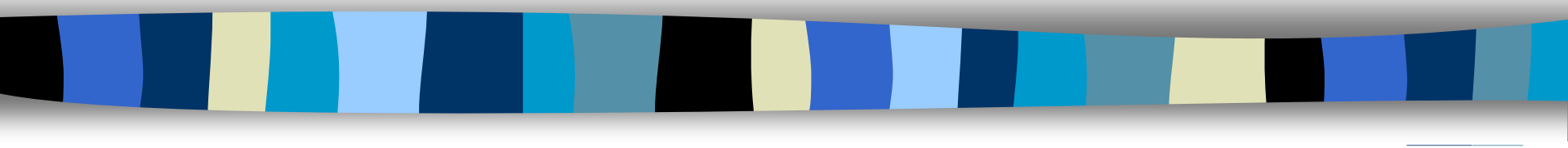
fands™

A Fast AND Steady Approach

Spring Integration



Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.

www.fandsindia.com

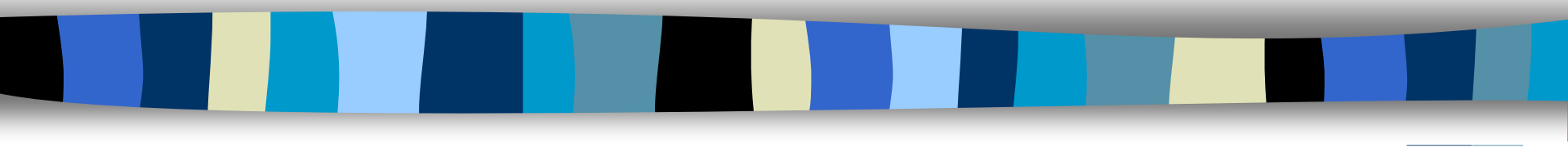
Ground Rules

- **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- **If you have questions or issues, please let me know immediately.**
- **Let us be punctual.**

Agenda

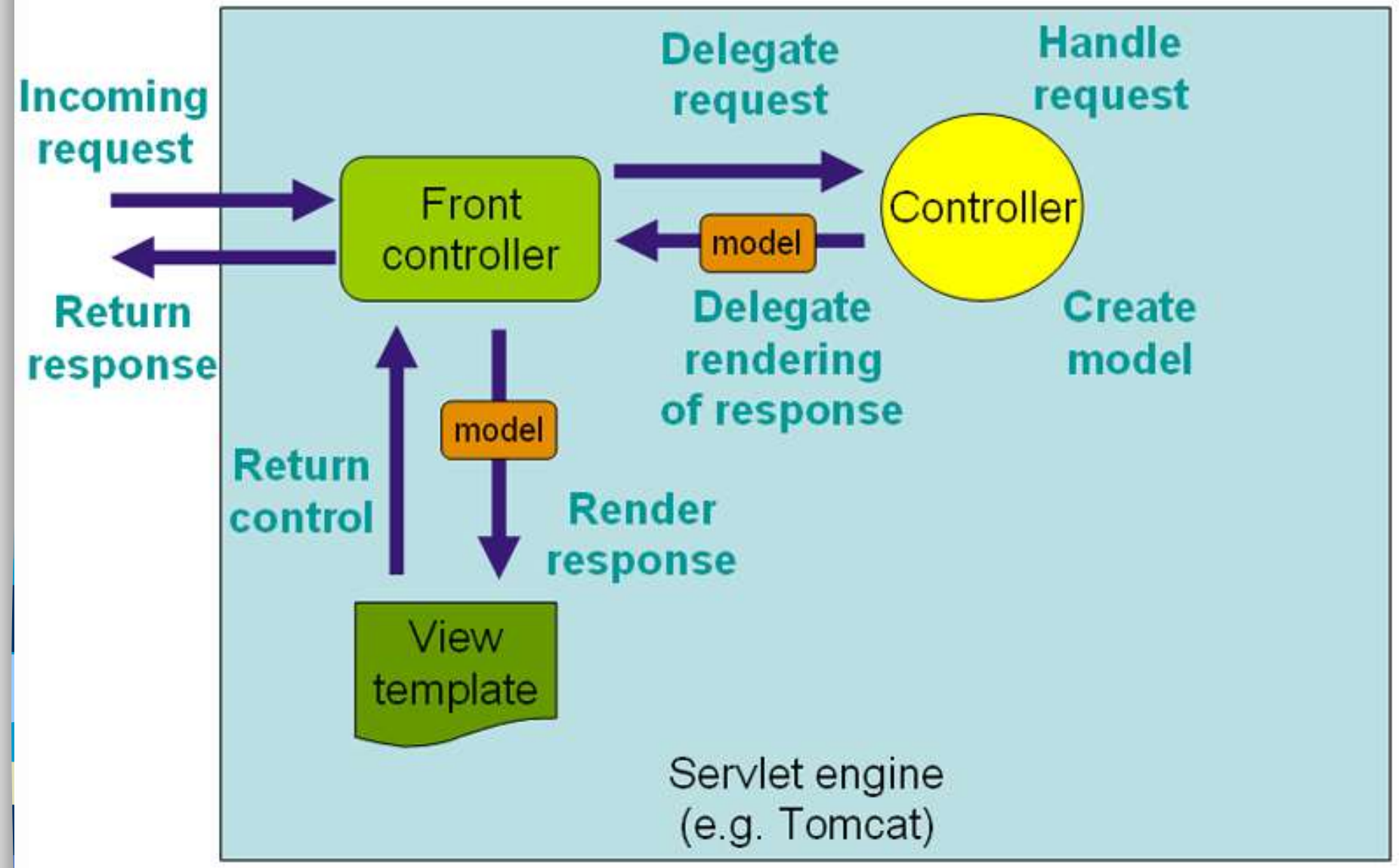
fands™
A Fast AND Steady Approach

Spring MVC



Spring MVC

- MVC Architecture
  - Design Patterns
    - MVC
    - Front Controller


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
Return control
model
Render response
View template
Servlet engine (e.g. Tomcat)

