Systems for Design and Implementation

2015-2016

Course 8

Contents

- SOAP
- Web Services
- Spring Remoting
- Protobuf

SOAP

- Simple Object Access Protocol (SOAP) is an XML based protocol designed for exchanging information (called SOAP messages) in a distributed computing environment.
- There is no concept of a central server in SOAP, all nodes can be considered equals.
- SOAP differs from RMI and Remoting in that it concentrates on the content, effectively decoupling itself from both implementation and underlying transport.
- The protocol is made up of a number of distinct parts:
  - The first is the envelope, used to describe the content of a message and some indications on how to process it.
  - The second part consists of the rules for encoding instances of custom data types.
  - The last part describes the application of the envelope and the data encoding rules.

SOAP envelope

- It is an XML document that includes all of the information necessary to process the message.
- A simple Envelope, with the namespace specified as SOAP version 1.2. It includes two sub elements, a Header and a Body:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
    <env:Header>
    </env:Header>
    <env:Body>
    </env:Body>
</env:Envelope>
```

SOAP Header

- The *Header* in a SOAP message is meant to provide information about the message itself, as opposed to information meant for the application. For example, the *Header* might include routing information:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
<env:Header>
    <wsa:ReplyTo xmlns:wsa= "http://schemas.xmlSOAP.org/ws/2004/08/addressing">
    <wsa:Address>
            http://schemas.xmlSOAP.org/ws/2004/08/addressing/role/anonymous
    </wsa:Address>
    </wsa:ReplyTo>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
</env:Header>
<env:Body>
</env:Body>
</env:Envelope>
```

SOAP Header

- The *Header* can include all kinds of information about the message itself.
- The SOAP specification makes no assumption that the message is going straight from one point to another, from client to server.
- It allows for the idea that a SOAP message might actually be processed by several intermediaries, on its way to its final destination, and the specification is very clear on how those intermediaries should treat the information that they find in the *Header*.
- The SOAP *Header* is optional. If it is present, it must be the first subelement of the Envelope element.

SOAP Body

- When you send a SOAP message, you try to tell the receiver to do something, or you try to impart information to the server. This information is called the *"payload"*. The payload goes in the *Body* of the *Envelope*.
- The payload also has its own namespace, that is completely arbitrary. It just needs to be different from the SOAP namespace.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
<env:Header>... </env:Header>
<env:Body>
    <m:GetCurrentTemperature xmlns:m="WeatherStation">
        <m:scale>Celsius</m:scale>
    </m:GetCurrentTemperature>
</env:Body>
</env:Envelope>
```

- The choice of how to structure the payload involves the style and encoding.

SOAP Body - Style and Encoding

- Two different programming styles for SOAP messages predominate.
- The first is the RPC style, based on the concept of using SOAP messages to create Remote Procedure Calls. The idea is that you are sending a command to the server, such as “get current temperature", and you are including the parameters for that command, such as the scale to use, as child elements of the overall method.
- The alternative to the RPC style involves simply having your data as the content of the SOAP body, and including the information as to what procedure or function it pertains to in the routing of the message by the application server.

Message exchange patterns

- Two different patterns for exchanging SOAP messages:
  - **Request/Response**: In the Request/Response pattern, you send a request in the form of a SOAP message and then you wait for a response back. The request may be synchronous or asynchronous.
  - **One-way messaging**: This case, also known as the "fire and forget“ method, involves sending the request and then simply forgetting about it. You can use this in situations in which you are simply imparting information, or any other situation in which you do not really care what the receiver has to say about it.

Web Services

- Distributed computing enables different applications to talk to each other, even if they are not on the same computer. Technologies such as Remoting and RMI provide a system that included a registry of sorts so that applications can find components with which they wish to interact, and then call these components as though they were located on the local machine.
- Disadvantages:
  - the need to use the same technology for both the client and the server,
  - the system cannot be accessed from outside
- Web services provides a text based (actually, XML-based) means for transferring messages back and forth, which means that the applications are not just machine independent, but also operating system and programming language independent.
- As long as both parties follow the web services standards, it doesn't matter what software is running on either side.

Web Services

- Kinds of web services:
  - SOAP-based web service involves the exchanging of SOAP based messages, usually via HTTP.
  - REST is a type of web service in which the user simply accesses a URL, and the response is a straight XML document. There is no particular format to response messages. It is just whatever the data happens to be. It is application dependent.
  - XML-RPC based web services involve the use of XML-RPC standard for exchanging messages:

