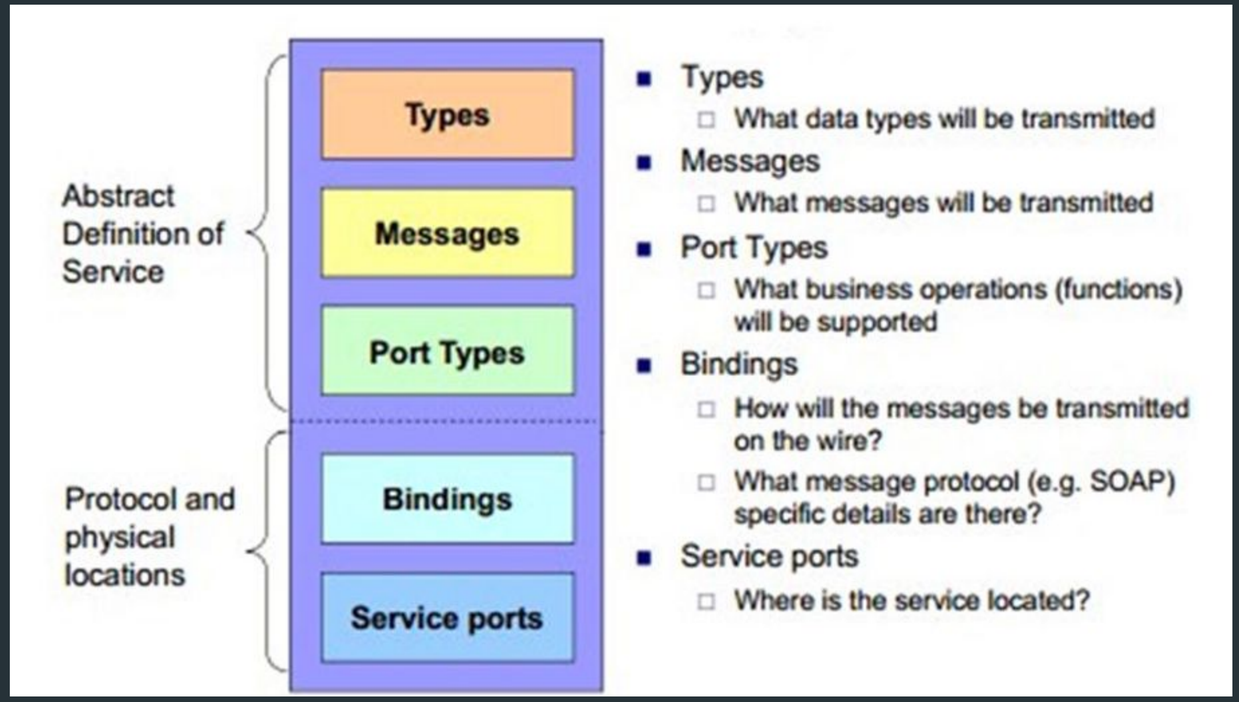
Lecture #3
WSDL
Registering and Discovering

