Deployment and Operations for Software Engineers 2nd Ed

Chapter 11—Design options

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

Communication styles

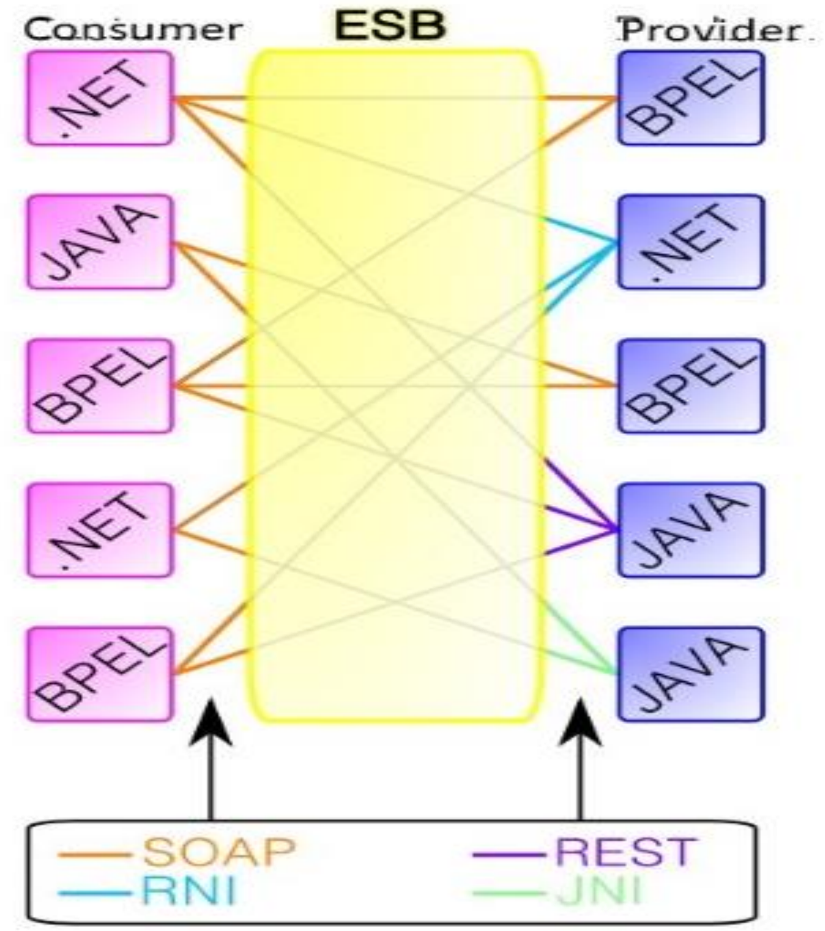
Structuring request and response data

Service Oriented Architecture

- A service-oriented architecture (SOA) has
    - A collection of distributed components that provide and/or consume services.
    - Enterprise Service Bus (ESB).
- An ESB supports
    - Discovery for consumers
    - Providers must register with the ESB
    - Communication between providers and consumers
    - Protocol translation

ESB

- Any consumer can talk to any producer and vice versa
- ESB performs protocol translation



Components in a SOA

- Providers and consumers are
  - standalone entities
  - deployed independently.
- Components have interfaces that describe the services they request from other components and the services they provide.
- Services can be implemented heterogeneously, using whatever languages and technologies are most appropriate

Example of the use of SOA

- Suppose your organization is a bank that has just acquired another bank.
- You now have two copies of loan management software, fraud detection software, and account management software.
- Each of these systems has their own user interface and process assumptions.
- Integration using SOA
  - Create a uniform database for accounts and loans.
  - Attach the new and legacy databases to the ESB
  - attach legacy systems to the ESB.
  - Use the ESB to translate from the original bank specific formats to and from the new uniform database format.

Discussion questions

1. It is possible that components in the SOA are managed by the different organizations. What additional elements are required if the producer and comsumer components are managed by different organizations.
2. Develop a use case for SOA that is different from the banking example.

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

Communication styles

Structuring request and response data

Microservice architecture

- Around 2002 Amazon promulgated the following rules for their developers.
  - All teams will henceforth expose their data and functionality through service interfaces.
  - Teams [software] must communicate with each other through these interfaces.
  - There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no backdoors whatsoever. The only communication allowed is via service interface calls over the network.
  - It doesn't matter what technology they [services] use.