```
<?xml version="1.0"?>
<methodCall>
    <methodName>CMS.getNumberOfArticles</methodName>
    <params>
    <param> <value><string>classifieds</string></value> </param>
    <param> <value><string>forsale</string></value> </param>
    </params>
</methodCall>
```

Web Services Specifications

- Web services specifications typically fall into two categories: basic web service specifications, and expanded web service specifications.
- Basis WS Specifications:
  - **SOAP**: The foundation of all SOAP-based web services, the SOAP specification details the format of the actual messages. It also details the way applications should treat certain aspects of the message, such as elements in the "header", which enable the creation of applications in which a message is passed between multiple intermediaries before reaching its final destination.
  - **WDSL**: Web Services Description Language is a specification that details a standard way to describe a SOAP-based web service, including the form the messages should take, and where they should be sent. It also details the response to such a message.
  - **UDDI**: Universal Description, Discovery and Integration specifies a registry of web services and other corporate information and is intended to provide a way to discover new web services to use.

Web Services Specifications

- Expanded WS Specifications (some):
  - **WS-Security**: This specification handles encryption and digital signatures, enabling you to create an application in which messages can't be eavesdropped.
  - **WS-Policy**: This specification expands on WS-Security, enabling you to more specifically detail how and by whom a service can be used.
  - **WS-I**: Although web services are supposed to be designed for interoperability, in reality there is enough flexibility in the specifications that interpretations between different implementations can cause problems. WS-I provides a set of standards and practices to prevent these sorts of problems, as well as standardized tests to check for problems.

SOAP based Web Services Example

- The service receives a text and returns the received text in upper case to which it appends the current date.
- Java:
  - Tomcat web container
  - Apache Axis2™ is a Web Services / SOAP / WSDL engine.
- Axis2 uses AXIOM, or the AXIs Object Model, a DOM (Document Object Model)-like structure that is based on the StAX API (Streaming API for XML).
- Methods that act as services must take as their argument an `OMElement`, which represents an XML element that happens, in this case, to be the payload of the incoming SOAP message. Unless this is an "in only" service, these methods must return an OMElement, because that becomes the payload of the return SOAP message.

Service Example

```
public class HelloAxiom {
public OMElement toUpper(OMElement element) throws XMLStreamException
   {
     element.build();
     element.detach();
     OMElement textElement = element.getFirstElement();
     String text = textElement.getText();
     String returnText =text.toUpperCase()+'  '+(new Date());
     OMFactory fac = OMAbstractFactory.getOMFactory();
     OMNamespace omNs =
      fac.createOMNamespace("http://axiom.service.hello/xsd", "tns");
     OMElement method = fac.createOMElement("toUpperResponse", omNs);
     OMElement value = fac.createOMElement(“return", omNs);
     value.addChild(fac.createOMText(value, returnText));
     method.addChild(value);
     return method;
}}
```

Service Example

```
//   META-INF/services.xml
<service name="HelloAxiomService" scope="application">
    <description>
        Simple HelloAxiom Service
    </description>
   <operation name="toUpper">
        <messageReceiver
   class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
    </operation>
    <parameter name="ServiceClass">
        ro.ubbcluj.mpp.soap.axiom.HelloAxiom
    </parameter>
</service>
```

Client Example

```
public class HelloAxiomClient {
private static EndpointReference targetEPR =new
   EndpointReference("http://localhost:8080/axis2/services/
   HelloAxiomService");
    public static OMElement toUpperPayload(String text) {
        OMFactory fac = OMAbstractFactory.getOMFactory();
        OMNamespace omNs = fac.createOMNamespace("http://
   axiom.service.hello/xsd", "tns");
        OMElement method = fac.createOMElement("toUpper", omNs);
        OMElement value = fac.createOMElement("text", omNs);
        value.addChild(fac.createOMText(value, text));
        method.addChild(value);
        return method;
    }
//continue on next slide
```

Client Example