Spring 2024

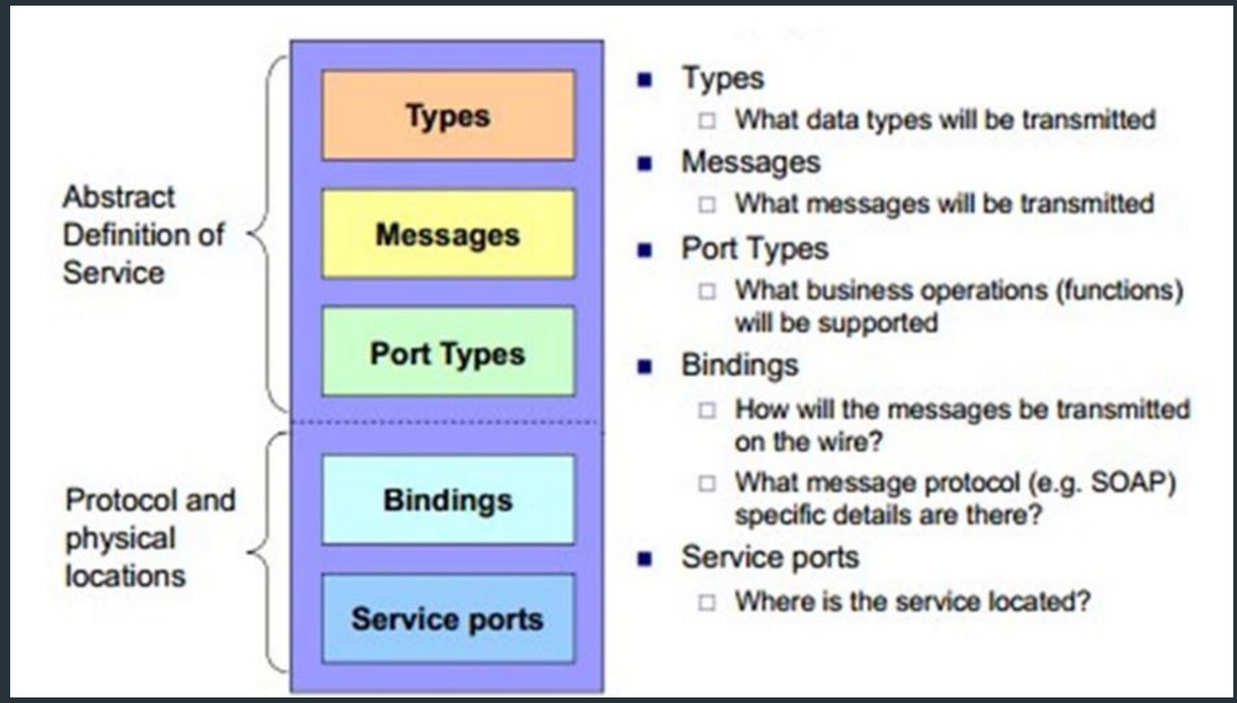
WSDL

- WSDL stands for Web Services Description Language
- WSDL is an XML-based language used for describing web services
- WSDL is used to describe the functionality offered by a web service, its inputs and outputs, and how to access it



Structure of WSDL

- WSDL is composed of three main parts:
  - Service description: describes the functionality of the web service
  - Types definition: defines the data types used in the web service
  - Message exchange patterns: describes how messages are exchanged between client and server



Example of WSDL

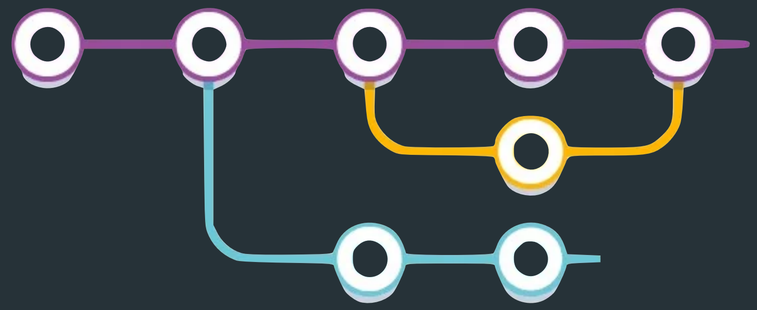
```
<wsdl:definitions name="StockQuoteService"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

<wsdl:types>
  <xs:schema targetNamespace="http://example.com/stockquote.wsdl">
    <xs:element name="GetStockPriceRequest">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="symbol" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="GetStockPriceResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="price" type="xs:decimal"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

```
    <xs:element name="GetStockPriceResponse">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="price" type="xs:decimal"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
  </xs:schema>
</wsdl:types>

<wsdl:message name="GetStockPriceRequest">
  <wsdl:part name="parameters" element="tns:GetStockPriceRequest"/>
</wsdl:message>
<wsdl:message name="GetStockPriceResponse">
  <wsdl:part name="parameters" element="tns:GetStockPriceResponse"/>
</wsdl:message>

<wsdl:portType name="StockQuotePortType">
  <wsdl:operation name="GetStockPrice">
    <wsdl:input message="tns:GetStockPriceRequest"/>
    <wsdl:output message="tns:GetStockPriceResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GetStockPrice">
```

```
  <wsdl:operation name="GetStockPrice">
    <wsdl:input message="tns:GetStockPriceRequest"/>
    <wsdl:output message="tns:GetStockPriceResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GetStockPrice">
    <soap:operation soapAction="http://example.com/GetStockPrice"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="StockQuoteService">
    <wsdl:port name="StockQuoteSoapPort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

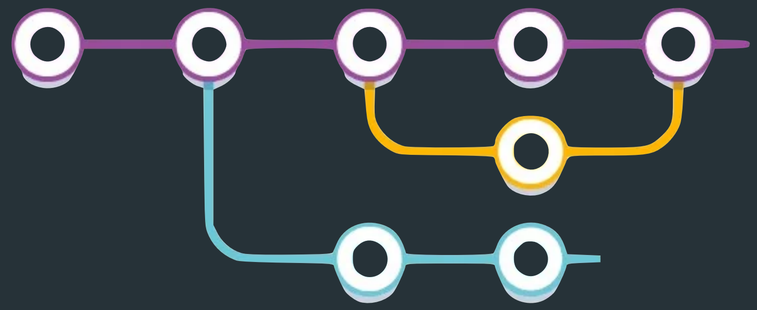
WSDL Versioning and Extensions

- WSDL allows for versioning of web services, enabling clients to use different versions of a service
- WSDL also supports extensions, allowing developers to add custom functionality to their web services
- WSDL versioning and extensions are important for maintaining compatibility and flexibility in web services