SOA Basics

- Service Oriented Architecture
- Types of Services
  - SOAP
  - REST

Demo

- Simple Annotation driven
- Working with
  - Component, Component-Scan
  - Bean, Configuration
  - Scope
    - Singleton
    - Prototype
  - Bean LifeCycle

Spring Boot

- Need of Spring Boot
- Simple Demo
  - SpringBootApplication
  - Runnable Jar file
  - Standalone vs Web Application

Spring Boot – Web Application

- Spring MVC Application
  - Embedded Tomcat
  - Parameter Processing
    - PathVariable
    - RequestParam
    - Model Transfer as Form Data or JSON
  - Working with Properties / Yml

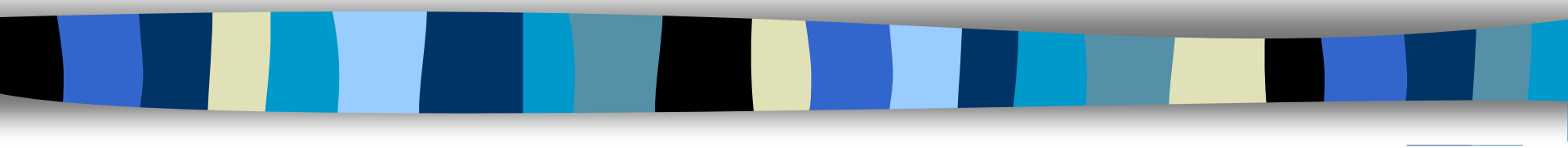
Creating SOAP Service

- Java Web Service
  - Code First -> generate wsdl file
- Spring Web Service
  - Contract First
    - Create java classes from xsd or wsdl using xjc plugin
    - Write code to implement business logic

Spring Boot

- For Operations & Health Monitoring
  - To get details during execution
    - Spring Actuators
    - Expose endpoints
    - Status/info/Heapdump/Threaddump
- For Development
  - DevTools

Spring Data



Spring Data

- Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.
- It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services.

Spring Data Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code
- Easy Spring integration via JavaConfig and custom XML namespaces
- Advanced integration with Spring MVC controllers
- Experimental support for cross-store persistence

Main Modules

- Spring Data Commons
  - Core Spring concepts underpinning every Spring Data project.
- Spring Data Gemfire
  - Provides easy configuration and access to GemFire from Spring applications.
- Spring Data JPA
  - Makes it easy to implement JPA-based repositories.
- Spring Data JDBC
  - JDBC-based repositories.
- Spring Data KeyValue
  - Map-based repositories and SPIs to easily build a Spring Data module for key-value stores.

Main Modules

- Spring Data LDAP
  - Provides Spring Data repository support for Spring LDAP.
- Spring Data MongoDB
  - Spring based, object-document support and repositories for MongoDB.
- Spring Data REST
  - Exports Spring Data repositories as hypermedia-driven RESTful resources.
- Spring Data Redis
  - Provides easy configuration and access to Redis from Spring applications.
- Spring Data for Apache Cassandra
  - Spring Data module for Apache Cassandra.
- Spring Data for Apache Solr
  - Spring Data module for Apache Solr.

Spring Data JPA

- Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Features

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of @Query annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing @EnableJpaRepositories.

Hibernate -> Spring ORM-> Data JPA

```
public void create(Dept dept) {
        Session session = null;
        Transaction tx = null;
        try {
            session = sf.openSession();
            tx = session.beginTransaction();
           session.save(dept);
           tx.commit();
           } catch (Exception e) {
                    ….
        } finally {
                            session.close();
                    }
        }
```

```
void create(Dept d){
template.save(d);
}
```

```
public interface
DeptRepository extends
CrudRepository<Dept,
Integer> {
..
}
```

m

Defining Query Methods

- Two ways to derive a store-specific query
  - Query from the method name directly
  - Using a manually defined query

Strategies

- CREATE
  - attempts to construct a store-specific query from the query method name.
- USE_DECLARED_QUERY
  - tries to find a declared query and will throw an exception in case it can't find one. The query can be defined by an annotation somewhere or declared by other means.
- CREATE_IF_NOT_FOUND
  - (default) combines CREATE and USE_DECLARED_QUERY.
  - It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query.

Query Creation

- The mechanism strips the prefixes find…By, read…By, query…By, count…By, and get…By from the method and starts parsing the rest of it.
- The introducing clause can contain further expressions such as a Distinct to set a distinct flag on the query to be created. However, the first By acts as delimiter to indicate the start of the actual criteria. At a very basic level you can define conditions on entity properties and concatenate them with And and Or.