Microservice services

- The basic packaging unit is a service. Services are independently deployable.
- The current location of a service must be dynamically discovered.
- Mechanisms for discovery-
  - DNS,--for VMs
  - ESB—for services in SOA
  - Service mesh—for microservices in an orchestration system

Microservice communication

- Microservices communicate only via network messages.
- Network communication is an inherent portion of a microservice architecture.
- It is not a coincidence that microservice architectures date from around 2002 since that is when cloud computing with fast network communication was possible.

Technology usage

- One common cause of integration errors is version incompatibility.
- Technology independence allows each team to independently
  - Choose implementation language
  - Choose libraries and dependencies
- No need to coordinate with other teams on any of these choices.

Microservices and teams

- Amazon has a “two-pizza” rule--Every team can be fed with two pizzas.
- In practice this rule limits team size to roughly seven people.
- If the service grows so that seven people is insufficient, the service will be split in two.

Microservice ownership

- Each microservice is owned by a single team. No coordination between teams is necessary over a microservice
- A single team may own multiple microservices, but no microservice has multiple owners.
- Limited coordination leads teams to treat other teams as they would treat outside entities--defensive programming.

Discussion questions

1. Compare microservices and SOA with respect to the ownership of the components.
2. What is the relation between microservices and APIs?

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

Communication styles

Structuring request and response data

Microservice qualities

- As microservice architecture can be analyzed in terms of its quality attributes
  - Availability,
  - Modifiability,
  - Performance
  - Reusability
  - Scalability
  - Security

Availability

- If an instance of a microservice fails, it will be detected by the load balancer which may be a portion of the orchestrator.
- The load balancer knows the instance has failed because it fails to send a health message within a specified time.
- The load balancer will not send any more requests to the failed instance
- The auto scaling capability may create a new instance to replace he failed instance.

State in microservices

- If the microservice is stateless, then a failed instance can be replaced just by creating a new one.
- If the microservice is stateful, then the state lost when the microservice failed must be recovered. The mechanisms for doing this depend on where the state is stored.

Modifiability

- Measures of modifiability are coupling and cohesion.
- Modifiability is enhanced by having low coupling among services and high cohesion within a single service.
- If a single service performs a single function
  - Cohesion is likely high.
  - Coupling will depend on the overlap between the function performed by the service and functions performed by other services.

Performance

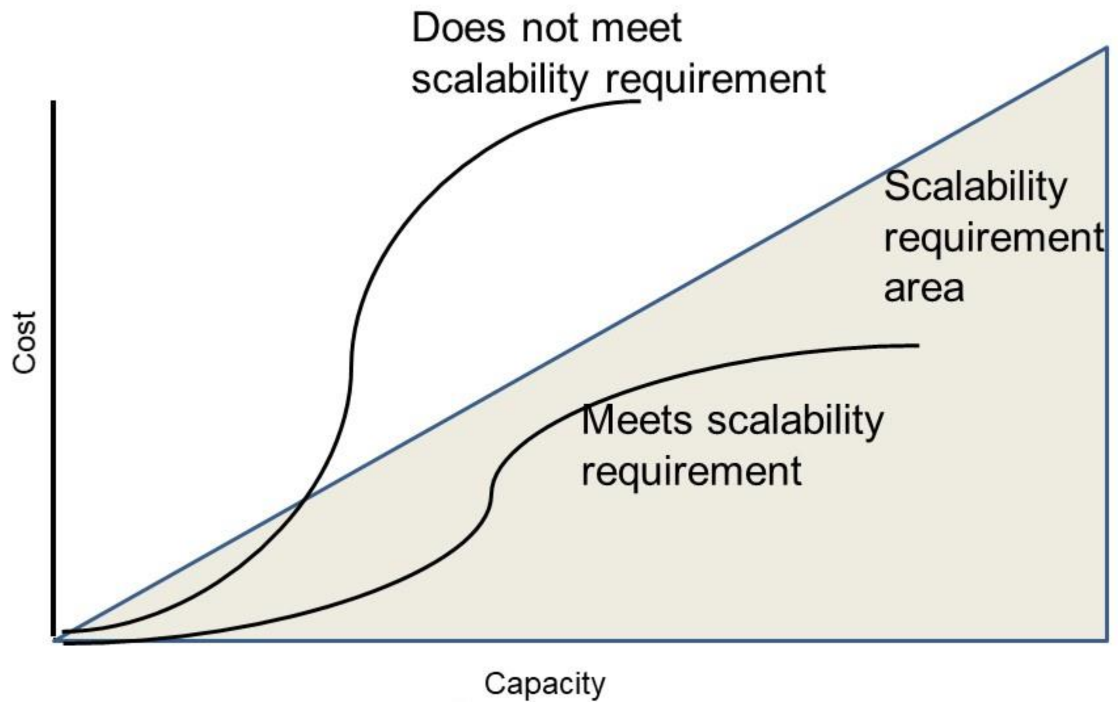
- Microservices depend crucially on network traffic
- A network message will take longer than a memory reference.
- The comparison of a system implemented as microservices to one implemented in a less distributed fashion will depend on how many messages must be sent in the microservice design.
- Techniques for improving the performance of a microservice design include
  - Caching
  - Sending messages in binary rather than in text form
  - Colocating services that have high rates of communication between them.