WSDL Versioning

- WSDL versioning allows for multiple versions of a web service to coexist
- A new version of a web service can be created by adding or modifying elements in the WSDL file
- Clients can use a specific version of a web service by specifying the version in the request



WSDL Extensions

- WSDL extensions allow developers to add custom functionality to their web services
- Extensions are defined using XML Schema and can be added to the WSDL file as needed
- Popular extensions include WS-Addressing, WS-Security, and WS-Policy

WS-Addressing

- WS-Addressing provides a standard way to include addressing information in SOAP messages
- It enables the identification of the intended recipient of a message and the reply address
- WS-Addressing also supports message correlation and the propagation of security and policy information

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  ...
  <wsdl:message name="MyRequest">
    <wsa:Action>http://example.org/MyService/Request</wsa:Action>
    ...
  </wsdl:message>
  <wsdl:message name="MyResponse">
    <wsa:Action>http://example.org/MyService/Response</wsa:Action>
    ...
  </wsdl:message>
  ...
</wsdl:definitions>
```

WS-Addressing

- WS-Addressing provides a standard way to include addressing information in SOAP messages
- It enables the identification of the intended recipient of a message and the reply address
- WS-Addressing also supports message correlation and the propagation of security and policy information

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  ...
  <wsdl:message name="MyRequest">
    <wsa:Action>http://example.org/MyService/Request</wsa:Action>
    ...
  </wsdl:message>
  <wsdl:message name="MyResponse">
    <wsa:Action>http://example.org/MyService/Response</wsa:Action>
    ...
  </wsdl:message>
  ...
</wsdl:definitions>
```

WS-Security

- WS-Security provides a standard way to secure SOAP messages over different transport protocols
- It supports message integrity, confidentiality, and authentication
- WS-Security also provides a framework for exchanging security tokens between service providers and consumers

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  ...
  <wsse:policy wsu:Id="ExampleServicePolicy"
               xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
      <wsse:usernameToken>
        <wsse:username/>
        <wsse:password/>
      </wsse:usernameToken>
  </wsse:policy>
  ...
</wsdl:definitions>
```

WS-Policy

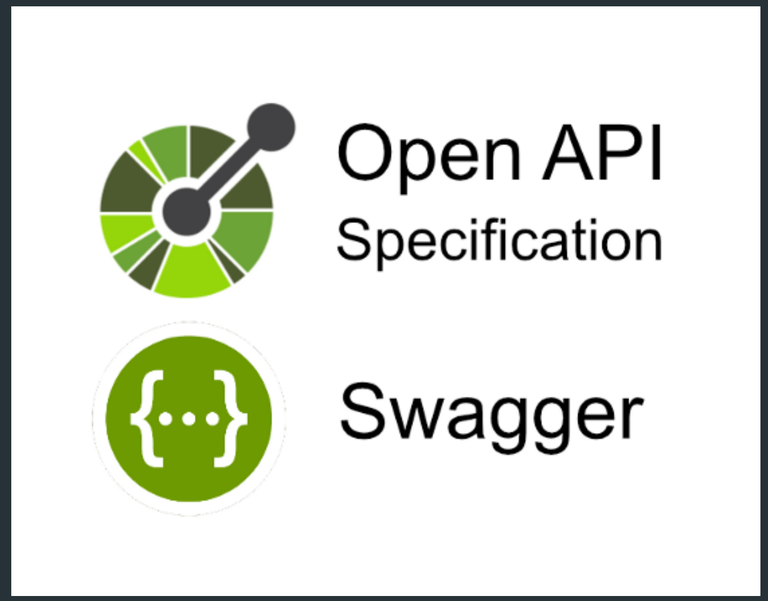
- WS-Policy provides a framework for describing the capabilities and requirements of web services
- It enables service providers to describe their policies and clients to specify their policy requirements
- WS-Policy also provides a standard way to express security requirements and preferences

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  ...
  <wsp:Policy wsu:Id="ExampleServicePolicy"
              xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
      <wsp:ExactlyOne>
        <wsp:All>
          <http:BasicAuthentication xmlns:http="http://
schemas.microsoft.com/wse/2003/06/http"/>
        </wsp:All>
      </wsp:ExactlyOne>
  </wsp:Policy>
  ...
</wsdl:definitions>
```