Query Examples

```
interface PersonRepository extends Repository<User, Long> {
  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);
  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);
  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Special Parameter Handling

- To handle parameters in your query you simply define method parameters as already seen in the examples above.
- Besides that the infrastructure will recognize certain specific types like Pageable and Sort to apply pagination and sorting to your queries dynamically

Special Parameter Handling

- A Page knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number.
- As this might be expensive depending on the store used, Slice can be used as return instead. A Slice only knows about whether there's a next Slice available which might be just sufficient when walking through a larger result set.

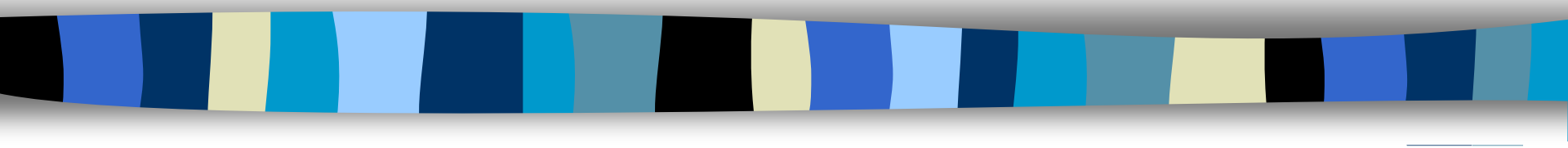
Examples

- Page<User> findByLastname(String lastname, Pageable pageable);
- Slice<User> findByLastname(String lastname, Pageable pageable);
- List<User> findByLastname(String lastname, Sort sort);
- List<User> findByLastname(String lastname, Pageable pageable);

fands™

A Fast AND Steady Approach

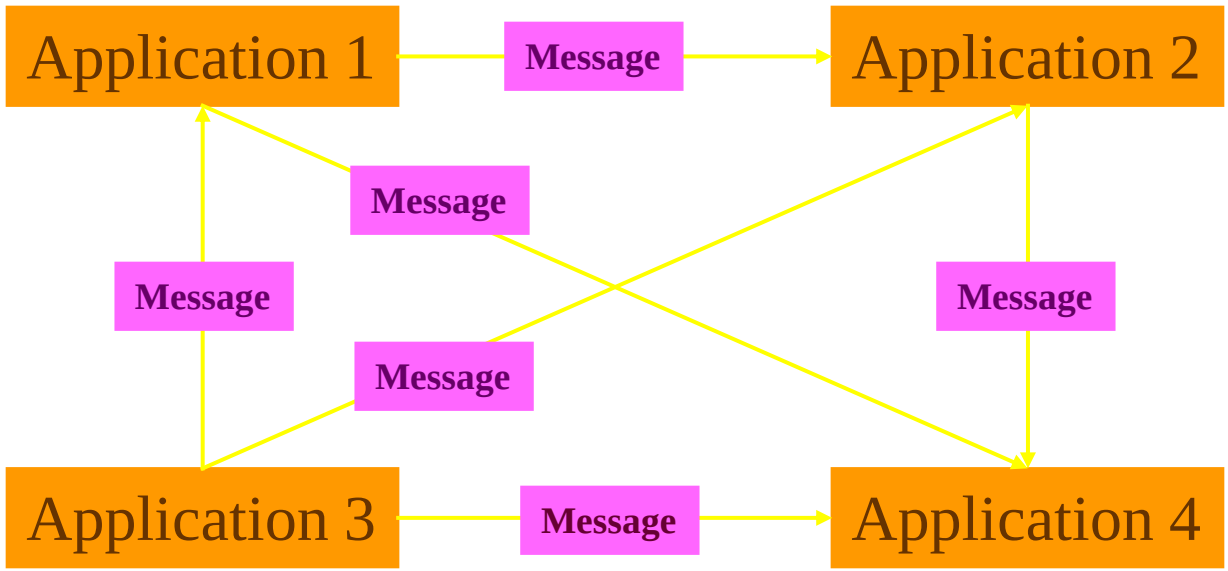
JMS



What is a Messaging System?

- A system that allows separate, uncoupled applications to reliably communicate asynchronously
- Supports peer-to-peer communication between components

Application Distribution



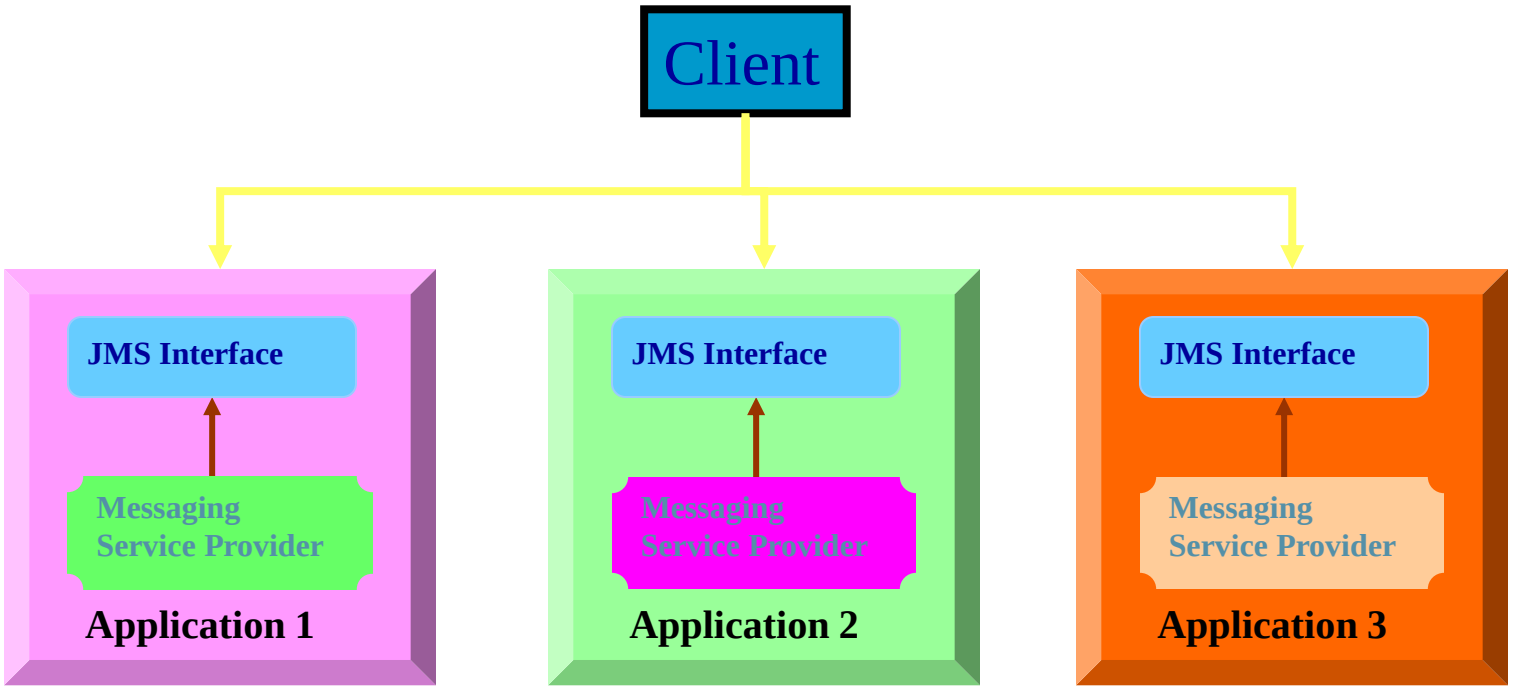
Problems?

- Vendor specific Messaging API
- Non-interoperable
- A standard API required for interoperability