```
public class HelloAxiomClient {
//
public static void main(String[] args) {
   try {
     OMElement toUpperPayload = toUpperPayload("maria ana");
     Options options = new Options();
     options.setTo(targetEPR);
   options.setTransportInProtocol(Constants.TRANSPORT_HTTP);
    ServiceClient sender = new ServiceClient();
    sender.setOptions(options);
    // sender.fireAndForget(somePayload);
    OMElement result = sender.sendReceive(toUpperPayload);
    String response = result.getFirstElement().getText();
    System.out.println("Text received: " + response);
   } catch (Exception e) {
            e.printStackTrace();
  }}}
```

Web Service C# Consumer

Steps for creating a Web Service Client using C#:

1. Generate a Proxy for the Web Service

```
wsdl /l:CS /protocol:SOAP http://localhost:8080/axis2/services/
   HelloAxiomService?wsdl
Result:
   HelloAxiomService.cs
```

2. Compile the Proxy as a DLL library

```
csc /t:library /r:System.Web.Services.dll /r:System.Xml.dll
   HelloAxiomService.cs
```

Web Service C# Consumer

3. Develop the Client class file

```
class Program{
   static void Main(string[] args){
     HelloAxiomService helloService = new HelloAxiomService();
     String text = "Ana are mere";
     Console.WriteLine("sending text ... " + text);
     XmlNode[] response =(XmlNode[]) helloService.toUpper(text);
     XmlNode root = response[0];
     Console.WriteLine("Text received: " + root.Value);
     }
}
```

4. Build the client assembly

```
csc Client.cs /r:HelloAxiomService.dll /r:System.Web.Services.dll /
     r:System.Xml.dll
```

Spring Remote Services

RMI is a good way to communicate with remote services, but it has some limitations:

- RMI has difficulty working across firewalls. RMI uses arbitrary ports for communication (something that firewalls typically do not allow).
- RMI is Java-based (both the client and the service must be written in Java) and uses Java serialization. The types of the objects being sent across the network must have the exact same version on both sides of the call.
- Caucho Technology has developed two remoting solutions that address the limitations of RMI: Hessian and Burlap.
- Hessian and Burlap are two solutions that enable lightweight remote services over HTTP. They each aim to simplify web services by keeping both their API and their communication protocols as simple as possible.

Hessian vs Burlap

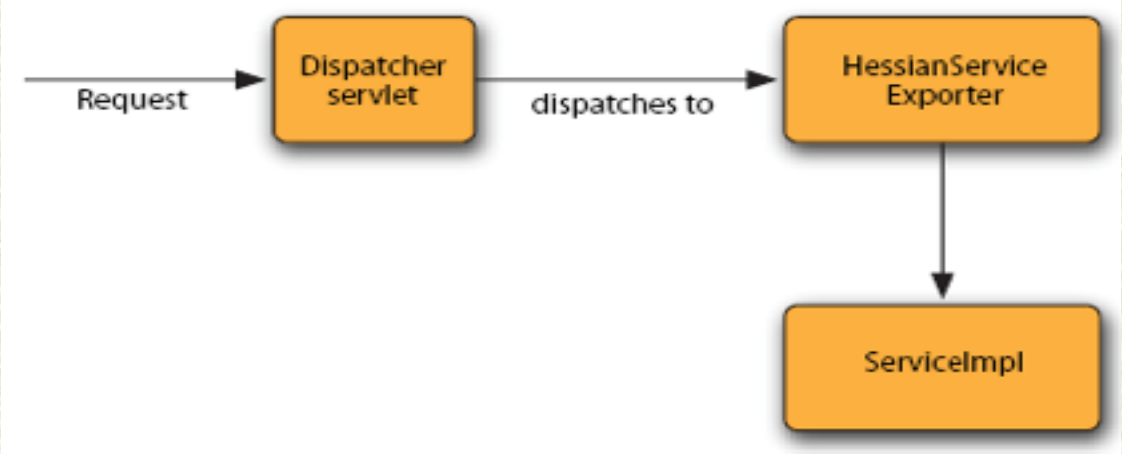
- Hessian and Burlap are two solutions for the same problem, but each serves slightly different purposes.
- Hessian, like RMI, uses binary messages to communicate between client and service. The binary message is portable to languages other than Java, including PHP, Python, C++, and C# (difference from other binary remoting technologies like).
- Burlap is an XML-based remoting technology, which automatically makes it portable to any language that can parse XML. It is more easily human readable than Hessian's binary format. Unlike other XML-based remoting technologies (like SOAP or XML-RPC), Burlap's message structure is as simple as possible and does not require an external definition language (such as WSDL or IDL).
- For the most part, they are identical. The only difference is that Hessian messages are binary (less bandwidth) and Burlap messages are XML (human readability).

Exporting a Hessian Service

- Exporting a Hessian service in Spring is similar to implementing an RMI service in Spring.
- We need to configure an exporter bean in the Spring configuration file of type `HessianServiceExporter`.
- `HessianServiceExporter` performs the same function for a Hessian service as `RmiServiceExporter` does for an RMI service: it exposes the public methods of a POJO as methods of a Hessian service.
- It is a Spring MVC controller that receives Hessian requests and translates them into method calls on the exported POJO.

Exporting a Hessian Service