Comparing WSDL and OpenAPI/Swagger

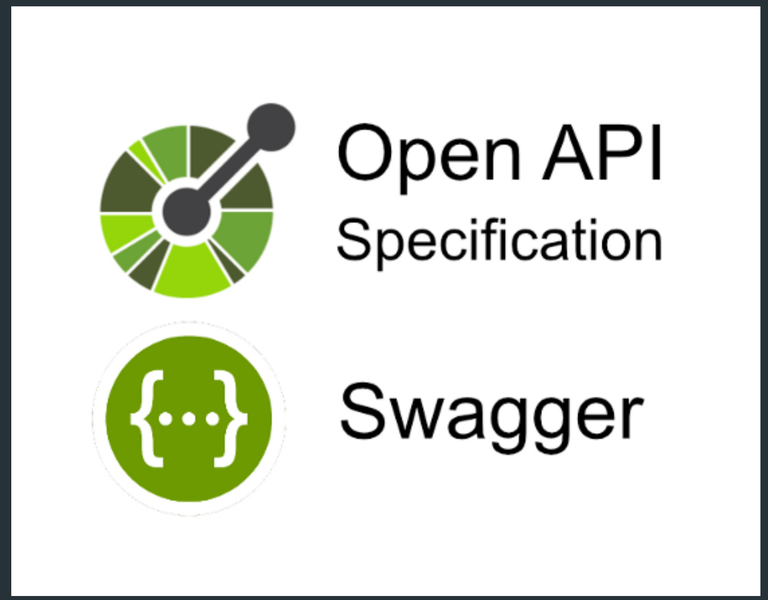
What is OpenAPI/Swagger?

- OpenAPI/Swagger is a specification for building and documenting APIs.
- It defines a standard, language-agnostic interface for RESTful APIs.
- It allows developers to generate client code, server stubs, and interactive documentation automatically.

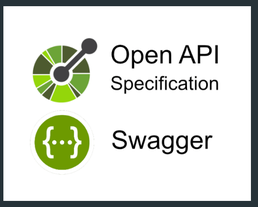


How does OpenAPI/Swagger work?

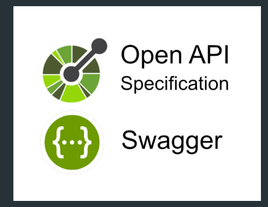
- The OpenAPI/Swagger specification is written in YAML or JSON.
- It defines the API endpoints, request parameters, response payloads, and other details.
- The specification can be used to generate client code, server stubs, and interactive documentation.



Code Example




```yaml
openapi: "3.0.0"
info:
  title: "My API"
  description: "This is a sample API"
  version: "1.0.0"
servers:
  - url: "https://api.example.com"
paths:
  /users:
    get:
      summary: "Get a list of users"
      responses:
        '200':
          description: "Successful response"
          content:
            application/json:
              schema:
                type: "array"
                items:
                  type: "object"
```




```yaml
openapi: "3.0.0"
info:
  title: "My API"
  description: "This is a sample API"
  version: "1.0.0"
servers:
  - url: "https://api.example.com"
paths:
  /users:
    get:
      summary: "Get a list of users"
      responses:
        '200':
          description: "Successful response"
          content:
            application/json:
              schema:
                type: "array"
                items:
                  type: "object"
                  properties:
                    id:
                      type: "integer"
                      description: "User ID"
                    name:
                      type: "string"
                      description: "User name"
```

Key Features of OpenAPI/Swagger

- OpenAPI specification defines a standard, language-agnostic interface for REST APIs.
- The specification is machine-readable, which means it can be easily interpreted by tools and software.
- OpenAPI provides a comprehensive documentation for APIs, including endpoints, methods, parameters, request/response structures, and error codes.

```yaml
paths:
  /pets:
    get:
      summary: Returns all pets
      responses:
        '200':
          description: A list of pets.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

Benefits of OpenAPI/Swagger

- OpenAPI makes it easier to build and maintain APIs by providing a standardized interface that is easy to use and understand.
- The specification is self-documenting, which means that it can be used to generate API documentation automatically.
- OpenAPI can be used to generate client libraries in a variety of programming languages, which makes it easier for developers to use your API.

```yaml
openapi: 3.0.0
info:
  title: Petstore API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /pets:
    get:
      summary: Returns all pets
      responses:
        '200':
          description: A list of pets.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

Benefits of OpenAPI/Swagger

- OpenAPI makes it easier to build and maintain APIs by providing a standardized interface that is easy to use and understand.
- The specification is self-documenting, which means that it can be used to generate API documentation automatically.
- OpenAPI can be used to generate client libraries in a variety of programming languages, which makes it easier for developers to use your API.

```yaml
openapi: 3.0.0
info:
  title: Petstore API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /pets:
    get:
      summary: Returns all pets
      responses:
        '200':
          description: A list of pets.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

Limitations of OpenAPI/Swagger

- OpenAPI is limited to RESTful APIs and does not support other types of APIs, such as SOAP or GraphQL.
- The specification can become overly complex for larger APIs, which can make it difficult to maintain.
- OpenAPI can be time-consuming to implement, especially for smaller APIs with limited resources.