Java Message Service

- Defines a standard Java-based interface
- Messaging Service vendors implement the interface
- Clients use the interface to interoperate with other vendor's messaging service

Application Architecture



Some Definitions

- **Messaging Client**
  - Messaging is peer-to-peer. Every peer is called a client
- **Message Producer**
  - Message Sender is called a producer
- **Message Consumer**
  - Message Receiver is called a consumer

A Client may be a Producer, a Consumer, or both

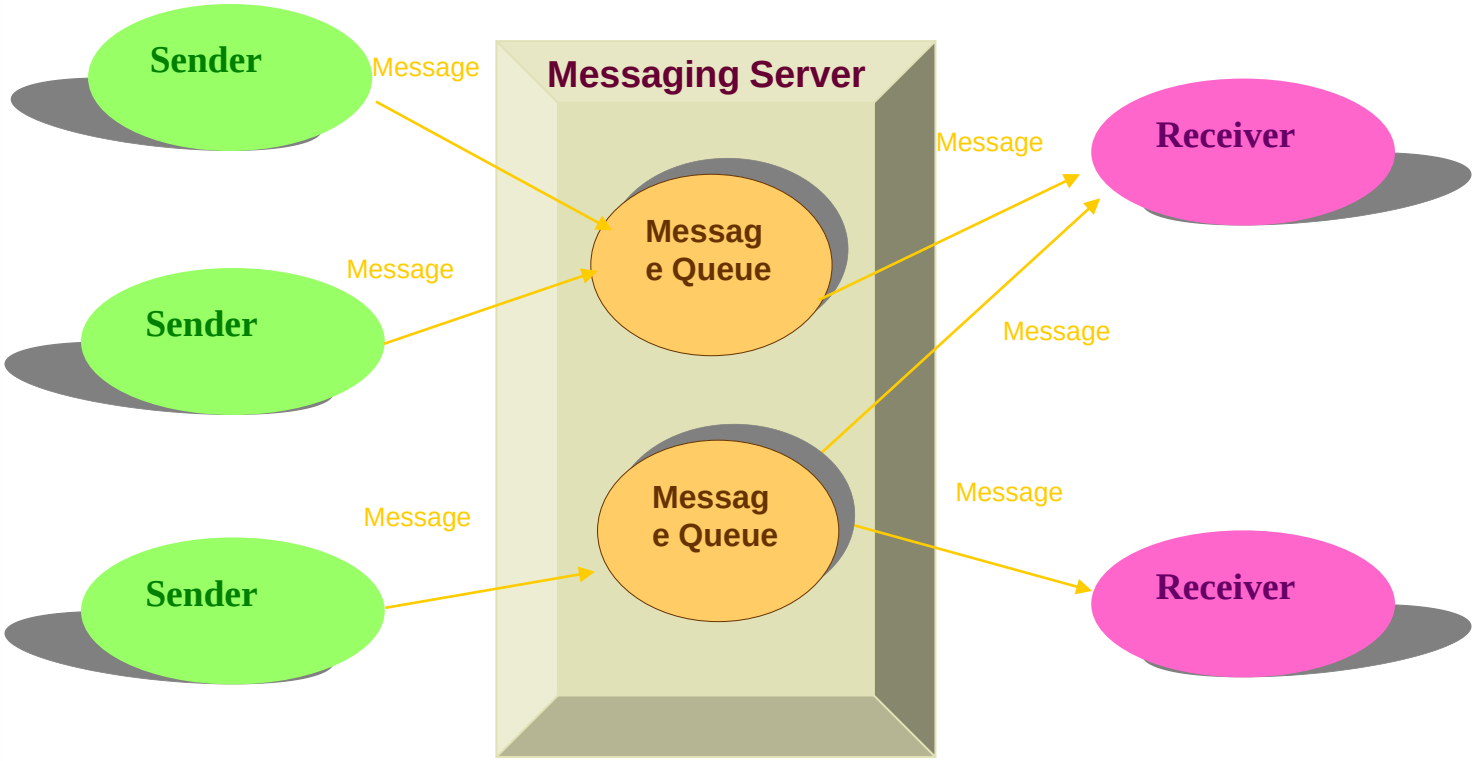
Some Definitions

- **Durable Message Delivery**
  - Router holds the message until the time client becomes active and receives message
- **Persistent Message Delivery**
  - If Messaging Server goes down before a message is delivered, the delivery is still guaranteed

Communication Models

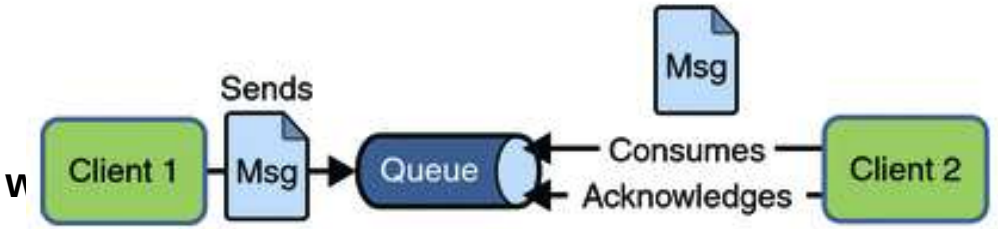
- **Point-To-Point (PTP)**
  - Used if there is one and only one receiver for each message
- **Publish-Subscribe (PS)**
  - Used for general broadcast kind of applications

Point-To-Point Model

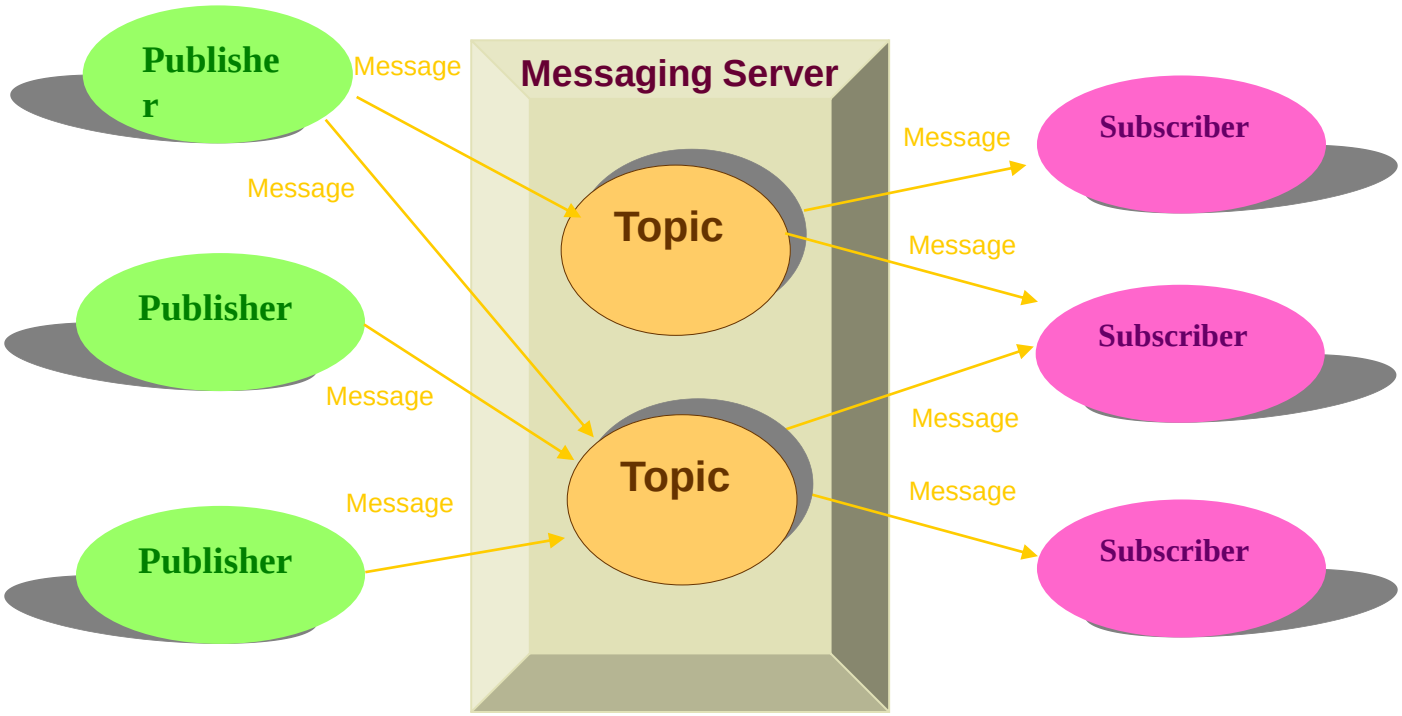


PTP Features

- Producer sends Messages to the queue
- Consumer receives message from the queue
- Multiple senders may send messages to same queue
- One and only one consumer for every message
- System may provide multiple queues.

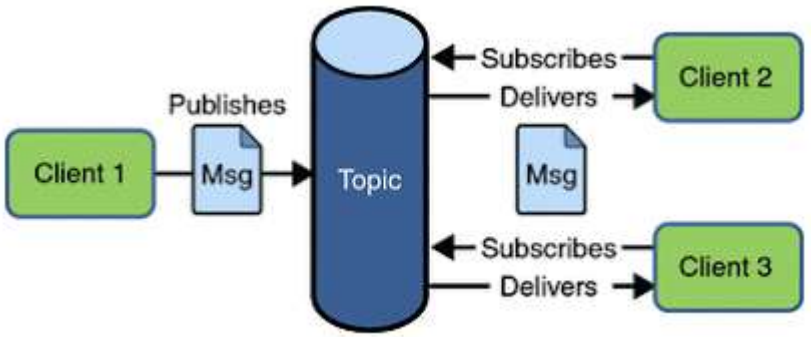


Publish-Subscribe Model



PS Features

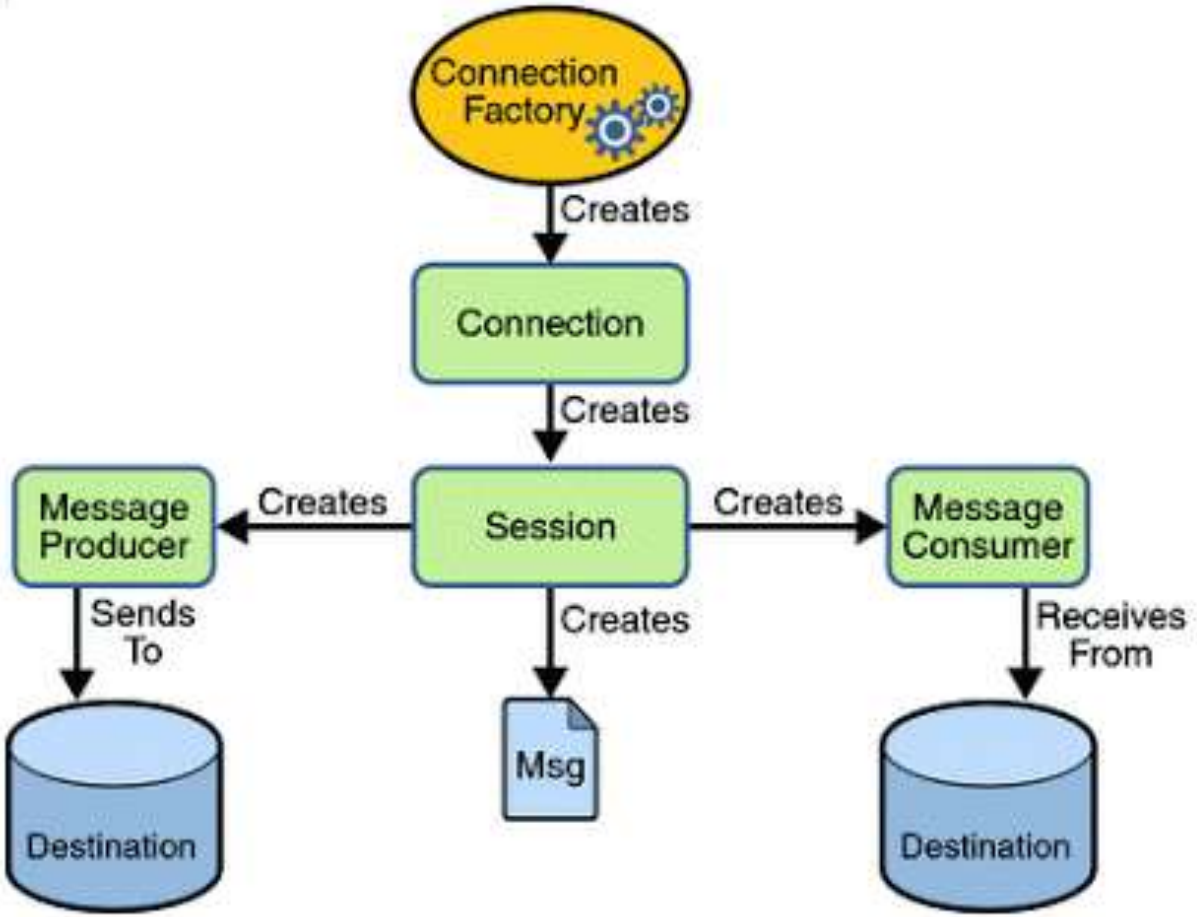
- Producer publishes messages to a topic
- Consumer subscribes to interested topic
- A message may be consumed by multiple subscribers
