhaps the most intangible yet may also be the most important. We can broadly define culture as a set of values, beliefs, or ideals that are shared by all of the workers within an organization. Your organization’s culture is important because it shapes all of the atomic decisions that people within the system will make. This large scope of influence is what makes it such a powerful tool in your system design endeavor.	

Much like organizational design, culture is a context-sensitive feature of your system. What works in Japan may not work in the United States and what works in a large insurance firm may not work at an ecommerce company. So, you'll need to be cautious when attempting to emulate the practices that work in a company whose culture you admire. There is no recipe or playbook that will guarantee you the same results.

Embracing Change : Time is an essential element of a microservice system and failing to account for it is a grave mistake. All of the decisions you make about the organization, culture, processes, services, and solutions should be rooted in the notion that change is inevitable.
You cannot afford to be purely deterministic in your system design; instead, you should design adaptability into the system as a feature**.**
A good microservice designer understands the need for adaptability and endeavors to continually improve the system instead of working to simply produce a solution.

c) **Brief about goals for the Microservices Way.**
In building applications in the microservices way: *finding the right harmony of speed and safety at scale is the major goal*
Four Goals to Consider:
1. Reduce Cost: Will this reduce overall cost of designing, implementing, and maintaining
IT services? 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. (example)
2. Increase Release Speed: Will this increase the speed at which my team can get from idea to deployment of services? Increasing the speed of the "from design to deploy" cycle is another common goal. A more useful way to view this goal is that you want to shorten the time between idea and deployment. (example)
3. Improve Resilience: Will this improve the resilience of our service network? No matter the speed or cost of solutions, it is also important to build systems that can "stand up" to unexpected failures. In other words, systems that don't crash, even when errors occur. (example)
4. Enable Visibility: Does this help me better see what is going on in my service network? Another key goal should be to enable runtime visibility. In other words, improve the ability of stakeholders to see and understand what is going on in the system. There is a good set of tools for enabling visibility during the coding process. (example)
Trade-offs : Each of these are important goals and sometimes they are competing goals. There are trade-offs to consider. You might be able to reduce your overall costs, but it might adversely affect runtime resilience.

2	Attempt ***any one*** of the following:		6
	a)	**Brief about API Design for Microservices.** When considering microservice component boundaries, the source code itself is only part of our concern. Microservice components only become valuable when they can communicate with other components in the system. They each have an interface or API. Just as we need to achieve a high level of separation, independence, and modularity of our code we need to make sure that our APIs, the component interfaces, are also loosely coupled. Otherwise, we won't be able to deploy two microservices independently, which is one of our primary goals in order to balance speed and safety. We see two practices in crafting APIs for microservices worth mentioning here: • Message-oriented • Hypermedia-driven **Messsage-Oriented** Just as we work to write component code that can be safely refactored over time, we need to apply the same efforts to the shared interfaces between components. The most effective way to do this is to adopt a message-oriented implementation for microservice APIs. The notion of messaging as a way to share information between components dates back to the initial ideas about how object-oriented programming would work. **Hypermedia-Driven** In these instances, the messages passed between components contain more than just data. The messages also contain descriptions of possible actions (e.g., links and forms). Now, not just the data is loosely coupled—so are the actions. For example, Amazon's API Gateway and App- Stream APIs both support responses in the Hypertext Application Language (HAL) format. Hypermedia-style APIs embrace evolvability and loose coupling as the core values of the design style. You may also know this style as APIs with Hypermedia As The Engine Of Application State (HATEOAS APIs). Regardless of the name used, if we are to design proper APIs in microservice architecture, it helps to get familiar with the hypermedia style. Hypermedia style is essentially how HTML works for the browser. HTTP messages are sent to an IP address and a port number (usually "80" or "443"). The messages contain the data and actions encoded in HTML format.	
	b)	**Explain Asynchronous Message-Passing and Microservices.** Asynchronous message-passing plays a significant role in keeping things loosely coupled in a microservice architecture. You probably noticed that in one of the examples earlier in this chapter, we used a message broker to deliver event notifications from our Shipment Management microservice to the Shipment Tracking microservice in an asynchronous manner. That said, letting microservices directly interact with message brokers (such as RabbitMQ, etc.) is rarely a good idea. If two microservices are directly communicating via a message-queue channel, they are sharing a data space (the channel) and we have already talked, at length, about the evils of two microservices sharing a data space. Instead, what we can do is encapsulate message-passing behind an	

independent microservice that can provide message-passing capability, in a loosely coupled way, to all interested microservices.