```yaml
openapi: 3.0.0
info:
  title: Petstore API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /pets:
    get:
      summary: Returns all pets
      responses:
        '200':
          description: A list of pets.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

Drawbacks of OpenAPI/Swagger

- OpenAPI is limited to describing the structure and behavior of APIs and does not provide guidance on best practices or design patterns.
- The specification can be restrictive and may not support all of the features or behaviors of your API.
- OpenAPI does not provide any mechanism for testing or validating APIs, which can make it difficult to ensure compliance with the specification.

```yaml
swagger: '2.0'
info:
  version: 1.0.0
  title: Swagger Petstore
  description: A sample API that uses a petstore
    as an example to demonstrate features
    in the swagger-2.0 specification
host: petstore.swagger.io
basePath: /v2
schemes:
  - http
paths:
  /pet/{petId}:
    get:
      summary: Find pet by ID
      description: Returns a single pet
      operationId: getPetById
      produces:
        - application/json
      parameters:
        - name: petId
          in: path
          description: ID of pet
```

Syntax Comparison: WSDL vs OpenAPI/Swagger

- WSDL uses XML to define services, messages, operations, and endpoints.
- OpenAPI/Swagger uses YAML or JSON to define API paths, operations, parameters, and responses.
- WSDL requires separate files for each endpoint, while OpenAPI/Swagger defines all endpoints in a single file.

```xml
<definitions name="MyService" targetNamespace="http://example.com/myservice.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://example.com/myservice.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="RequestMessage">
    <part name="param" type="xsd:string"/>
  </message>

  <message name="ResponseMessage">
    <part name="result" type="xsd:string"/>
  </message>

  <portType name="MyPortType">
    <operation name="MyOperation">
      <input message="tns:RequestMessage"/>
      <output message="tns:ResponseMessage"/>
    </operation>
  </portType>

  <binding name="MyBinding" type="tns:MyPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="MyOperation">
      <soap:operation soapAction="http://example.com/MyService/MyOperation" style="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="MyService">
    <port name="MyPort" binding="tns:MyBinding">
      <soap:address location="http://example.com/myservice"/>
    </port>
  </service>

</definitions>
```

Syntax Comparison: WSDL vs. OpenAPI/Swagger

- WSDL uses SOAP for communication and data exchange, while OpenAPI/Swagger uses a variety of formats, including JSON and XML.
- WSDL defines both the message structure and the transport protocol, while OpenAPI/Swagger only defines the message structure.
- WSDL allows for more fine-grained control over message and operation definitions, while OpenAPI/Swagger has a simpler syntax for defining API operations.

```yaml
openapi: 3.0.0
info:
  title: Petstore API
  version: 1.0.0
servers:
  - url: https://api.example.com/v1
paths:
  /pets:
    get:
      summary: Returns all pets
      responses:
        '200':
          description: A list of pets.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

Syntax Comparison: WSDL vs. OpenAPI/Swagger

- WSDL supports more complex types and operations than OpenAPI/Swagger.
- OpenAPI/Swagger supports more modern API technologies, such as REST and JSON, while WSDL is primarily used with SOAP.
- WSDL has been around for longer and has a larger developer community, while OpenAPI/Swagger is more lightweight and easier to use for smaller projects.

```
<xs:element name="getStockPriceRequest">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="stockSymbol" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="getStockPriceResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

OpenAPI/Swagger Conclusion

- OpenAPI/Swagger is a powerful tool for building and documenting REST APIs.
- The specification provides a standardized, machine-readable interface that is easy to use and understand.
- OpenAPI/Swagger can help to increase adoption and usage of your API, which can lead to increased business opportunities.

```yaml
swagger: '2.0'
info:
  version: 1.0.0
  title: Swagger Petstore
  description: A sample API that uses a
      petstore as an example to demonstrate
      features in the swagger-2.0 specification
host: petstore.swagger.io
basePath: /v2
schemes:
  - http
paths:
  /pet/{petId}:
    get:
      summary: Find pet by ID
      description: Returns a single pet
      operationId: getPetById
      produces:
        - application/json
      parameters:
        - name: petId
          in: path
          description: ID of pet
```

Non-Functional Descriptions in WSDL

Non-Functional Descriptions in WSDL

- WSDL includes non-functional descriptions, which provide additional information about the service that cannot be conveyed through the functional descriptions alone.
- Non-functional descriptions in WSDL include information such as security requirements, quality of service parameters, and endpoint information.
- These non-functional descriptions are critical to ensuring that the service can be properly consumed and managed by clients and other applications.

```
<wsdl:service name="StockQuoteService">
  <wsdl:port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/stockquote"/>
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsa:EndpointReference>
      <wsa:Address>http://example.com/stockquote</wsa:Address>
      <wsa:ReferenceParameters>
        <wsa:Metadata>
          <wsdl:service name="StockQuoteService">
            <wsdl:port name="StockQuotePort">
              <wsdlsoap:address location="http://example.com/stockquote"/>
              <wsd:UsingAddressing xmlns:wsd="http://www.w3.org/2006/05/addressing/wsdl"/>
            </wsdl:port>
          </wsdl:service>
        </wsa:Metadata>
      </wsa:ReferenceParameters>
    </wsa:EndpointReference>
  </wsdl:port>
</wsdl:service>
```