```
//configuration file
 <bean id="transformerService"
   class="transformer.impl.TransformerImpl"/>
<bean id="hessianTransformerService"
   class="org.springframework.remoting.caucho.HessianServiceExporter">
      <property name="service"  ref= "transformerService"/>
      <property name="serviceInterface"
   value="transformer.services.ITransformer"/>
</bean>
```



Configuring The Hessian Controller

- Hessian is HTTP-based, so the **`HessianServiceExporter`** is implemented as a Spring MVC Controller.
- In order to use exported Hessian services, we need to perform the following configuration steps:
  - Configure a Spring **`DispatcherServlet`** in **`web.xml`** and deploy the application as a web application.

```
<servlet>
        <servlet-name>transformer</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
        <servlet-name>transformer</servlet-name>
        <url-pattern>*.service</url-pattern>
 </servlet-mapping>
```

Configuring The Hessian Controller

- Configure a URL handler in the Spring configuration file to dispatch Hessian service URLs to the appropriate Hessian service bean.

```
//configuration file transformer-servlet.xml
<bean id="transformerService" class="transformer.impl.TransformerImpl"/>
<bean id="hessianTransformerService"
   class="org.springframework.remoting.caucho.HessianServiceExporter">
            <property name="service"  ref= "transformerService"/>
            <property name="serviceInterface"
   value="transformer.services.ITransformer"/>
 </bean>
 <bean id="urlMapping"
   class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
       <value> /transformer.service=hessianTransformerService </value>
   </property>
  </bean>
```

Accessing Hessian services

- In the client Spring configuration file you have to declare a proxy bean:

```
<bean id="transformerService"
  class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
   <property name="serviceUrl" value="http://localhost:8080/
  transformer/transformer.service"/>
   <property name="serviceInterface"
  value="transformer.services.ITransformer"/>
  </bean>
```

- Java file