- Multiple publishers and subscribers to a single topic



Message Delivery Model

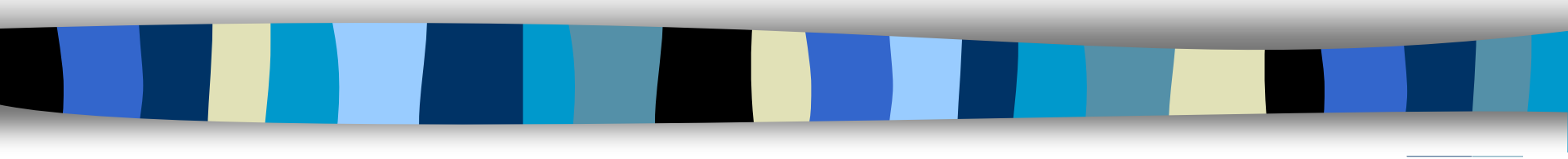
- **Synchronous**
  - Consumer is blocked until
    - Message is received
    - Timeout set by consumer expires
    - Consumer closes
- **Asynchronous**
  - Messaging system responsible for callback on consumer

Messaging Model



fands™
A Fast AND Steady Approach

Spring Integration



What is EAI?

- Enterprise Application Integration
  - Been around for as long as disparate computing paradigms have
- EAI is integration of services and / or data.
- Enterprise Application Integration platforms have changed names
  - Now known as: an Enterprise Service Bus (ESB)
- Types of EAI:
  - File Transfer
  - Shared Database
  - Remote Procedure Call
  - Messaging

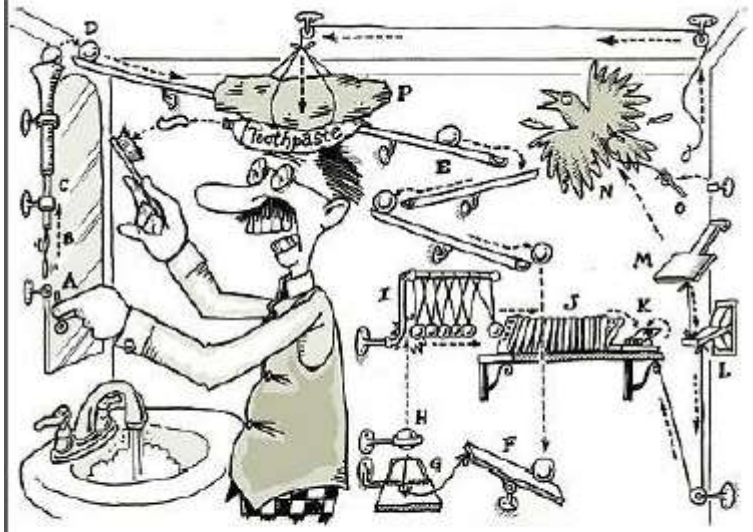
Common ESB Capabilities

- Location Transparency
- Transport Conversion
- Message Transformation / Routing / Enhancement
- Security
  - Spring Security Integration's on the roadmap
- Monitoring and management
  - you might use JMX for Spring Integration
- Process management (BPMs, orchestration)
- Complex Event Processing

Spring Integration

- New addition to the Spring Portfolio
- Provides support for SOA, EDA and EAI
  - It went live with a 1.0 late in 2008.
- Provides philosophical consistency with core Spring principles.
- Spring Integration is an API geared towards building ESB - centric solutions, not another name for the Spring remoting APIs.

The ESB Landscape



* Rube Goldberg solution -- you've never built something like this, have you? Really??