Non-Functional Descriptions in WSDL

- Performance can be specified using WS-Policy, which provides a standard way of specifying performance requirements such as response time and throughput.
- WS-Addressing can be used to specify message addressing requirements, such as the source and destination of messages.
- However, the use of these non-functional descriptions can add complexity to WSDL and may require additional tools or frameworks to implement.

```
<wsdl:binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <wsp:PolicyReference URI="#StockQuoteSoapBindingPolicy"/>
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="GetStockPrice">
    <soap:operation soapAction="http://example.com/GetStockPrice"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsp:PolicyReference URI="#GetStockPricePolicy"/>
  </wsdl:operation>
</wsdl:binding>
```

Registering and Discovering Web Services

Service Registries

- Service registries are a type of service discovery mechanism used to locate web services.
- Service registries provide a central location for web services to register their availability and for clients to discover available services.
- Service registries can be implemented using various technologies such as UDDI and Consul.

```
{
  "name": "my-web-service",
  "tags": ["web", "service"],
  "address": "localhost",
  "port": 8080
}
```

Service Registries

- Service registries can be implemented as standalone systems or as part of a larger service mesh infrastructure.
- Standalone service registries can be implemented using open source software such as Eureka or ZooKeeper.
- Service mesh infrastructure typically includes service registry functionality as well as other features such as traffic management and security.

```
<wsdl:service name="StockQuoteService">
  <wsdl:port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/StockQuoteService"/>
  </wsdl:port>
</wsdl:service>
```

Service Discovery

- Service discovery is the process of automatically locating web services.
- Service discovery can be accomplished using various mechanisms such as DNS, service registries, or load balancers.
- Service discovery enables dynamic routing of requests to available web services.

```
GET /api/v1/stock-quote HTTP/1.1
Host: service-discovery.example.com
```

Service Discovery

- DNS-based service discovery uses DNS records to publish service endpoints and discover available services.
- Service registries provide a central location for web services to register their availability and for clients to discover available services.
- Load balancers can be used for service discovery by routing traffic to available endpoints based on defined load balancing rules.

```
{
  "name": "my-web-service",
  "tags": ["web", "service"],
  "address": "localhost",
  "port": 8080
}
```

Service Discovery

- Service discovery can be implemented as part of a larger service mesh infrastructure.
- Service mesh infrastructure typically includes service discovery functionality as well as other features such as traffic management and security.
- Service mesh infrastructure can be implemented using open source software such as Istio or Linkerd.

```
<wsdl:service name="StockQuoteService">
  <wsdl:port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/StockQuoteService"/>
  </wsdl:port>
</wsdl:service>
```

UDDI: Universal Description, Discovery, and Integration

- UDDI is a platform-independent, XML-based registry for businesses to list their services and for clients to discover available services.
- UDDI supports a hierarchical structure of service providers, service descriptions, and service bindings.
- UDDI enables searching for services based on keywords, service categories, and location.

```
<bindingTemplate>
  <accessPoint useType="endPoint">
    http://example.com/StockQuoteService
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:uddi.org:categorization:types"
                        instanceParms="stock quote, finance"/>
  </tModelInstanceDetails>
</bindingTemplate>
```

UDDI: Universal Description, Discovery, and Integration

- UDDI provides mechanisms for service providers to manage the lifecycle of their services, including publishing, updating, and deleting services.
- UDDI includes security mechanisms such as authentication and authorization to control access to the registry.
- UDDI can be used in combination with other service discovery mechanisms such as DNS-based discovery and service meshes.

```
<bindingTemplate>
  <accessPoint useType="endPoint">
    http://example.com/StockQuoteService
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="uddi:uddi.org:categorization:types"
                        instanceParms="stock quote, finance"/>
  </tModelInstanceDetails>
  <bindingDescription>
    Provides stock quote information for US equities.
  </bindingDescription>
</bindingTemplate>
```

UDDI: Universal Description, Discovery, and Integration

- UDDI has largely been replaced by newer service discovery mechanisms such as service registries and service meshes.
- UDDI has limited adoption due to complexity, lack of standardization, and the rise of RESTful web services.
- UDDI is still used in some industries such as healthcare and government where interoperability between different systems is critical.

```
<businessService>
  <name>Payment Processing Service</name>
  <description>Provides payment processing for e-commerce sites</description>
  <bindingTemplates>
    <bindingTemplate>
      <accessPoint>http://example.com/payments</accessPoint>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

Mapping WSDL Services to UDDI

- WSDL service maps to a UDDI business entity
- Each WSDL port maps to a UDDI binding template
- A single WSDL port can map to multiple UDDI binding templates if multiple protocols are used
- UDDI tModels can be used to store additional service metadata not present in WSDL files