Reusability

- Reuse requires locating common functions and isolating them in a single service.
- This leads to low fan out among the services and more services involved in satisfying a single request.

Scalability

- A system is scalable if increasing resources results in linear or sub-linear increases in costs.
- This depends on the cloud provider’s charging model.
- Typically, adding instances of a stateless microservice will increase the costs linearly.



Security

- Some practices to increase security in a microservice architecture are
  - Use HTTPS instead of HTTP.
  - Apply patches promptly.
  - Delete unused resources promptly.
  - Do not write security-sensitive code such as password managers yourself.
  - Do not embed credentials into code or scripts.

Discussion questions

1. What is the tradeoff between performance and reusability in microservices?
2. What is the tradeoff between modifiability and performance in a microservice architecture?

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

Communication styles

Structuring request and response data

Containers and microservices

- Microservices communicate only with messages, and containers are accessible only through network interfaces that are intended for message-based communication.
- A pod is a collection of related containers that are deployed and scaled together.
  - The services that make up a PaaS or service mesh are ideal candidates to be collected in a pod with the microservice you are developing.
  - Placing the platform service resources near the microservice you are developing will reduce the latency of messages and resource usage.
- Containers provide more limited resources than VMs. Microservices are small and single purpose, and typically require fewer resources than multifunction processes.

Service mesh discovery

- Service meshes provide discovery services where the services that can be discovered are limited to those that satisfy some set of criteria.
  - Data center locality. The service should be local to the requesting microservice.
  - Another criterion can be canary or A/B testing. A canary or A/B test designates a collection of microservices as belonging to the version being tested. If the orchestrator is aware of which microservices are participating in the tests, it can populate the relevant discovery services appropriately.
  - Geographic locality. Services that interact with users can be specialized to language, to region, or other user visible attributes. Again, the orchestrator can populate the discovery service with specialized microservices.

Other service mesh microservices

- These are some microservices that can be bundled with your application microservices and discovered by the service mesh
  - Configuration
  - Distributed coordination
  - Logging
  - Tracing
  - Metrics
  - Dashboard
  - Alerts

Protecting against failure

- Recall the long tail. A request may fail because of problems somewhere in the chain of servicing the request that is out of your control.
- Some techniques to protect against long tail failures are
  - *Hedged requests*. Make more requests than you need and then cancel the requests (or ignore responses) after you receive sufficient responses. For example, if you want 10 new servers, issue 12 "launch servier" requests and cancel the 2 unsatisfied after 10 have completed.
  - A*lternative request*. Issue 10 requests, for example. When 8 requests have been satisfied, issued 2 more. After 10 total requests have been satisfied, cancel the unsatisfied requests..

Discussion questions

1. Why is configuration a service recommended for inclusion in a pod with your miroservice?
2. Is it possible to have a microservice architecture without packaging the microservices in containers? What are the tradeoffs involved in doing this?

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

C**ommunication styles**

Structuring request and response data

Communication styles

- two mechanisms for communication among components in a distributed system are common:
  - Remote Procedure Call (RPC)
  - REpresentational State Transfer (REST).

Remote Procedure Call (RPC)

- An RPC message allows code on one host to call a procedure (function or method) on another host.
- It contains four elements.
  - The IP address of the recipient
  - The name of the target procedure
  - The version number of the target procedure
  - An untyped data block that contains the request arguments or the response results.

Steps in sending an RPC message

- The basic steps involved, for most RPC implementations, are as follows.
  - Write a package definition file to define the software interface. A package definition file looks like a standard procedure declaration with parameters of specified types.
  - Run the RPC compiler to produce the stub code.
  - Link together the client modules (program, stub, RPC run time system) to make the client module.
  - Link the server modules (a standard main program, the server stub, the server routines themselves, and the RPC run time system).
- Then your code can call a procedure on another node using the IP address, procedure name, and procedure version.