- Traditional EAI solutions from the likes of TIBCO, Axway, WebMethods
- Open source / Java-centric solutions like ServiceMix, PEtALS, OpenESB, JBossESB, Mule (and of course Spring Integration)
- The alternatives: solutions strung together with bailing wire and tape, a veritable Rube Goldberg machine * Rube Goldberg solution -- you've never built something like

Spring Integration

- Uses standard Spring XML
- Idiomatic configuration: annotations, schema
- Spring Integration is embedded: deploy Spring
- Integration with your app - you don’t deploy your application to Spring Integration

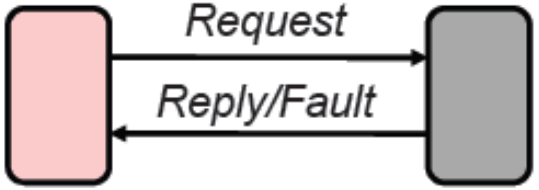
Spring Integration

- You'll typically deal with three things in an integration solution
  - Messages
  - Message Channels
  - Message Endpoints
    - Adapters

Synchronous and Asynchronous Interactions

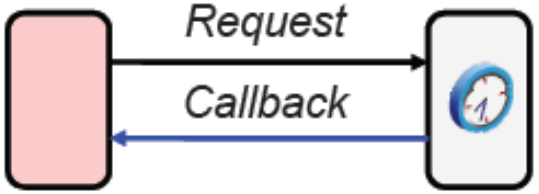
Synchronous request/response

- Real-time response or error feedback
- Client in waiting mode



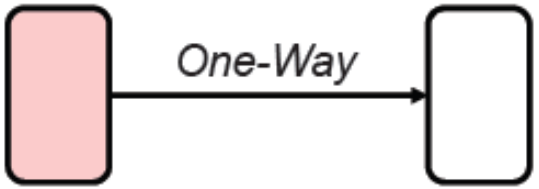
Asynchronous request/callback

- Client free after request submission
- Separate service invocation for response



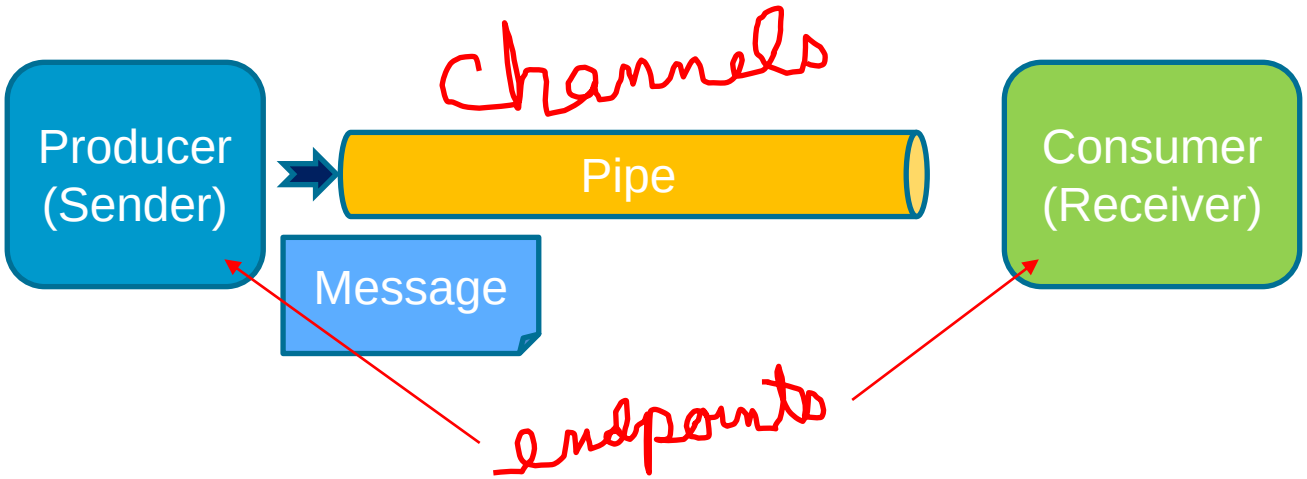
Asynchronous request only

- Also known as “fire and forget”
- Client free after request submission
- No response

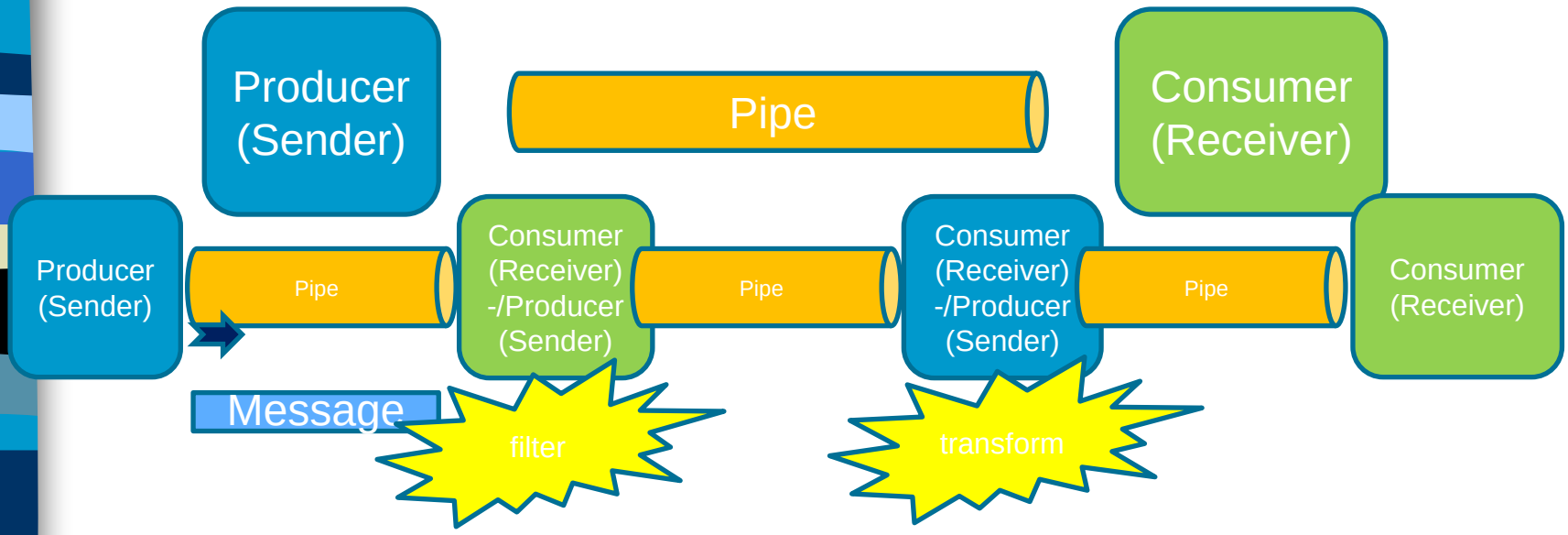


Spring Integration

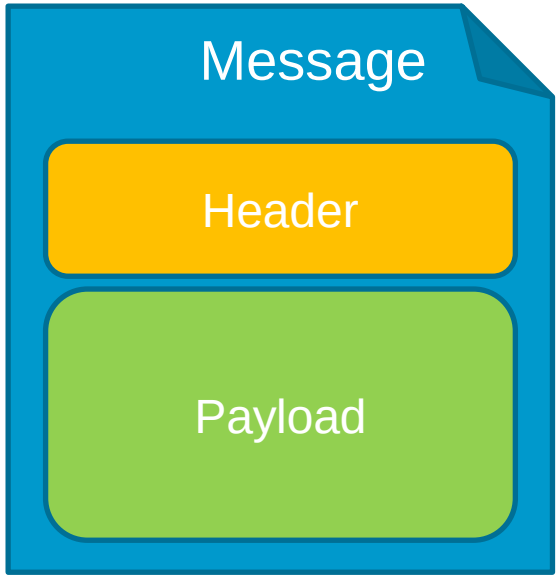
The main components

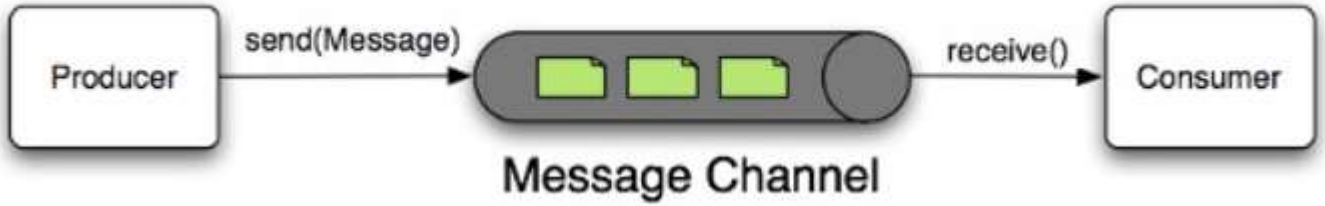


Spring Integration Applications



Message and Message Channel





Message Endpoints

- Adapters (connect your channel to some other system)