The message-passing workflow we are most interested in, in the context of microservice architecture, is a simple publish/subscribe workflow. How do we express it as an HTTP API/microservice in a standard way? We recommend basing such a workflow on an existing standard, such as PubSubHubbub. Now to be fair, PubSubHubbub wasn't created for APIs or hypermedia APIs, it was created for RSS and Atom feeds in the blogging context. That said, we can adapt it relatively well to serve a hypermedia API-enabled workflow.

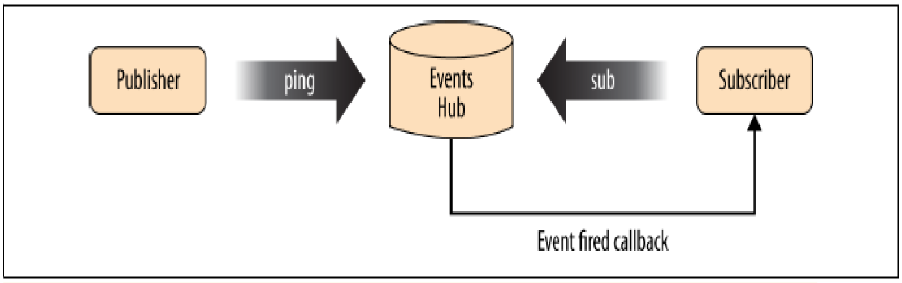

Figure 5-5. Asynchronous message-passing implemented with a PubSubHubbub-inspired flow

c) **Explain in detail about The Need for an API Gateway.**

A common pattern observed in virtually all microservice implementations is teams securing API endpoints, provided by microservices, with an API gateway. Modern API gateways provide an additional, critical feature required by microservices: transformation

and orchestration. Last but not least, in most mature implementations, API gateways cooperate with service discovery tools to route requests from the clients of microservices.

Security

Microservice architecture is an architecture with a significantly high degree of freedom. Or in other words, there are a lot more moving parts than in a monolithic application. Things can go horribly wrong securitywise when there are many moving parts. We certainly need some law and order to keep everything in control and safe. Which is why, in virtually all microservice implementations, we see API endpoints provided by various microservices secured using a capable API gateway.

APIs provided by microservices may call each other, may be called by "frontend," i.e., public-facing APIs, or they may be directly called by API clients such as mobile applications, web applications, and partner systems.

Transformation and Orchestration

Basically, to make microservices useful, we need an orchestration framework like Unix piping, but one geared to web APIs.

Microservices, due to their narrow specialization and typically small size, are very useful deployment units for the teams producing them. That said, they may or may not be as convenient for consumption, depending on the client. The Web is a distributed system.

If the team is building this system using a microservice architecture, they could end up creating two microservices for the main functionality:

1. Recommendations microservice, which takes user information in, and responds with the list containing the recommendations—i.e., suggested stock levels for various products that this customer typically ships.
2. Product Metadata microservice, which takes in an ID of a product type and retrieves all kinds of useful metadata about it.

Routing

Service discovery systems such as Consul and etcd will monitor your microservice instances and track metadata about what IPs and ports each one of your microservices is available at, at any given time.

Monitoring and Alerting

The same tools that we mentioned for service discovery can also provide powerful monitoring and failover capabilities. Let’s take Consul as an example. Not only does Consul know how many active containers exist for a specific service, marking a service broken if that number is zero, but Consul also allows us to deploy customized health-check monitors for any service.

3	**Attempt *any one* of the following:**	6

a) **Write short note on adding Startup Class and Middleware.**

Adding Startup Class

In classic ASP.NET, we had a *global.asax.cs* file that we could use to accomplish work during the various startup phases of the application. With ASP.NET Core, we can use the UseStartup<> generic method to define a startup class that handles the new startup hooks.

The startup class is expected to be able to support the following methods:

- A *constructor* that takes an IHostingEnvironment variable
- The Configure method, used to configure the HTTP request pipeline and the application
- The ConfigureServices method, used to add scoped services to the system to be made available via dependency injection

Adding Middleware

ASP.NET Core middleware components (request processors) are set up as a chain or pipeline and are given a chance to perform their processing in sequence during each request. It is the responsibility of the middleware component to invoke the next component in the sequence or terminate the pipeline if appropriate.