```
<!-- WSDL -->
<wsdl:service name="MyService">
  <wsdl:port name="MyPort" binding="tns:MyBinding">
    <soap:address location="http://example.com/myservice"/>
  </wsdl:port>
</wsdl:service>

<!-- UDDI Mapping -->
<business name="MyService">
  <bindingTemplate>
    <accessPoint useType="http" url="http://example.com/myservice"/>
  </bindingTemplate>
</business>
```

Mapping WSDL Services to UDDI

- WSDL types can be mapped to UDDI tModels
- UDDI tModels can be used to describe service metadata such as security policies, quality of service, etc.
- A WSDL type can map to multiple UDDI tModels if multiple facets are present

```
<!-- WSDL -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="MyElement" type="xs:string"/>
</xs:schema>

<!-- UDDI Mapping -->
<tModel name="MyElement">
 <categoryBag>
  <keyedReference tModelKey="uddi:uddi.org:categorization:types"
      keyName="type" keyValue="string"/>
 </categoryBag>
</tModel>
```

Mapping WSDL Services to UDDI

- Pros:
  - Enables automatic registration of web services on a UDDI registry
  - Makes it easier to discover and consume web services from a centralized registry
  - Facilitates the reuse of existing WSDL descriptions and metadata
- Cons:
  - Limited support for modern web service protocols and standards
  - Requires significant setup and configuration of both the WSDL and UDDI registry
  - Limited tooling and community support for the WSDL to UDDI mapping model.

Introduction to UDDI API

- UDDI API consists of two parts:
  - UDDI Inquiry API
  - UDDI Publish API

UDDI Inquiry API

- The UDDI Inquiry API provides read-only access to the UDDI registry.
- It is used to discover the available services and retrieve their details from the registry.
- The UDDI Inquiry API allows searching for services using different criteria such as business name, service name, or service category.

UDDI Publish API

- The UDDI Publish API provides write access to the UDDI registry.
- It is used to publish new services and update the details of existing services in the registry.
- The UDDI Publish API allows creating new businesses, services, and binding templates.

UDDI API Limitations

- The UDDI API has not gained widespread adoption due to several limitations.
- It is a complex and heavy-weight standard, making it difficult to implement and use.
- UDDI requires a centralized registry, which can become a single point of failure.

Introduction to Querying the UDDI Model

- Querying is the process of searching for and retrieving information from the UDDI registry
- The UDDI model defines a set of APIs that enable clients to query the registry
- The UDDI query language is based on XML and supports a wide range of query types and filters

UDDI Query Language Basics