- Filter (remove some messages from channels based on header, content, etc.)
- Transformer (convert a message content or structure)
- Enricher (add content to the message header or payload)
- Service activator (invoke service operations based on the arrival of a message)
- Gateway (connect your channels without SI coupling)

Message Channels

- Two general classifications of message channels
  - **Pollable Channel**
  - **Publish-Subscribe Channel**
- Another important consideration: Should the channel buffer messages
  - In Spring Integration, pollable channels are capable of buffering Messages within a queue.

Pollable Channels

- May buffer its messages
  - Requires a queue to hold the messages
  - The queue has a designated capacity
- Waits for the consumer to get the messages
  - Consumers actively poll to receive messages
- Typically a point-to-point channel
  - Only one receiver of a message in the channel
- Usually used for sending information or “document” messages between endpoints

Subscribable

- Allows multiple subscribers (or consumers) to register for its messages.
  - Messages are delivered to all registered subscribers on message arrival
  - It has to manage a list or registry of subscribers.
- Doesn't buffer its messages
- Usually used for "event" messages
  - Notifying the subscribers that something happened and to take appropriate action.

EndPoints

- Consuming endpoints (anything with an inputChannel) consist of two beans, the consumer and the message handler. The consumer has a reference to the message handler and invokes it as messages arrive.
- Consumer:
  - someComponent.someMethod.serviceActivator