Middleware components can be added to request processing using the following three methods:

- **Map**

Map adds the capability to branch a request pipeline by *mapping* a specific request path to a handler. You can also get even more powerful functionality with the MapWhen method that supports predicate-based branching.

- **Use**

Use adds a middleware component to the pipeline. The component's code must decide whether to terminate or continue the pipeline.

- **Run**

The first middleware component added to the pipeline via Run will terminate the pipeline. A component added via Use that doesn't invoke the next component is identical to Run, and will terminate the pipeline.

b) Explain in details CoreCLR and CoreFX.

1 The Core CLR:

The CoreCLR is a lightweight, cross-platform runtime that provides many of the same features that the Common Language Runtime (CLR) provides on the Windows desktop or server, including:

Garbage collection: A garbage collector is responsible for the cleanup of unused object references in a managed application. If you've used any of the previous versions of .NET (or Java), then you should be familiar with the concept. Despite the differences between the CLR and CoreCLR, they both follow the same fundamental principles when it comes to garbage collection.

JIT compilation: As with previous versions of .NET, the Just-in-Time (JIT) compiler is responsible for compiling the Intermediate Language (IL) code in the .NET assemblies into native code on demand. This holds true now for Windows, Linux, and macOS.

Exception handling: For a number of reasons beyond the scope of this book, exception handling (e.g., try/catch statements) is a part of the runtime and not the base class library.

2 CoreFX:

CoreFX is a set of modular assemblies (available as NuGet packages and completely open source, available on GitHub) from which you can pick and choose. Your application no longer needs to have every single class library assembly installed on the target server. With CoreFX, you can use only what you need, and in true cloud-native fashion you should vendor (bundle) those dependencies with your application and expect nothing of your target deployment environment. The burden of dependency management is now reversed—the server should have nothing to do with it. Building .NET applications is no longer about closed-source, vendor-locked development on Windows.

	c)	**What are the Strategies for Sharing Models between Services?** There are a few things that are required for an environment to be considered a microservice ecosystem. The first, obviously, is that you need more than one service. The second is that the services within this ecosystem communicate with each other. Without the latter, you're just standing up an array of isolated and unrelated services. If we're being diligent about following some cloud-native best practices like API First, then all of our services will have documented, versioned, well understood public APIs. We might be using a YAML standard like Swagger to document our APIs, or we could be using one based on Markdown, like API Blueprint. The mechanism of defining and documenting our APIs is not nearly as important as the discipline we put into designing our APIs before we write our code. With a well-defined, versioned API that we know isn't going to break out from underneath us, the services within our ecosystem can be built by different teams. Consuming the API from those services then becomes merely a matter of writing simple REST clients. Teams frequently make some architectural decisions early on during a project that won't cause trouble until far into the future, when the cost of untangling the mess can get exorbitant. When one service changes the model in order to accommodate what should be an internal concern, the other service is affected and potentially has builds and tests broken as a result. They've lost the flexibility of true independence, and instead of being a source of flexibility, the canonical model is now a source of tight coupling and is preventing independent team deployment schedules. A microservice is an embodiment of the Single Responsibility Principle (SRP) and the Liskov Substitution Principle (LSP). A change to one service should never have any impact on any other service. A change to the internal model should be possible without corrupting the service's public API or any external models.	
4	**Attempt *any one* of the following:**		**6**
	a)	**Explain the concept of Event Sourcing. How it is defined?** To help explain how Event Sourcing works, we'll use an analogy: reality itself. 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 (our minds compensate for the fact that we process audio and visual stimuli at different speeds, giving us the illusion of synchronized stimuli). 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. This is a very different model than microliths that just expose simple, synchronous query and store–type operations.

Event Sourcing Defined

There are a number of extremely good sources of information available on Event Sourcing. ES is not a brand new pattern. It is, however, gaining new traction as a viable way to deal with the types of elastic scaling and reliability that are required by cloud services. Event Sourcing takes care of that problem, and much more, by separating the concern of state management from the concern of receiving stimuli that result in state changes. To make this happen, there are a number of requirements for an event-sourced system. It must be outlined in the following list:

Ordered : Event streams are ordered. Performing calculations against the same set of events but in a different sequence will produce different output. For this reason, ordering and proper time management are essential.