```
ApplicationContext factory = new
   ClassPathXmlApplicationContext("classpath:springWS-client.xml");
ITransformer
   transformerServ=(ITransformer)factory.getBean("transformerService");
 String text="Ana are mere";
String response=transformerServ.transform(text);
...
```

Burlap services

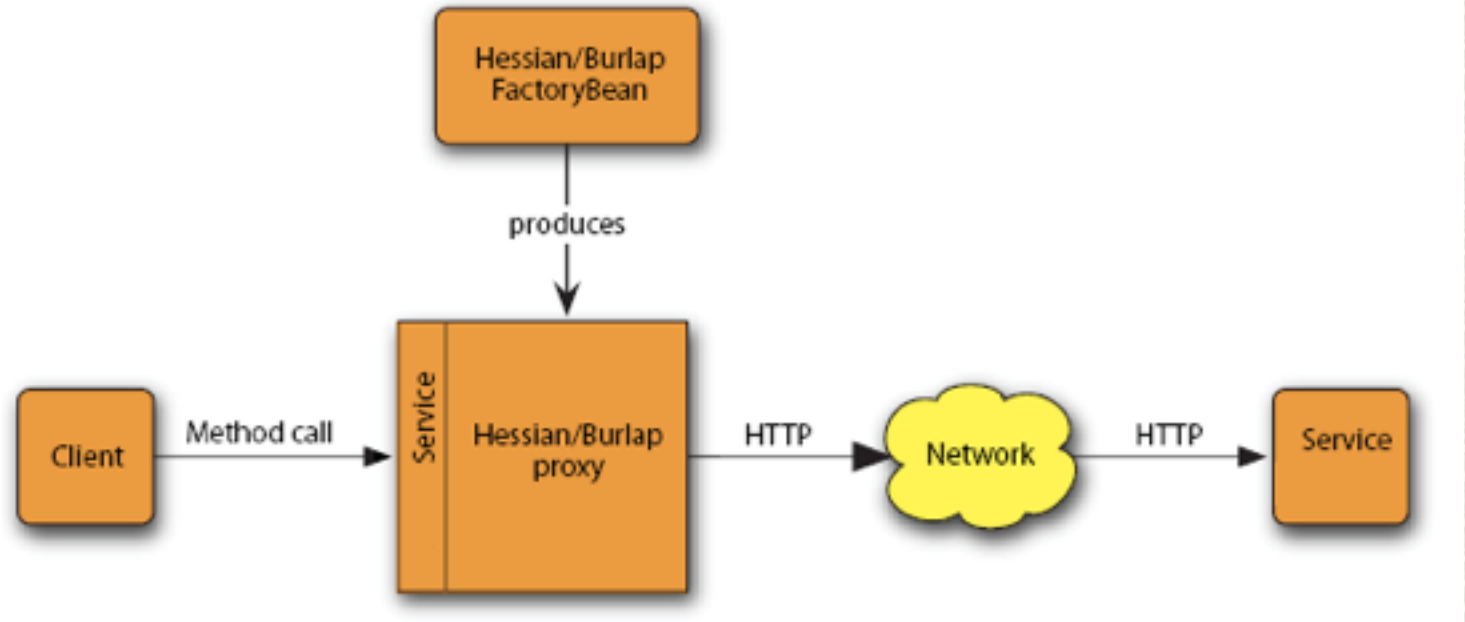
- You have to change Hessian with Burlap in the configuration files (both the service and the client). The rest of the code remains the same.
- Service Spring configuration file

```
<bean id="burlapTransformerService"
   class="org.springframework.remoting.caucho.BurlapServiceExporter">
          <property name="service"  ref= "transformerService"/>
          <property name="serviceInterface"
   value="transformer.services.ITransformer"/>
    </bean>
    <bean id="urlMapping"
   class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
       <property name="mappings">  <value>   /
   transformer.service=burlapTransformerService </value></property>
    </bean>
```

- Client Spring configuration file

```
 <bean id="transformerService"
   class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
    <property name="serviceUrl" value="http://localhost:8080/transformer/
   transformer.service"/>
    <property name="serviceInterface" value="transformer.services.ITransformer"/>
   </bean>
```

Accessing Hessian/Burlap Services



Protocol Buffers (Protobuf)

- They are a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.
- Protocol buffers are flexible, efficient, have an automated mechanism for serializing structured data – similar with XML, but smaller, faster, and simpler.
- You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.
- You can even update your data structure without breaking deployed programs that are compiled against the "old" format.
- You specify how you want the information you're serializing to be structured by defining protocol buffer message types in `.proto` files.
- Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.

Protocol Buffers vs XML

- Protocol buffers have many advantages over XML for serializing structured data:
  - are simpler
  - are 3 to 10 times smaller
  - are 20 to 100 times faster
  - are less ambiguous
  - generate data access classes that are easier to use programmatically
- Protocol buffers are not always a better solution than XML:
  - Protocol buffers would not be a good way to model a text-based document with markup (e.g. HTML), since you cannot easily interleave structure with text.
  - XML is human-readable and human-editable;
  - Protocol buffers are not human-readable and human-editable.
  - XML is self-describing.
  - A protocol buffer is only meaningful if you have the message definition (the .proto file).

Protocol Buffers vs XML

//XML

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
  </person>
```