```
<?xml version="1.0"?>
<find_business
xmlns="urn:uddi-org:api_v3"
xmlns:uddi="urn:uddi-org:api_v3"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <authInfo>REPLACE_WITH_AUTH_TOKEN</authInfo>
  <name>example business</name>
</find_business>
```

- This example shows a UDDI query that searches for a business with the name "example business"
- The query is sent using the find_business API, which is used to search for businesses in the registry
- The authInfo element contains the authentication token that is required to access the registry

UDDI Usage Model and Deployment Variants

- UDDI usage model can vary depending on the needs of the organization.
- There are different deployment variants of UDDI to meet the needs of different organizations.
- The deployment of UDDI can be done on a local server, a public server or in the cloud.

UDDI Deployment Variants

- Local server deployment is suitable for organizations that prefer to keep their services on-premise.
- Public server deployment is suitable for organizations that want to make their services available to the public.
- Cloud deployment is suitable for organizations that want a scalable and flexible solution.

UDDI Adoption Challenges

- Adoption of UDDI can be challenging due to its complexity.
- UDDI adoption can require significant resources and investment.
- Organizations may face challenges in integrating UDDI with their existing systems.

Addressing and Notification

Web Services and Stateful Resources

- In stateful resources, data changes over time, and the application's state changes along with it.
- Web services are inherently stateless, meaning that they don't maintain information about previous requests.
- To deal with stateful resources, developers need to use a variety of techniques, such as cookies or hidden form fields.

A shopping cart is an example of a stateful resource. In a web service, we can represent a shopping cart as an object with its own state, and each item added to the cart changes the state of the object.

Stateful vs. Stateless Web Services

- Stateless web services do not maintain any state information between requests.
- Stateless web services are easier to scale because they do not require any session management.
- Stateless web services require that all of the information needed for a request is included in that request.

A stateless web service might be used for retrieving information from a database. Each request includes all the information needed to retrieve the data, so there is no need to maintain any session information.

Techniques for Managing Stateful Resources in Web Services

- Cookies are a popular way to manage stateful resources.
- Hidden form fields can also be used to manage stateful resources.
- Developers can also use a unique identifier in each request to identify the session associated with that request.

A session token could be used to identify the session associated with a particular request. The token would be included in each request, allowing the server to associate the request with the correct session.

WS-Resource Framework Introduction

- The WS-Resource Framework is a set of specifications that provides a framework for working with stateful resources in web services.
- The framework defines the following components: WS-Resource, WS-ResourceProperties, and WS-ResourceLifetime.
- The WS-Resource Framework is intended to enable interoperable communication between different web service implementations.

WS-Resource Framework Components

- WS-Resource: A web service that provides access to a stateful resource. A WS-Resource can be accessed through a set of operations, which can create, read, update, and delete the stateful resource.
- WS-ResourceProperties: A mechanism for describing the stateful resource being accessed. The WS-ResourceProperties specification defines an XML format for describing the properties of a WS-Resource.
- WS-ResourceLifetime: A set of operations for managing the lifetime of a WS-Resource. The WS-ResourceLifetime specification defines a set of operations for creating, destroying, and renewing a WS-Resource.

Pros and Cons of WS-Resource Framework

- Pros:
  - Provides a framework for working with stateful resources in web services.
  - Enables interoperable communication between different web service implementations.
  - Offers a standardized mechanism for managing the lifetime of a WS-Resource.
- Cons:
  - The WS-Resource Framework can be complex to implement and use.
  - The framework may not be appropriate for all web service scenarios.
  - The use of WS-ResourceProperties may introduce additional overhead in web service communication.

Web Services Notification

- Web Services Notification (WSN) is a specification for publishing and subscribing to notifications in a web services environment.
- It allows service providers to send notifications to interested parties about events that have occurred.
- WSN provides a flexible and extensible framework for notification delivery that is independent of the underlying transport protocol.

WSN Features

- WSN defines a standard set of message formats for publishing and subscribing to notifications.
- It provides a flexible and extensible framework for notification delivery, allowing for different delivery modes, such as publish/subscribe and request/response.
- WSN also includes mechanisms for handling security and reliability.

WSN Example

- Here is an example of a WSN message format for publishing a notification:

```
<wsnt:Notify xmlns:wsnt="http://docs.oasis-open.org/wsn/b-2">
<wsnt:NotificationMessage
<wsnt:Topic>
/example/topic
</wsnt:Topic>
<wsnt:Message>
Notification message content.
</wsnt:Message>
</wsnt:NotificationMessage>
</wsnt:Notify>
```

- This message publishes a notification with the topic "/example/topic" and the message content "Notification message content."
- Interested parties can subscribe to this topic and receive notifications when they are published.

Web Services Eventing

- Web Services Eventing is a standard for asynchronous event notification between web services.
- It allows a service to notify another service about the occurrence of an event or state change in the first service.
- This enables a decoupled architecture, where services are not tightly coupled to each other.

```
<wsa:Action>http://example.org/events/myEvent</wsa:Action>
<wsa:MessageID>urn:uuid:abc12345-6789-0abc-defg-1234567890hi</wsa:MessageID>
<wsa:ReplyTo>
  <wsa:Address>http://example.org/services/eventConsumer</wsa:Address>
</wsa:ReplyTo>
```

How Web Services Eventing Works

- Web Services Eventing uses a combination of the WS-Eventing and WS-Addressing specifications to enable event-based communication.
- Publishers create events and send them to an event source, which is typically a message broker or a dedicated eventing service.
- Subscribers register with the event source to receive events that match their interests, typically by specifying a set of filters.
- When an event is published that matches a subscriber's filters, the event source sends the event to the subscriber.

```
EndpointReferenceType eventSource = ...; // create event source endpoint
EndpointReferenceType subscriber = ...; // create subscriber endpoint

// create WS-Eventing subscription manager
SubscriptionManager subscriptionManager = SubscriptionManager.getInstance();

// create subscription request
SubscriptionRequest subscriptionRequest = new SubscriptionRequest(eventSource, subscriber);
```

```
EndpointReferenceType eventSource = ...; // create event source endpoint
EndpointReferenceType subscriber = ...; // create subscriber endpoint

// create WS-Eventing subscription manager
SubscriptionManager subscriptionManager = SubscriptionManager.getInstance();

// create subscription request
SubscriptionRequest subscriptionRequest = new SubscriptionRequest(eventSource, subscriber);

// add filter to the subscription request
Filter filter = new Filter();
filter.addTopic("my-topic");
subscriptionRequest.setFilter(filter);

// subscribe to the event source
subscriptionManager.subscribe(subscriptionRequest);

// publish event to the event source
Event event = new Event();
event.setTopic("my-topic");
event.setMessage("Hello, world!");
subscriptionManager.publish(event);
```

Lecture outcomes

- WSDL
- UDDI
- Registry
- Addressing
- Notifications