Idempotent: Any function that operates on an event stream must always return the exact same result for identical ordered event streams. This rule is absolutely mandatory, and failing to abide by it will cause untold levels of disaster.

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.

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.

b) What is Event Processor? What procedure to follow in order to keep the code clean and testable?

To detect team members within some range of each other is a hefty work. The bulk of this work is done by 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 (discussed next).

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	
	c)	**What are Backing Services?** 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** Statically bound resources are the ones we've been using in all of our sample code up to this point. While we've been careful 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*. While this certainly satisfies the external configuration requirement for cloud-native applications, it might not be flexible enough for your needs. Maybe you want something a little more dynamic and powerful. **Dynamically bound resources** A dynamically bound resource is one where the binding occurs at *runtime*. Moreover, this binding is not fixed and can actually change at runtime between requests to the application. In addition to freeing up the developers of the application from a little bit of complexity, it also allows for even looser coupling. This dynamic, loose runtime coupling between apps and bound resources facilitates more advanced functionality like failover, load balancing, and fault tolerance—all with no visible impact to the application code.	
5	**Attempt *any one* of the following:**		**6**
	a)	**What are the factors required for configuring Microservice ecosystem?** Configuration in a microservice ecosystem requires attention to a number of other factors, including: • Securing read and write access to configuration values • Ensuring that an audit trail of value changes is available • Resilience and reliability of the source of configuration information • Support for large and complex configuration information likely too burdensome to cram into a handful of environment variables	

		• Determining whether your application needs to respond to live updates or real-time changes in configuration values, and if so, how to provision for that • Ability to support things like feature flags and complex hierarchies of settings • Possibly supporting the storage and retrieval of secure (encrypted) information or the encryption keys themselves Not every team has to worry about all of these things, but this is just a hint as to the complexity of configuration management lying below the surface waiting to strike those who underestimate this problem.	
	b)	**How to configure Microservices with etcd?** The etcd is a lightweight, distributed key-value store. This is where you put the most critical information required to support a distributed system. etcd is a clustered product that uses the Raft consensus algorithm to communicate with peers. There are more than 500 projects on GitHub that rely on etcd. One of the most common use cases for etcd is the storage and retrieval of configuration information and feature flags. To get started with etcd, check out the documentation. You can install a local version of it (it really is a small-footprint server) or you can run it from a Docker image. Another option is to use a cloud-hosted version. For the sample in this chapter, I went over to compose.io and signed up for a free trial hosting of etcd (you *will* have to supply a credit card, but they won't charge you if you cancel within the trial period). To work with the key-value hierarchy in etcd that resembles a simple folder structure, you're going to need the etcdctl command-line utility. This comes for free when you install etcd. On a Mac, you can just brew install etcd and you'll have access to the tool. Check the documentation for Windows and Linux instructions. Now that you've got the alias configured and you have access to a running copy of etcd, you can issue some basic commands: **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.	

	c)	**Explain how to secure a Service with Client Credentials?** The client credentials pattern is one of the simplest ways to secure a service. First, you communicate with the service only via SSL, and second, the code consuming the service is responsible for transmitting credentials. These credentials are usually just called a username and password, or, more appropriate for scenarios that don't involve human interaction, a client key and a client secret. Any time you're looking at a public API hosted in the cloud that requires you to supply a client key and secret, you're looking at an implementation of the client credentials pattern. It is also fairly common to see the client key and secret transmitted in the form of custom HTTP headers that begin with the X- prefix; e.g., X-MyApp-ClientSecret and X-MyApp-ClientKey. The code to implement this kind of security is actually pretty simple, so we'll skip the sample here. There are, however, a number of downsides to this solution that stem from its simplicity. For example, what do you do if a particular client starts abusing the system? Can you disable its access? What if a set of clients appear to be attempting a denial of service attack? Can you block all of them? Perhaps the scariest scenario is this: what happens if a client secret and key is compromised and the consumer gains access to confidential information *without* triggering any behavioral alerts that might get them banned? What we need is something that combines the simplicity of portable credentials that do not require communication with a third party for validation with some of the more practical security features of OpenID Connect, like validation of issuers, validation of audience (target), expiring tokens, and more.	