//Textual representation of Protobuf (not the binary used to wire)

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

Protocol Buffers vs XML

//XML manipulation

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
  </person>
//Java
 person.getElementsByTagName("name").item(0).getNodeValue()
 person.getElementsByTagName("email").item(0).getNodeValue();
```

//Protobuf manipulation

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
System.out.println("Name: " + person.getName());
System.out.println("E-mail: "+ person.email());
```

Protocol Buffers – Example

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Defining A Message Type

- The messages are saved in a `.proto` file

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

- Each field has a name, a type and a tag.
- The type can be:
  - a scalar type (double, float, int32, int64, bool, string, bytes, etc).
  - a composite type (enumeration)
  - another message type

Defining A Message Type

- Enumerations: a field with an enum type can only have one of a specified set of constants as its value.
- If you try to provide a different value, the parser will treat it like an unknown field

```
enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}
```

Specifying Field Rules

- You specify that message fields are one of the following:
  - `required`: a well-formed message must have exactly one of this field.
  - `optional`: a well-formed message can have zero or one of this field (but not more than one).
  - `repeated`: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

- A well-formed message may or may not contain an optional element.
- When a message is parsed, if it does not contain an optional element, the corresponding field in the parsed object is set to the default value for that field.
- The default value can be specified as part of the message description.

```
optional Corpus corpus = 4 [default = UNIVERSAL];
```

Assigning Tags

- Each field in the message definition has a unique numbered tag.
- These tags are used to identify the fields in the message binary format, and should not be changed once your message type is in use.
- The tags with values in the range 1 through 15 take one byte to encode, including the identifying number and the field's type
- Tags in the range 16 through 2047 take two bytes.
- You should reserve the tags 1 through 15 for very frequently occurring message elements.
- The smallest tag number you can specify is 1, and the largest is 536,870,911.
- You cannot use the numbers 19000 though 19999 as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your .proto.

Multiple Message Types

- Multiple message types can be defined in a single .proto file.
- This is useful if you are defining multiple related messages.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

Importing Definitions

- You can use definitions from other `.proto` files by importing them. To import another `.proto's` definitions, you add an import statement to the top of your file:

```
import "myproject/other_protos.proto"
```

- The protocol compiler searches for imported files in a set of directories specified on the protocol compiler command line using the `-I/--proto_path` flag.
- If no flag was given, it looks in the directory in which the compiler was invoked.

Nested Types

- You can define and use message types inside other message types:

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

- If you want to reuse this message type outside its parent message type, you refer to it as `Parent.Type:`

```
message SomeOtherMessage {
  optional SearchResponse.Result result = 1;
}
```

Union Types

- Sometimes a message can be one of several different types.
- Protocol buffer parsers cannot necessarily determine the type of a message based on the contents alone.

```
message OneMessage {
  enum Type { FOO = 1; BAR = 2; BAZ = 3; }

  // Identifies which field is filled in.
  required Type type = 1;

  // One of the following will be filled in.
  optional Foo foo = 2;
  optional Bar bar = 3;
  optional Baz baz = 4;
}
```

Packages

- You can add an optional package specifier to a `.proto` file to prevent name clashes between protocol message types.

```
package foo.bar;
message Open { ... }
```

- You can then use the package specifier when defining fields of your message type:

```
message Foo {
  ...
  required foo.bar.Open open = 1;
  ...
}
```

- The way a package specifier affects the generated code depends on your chosen language:
  - In Java, the package is used as the Java package, unless you explicitly provide the `option java_package` in your .proto file.