Using RPC

- A remote procedure call creates a strong contract between the sender and the recipient. Both sides must agree on exactly which procedures (methods) will be implemented by the service, and on how those procedures will be numbered.
- RPC requests can be stateful or stateless.
  - A stateful request depends on previous requests made by the client to the service. E.g. you must first open a log file before you can write to it.
  - A stateless request has no such constraints.

Message transport

- One option for message transport is TCP (or TLS if you want to improve security).
- Some RPC implementations use an HTTP POST request to send the request and response.

Representational State Transfer

- RPC requires close coupling of a client to a service. They must agree on
  - Parameters
  - Procedure names
- Representational State Transfer (REST) allows looser coupling.
- REST was developed in parallel with the emergence of the World Wide Web.

Elements of REST

- Requests are stateless.
  - There is no assumption in the protocol that any information is retained from one request to the next.
  - Any state maintained is through agreement between client and server but not supported in the protocol
- Information exchanged is textual
  - services and methods are accessed by name.
  - The web was designed to be heterogeneous and hetrogenity requires textual exchange of information

REST methods

- REST restricts methods to PUT, GET, POST, and DELETE.
- REST requires that the element be self-describing by labeling it with an internet media data type (or MIME type) so that any receiver knows how to interpret the data.
- HTTP is used as the mechanism for exchanging data

REST vs RPC

- RPC favors high performance
- RPC supports a programming style that allows distributed services to be called just like local services,
- REST promotes interoperability
- REST enables rapid and independent evolution of clients and servers.
- Both are used extensively to build microservice-based systems:
  - RPC is applied to parts of the system where interactions between services are well understood or where performance is a priority,
  - REST is used in end-user-facing services and areas that are evolving faster.

Discussion questions

1. Could the World Wide Web have been developed using RPC?
2. How does the World Wide Web manage state communication between two services if REST is stateless?

Outline

Service oriented architecture

Microservice architecture

Microservice qualities

Microservices in context

Communication styles

Structuring request and response data

Message packaging protocols

- Requests and responses in both RPC and REST contain data elements that must be interpreted by both the client and the service. This data must be packaged into a message sent over the network.
- Three common protocols for packaging messages are
  - Extensible Markup language (XML)
  - Protocol Buffers
  - JSON

Extensible Markup Language (XML)

- XML adds annotations (called tags) to a textual document.
- The tags specify how to interpret the information in the document by breaking the information into chunks or fields and identifying the data type of each field.
- XML is a meta-language: It allows you to define a customized language to describe your data.
- Your customized language is defined by an *XML schema*, which is itself an XML document.

HTML and XML

- HTML is a markup language much like XML
- XML is designed for data transferred between services
- HTML is designed to describe the layout of web pages.

Protocol Buffers

- Protocol Buffers provide a specific syntax for the package definition file.
- This defines the structure of the untyped block of an RPC message.
- They are usually used in conjunction with gRPC, a binary form of RPC.
- Protocol Buffers use data types that are close to programming-language data types, making marshaling and unmarshaling efficient.

Using .proto files to support message passing between different languages

- Suppose you wish Service A (written in Java) and Service B (written in Python) to communicate.
- Place your protocol buffer specification in a .proto file.
- Language specific compilers translate the .proto file into code that marshals, unmarshals, and sends a message.
  - Compile the .proto file using the Java Protocol buffer compiler. Link it into Service A
  - Compile the .proto file using the Python Protocol Buffer compiler. Link it into Service B
- Services A and B can now communicate using the procedures generated by the two language specific .proto compilers.

Explicit schema

- XML and Protocol Buffers have explicit schemas.
- This allows checking to see if messages are specified correctly.
- If interfaces are described explicitly, they can be saved in a data base
  - Allows search to see which variables are used by which services
  - Provides up to date documentation of the architecture.

JavaScript Object Notation (JSON)

- JSON structures data as name/value pairs and array data types.
- JSON is independent of any programming language.
- Like XML, JSON is a textual representation,
- Unlike XML, JSON has no schema capability to define a valid document structure.

Discussion questions

1. XML requires reading in the entire message before marshaling can begin. Why?
2. JSON can perform marshaling as the message is read in. Why?