Other Options

- Individual declarations in a .proto file can be annotated with a number of options.
- Options do not change the overall meaning of a declaration, but may affect the way it is handled in a particular context.
- The complete list of available options is defined in google/protobuf/descriptor.proto.
- Most commonly used options:
  - `java_package` (file option): The package you want to use for your generated Java classes. If no explicit java_package option is given in the .proto file, then by default the proto package (specified using the "package" keyword in the .proto file) will be used:

```
option java_package = "com.example.foo";
```

  - `java_outer_classname` (file option): The class name for the outermost Java class you want to generate. If no explicit `java_outer_classname` is specified in the `.proto` file, the class name will be constructed by converting the .proto file name to camel-case (so `foo_bar.proto` becomes `FooBar.java`).

```
option java_outer_classname = "Ponycopter";
```

Other Options

- `optimize_for` (file option): Can be set to `SPEED`, `CODE_SIZE`, or `LITE_RUNTIME`. This affects the C++ and Java code generators (and possibly third-party generators) in the following ways:
- `SPEED` (default): The protocol buffer compiler will generate code for serializing, parsing, and performing other common operations on your message types. This code is extremely highly optimized.
- `CODE_SIZE`: The protocol buffer compiler will generate minimal classes and will rely on shared, reflection-based code to implement serialization, parsing, and various other operations.
- `LITE_RUNTIME`: The protocol buffer compiler will generate classes that depend only on the "lite" runtime library (libprotobuf-lite instead of libprotobuf). The lite runtime is much smaller than the full library but omits certain features like descriptors and reflection.

```
option optimize_for = CODE_SIZE;
```

Other Options

- C# options

```
option (google.protobuf.csharp_file_options).namespace =
   "MyCompany.MyProject";
option (google.protobuf.csharp_file_options).umbrella_classname =
   "ProjectProtos";
```

Generating Java/C# classes

- To generate the Java, Python, or C++ code you need to work with the message types defined in a .proto file, you need to run the protocol buffer compiler protoc on the .proto.
- The Protocol Compiler is invoked as follows:

```
protoc --proto_path=IMPORT_PATH --java_out=DST_DIR  path/to/
   file(s).proto
```

- **IMPORT_PATH** specifies a directory in which to look for .proto files when resolving import directives. If omitted, the current directory is used. they will be searched in order. **-I=IMPORT_PATH** can be used as a short form of **--proto_path.**

- //C# proto-csharp

```
protogen MyFile.proto ..\protos\google\protobuf\csharp_options.proto ..
   \protos\google\protobuf\descriptor.proto --proto_path=..\protos;.
```

Generated Classes

- For each message M defined in the proto file there is class M defined in the generated code.
- Each class has its own Builder class that you use to create instances of that class.
- Both messages and builders have auto-generated accessor methods for each field of the message; messages have only getters while builders have both getters and setters.
- The message classes generated by the protocol buffer compiler are all immutable.
- Once a message object is constructed, it cannot be modified.
- To construct a message, you must first construct a builder, set any fields you want to set to your chosen values, then call the builder's build() method.

- Each method of the builder which modifies the message returns another builder. The returned object is actually the same builder on which you called the method. It is returned for convenience so that you can string several setters together on a single line of code.

Generated Classes

- Each message and builder class also contains a number of other methods that let you check or manipulate the entire message:
  - `toByteArray(): byte[]`: serializes the message and returns a byte array containing its raw bytes.
  - `parseFrom(byte[] data): Person`: parses a message from the given byte array.
  - `writeTo(OutputStream output);`: serializes the message and writes it to an OutputStream.
  - `writeDelimitedTo(OutputStream output);` serializes the message and the size and writes it to an OutputStream.
  - `parseFrom(InputStream input): Person;` reads and parses a message from an InputStream.
  - `parseDelimitedFrom(InputStream input): Person;` reads and parses a message from an InputStream (written with writeDelimitedTo).

Chat Case Study