- Handler:
  - someComponent.someMethod.serviceActivator.handler

Channel Adapters (Adapters)

- Adapters provide a higher level of abstraction over Spring's support for remoting, messaging, and scheduling.
- A Channel Adapter is a Message Endpoint that enables connecting a single sender or receiver to a Message Channel.
- Provides a number of adapters out of the box to support various transports
  - E.g JMS, File, HTTP, Web Services, Mail etc
- Inbound and Outbound adapters, and each may be configured with XML elements provided in the core namespace

Channel Types

- PublishSubscribeChannel
- QueueChannel
- PriorityChannel
- DirectChannel
- ExecutorChannel
- Scoped Channel

PublishSubscribeChannel

- Broadcasts any Message sent to it to all of its subscribed handlers.
- Often used for sending event messages, whose primary role is notification
- Intended for sending only.
  - Since it broadcasts to its subscribers directly when its send(Message) method is invoked, consumers cannot poll for messages **(it does not implement PollableChannel and therefore has no receive() method)**. Instead, any subscriber must itself be a MessageHandler, and the subscriber's handleMessage(Message) method is invoked in turn.

QueueChannel

- Point-to-point
- Wraps a queue - FIFO
- If the channel has multiple consumers, only one of them should receive any Message sent to that channel.
  - It provides a default no-argument constructor as well as a constructor that accepts the queue capacity
  - Additional timeout options

PriorityChannel

- Ordering based on priority
  - An alternative implementation to QueueChannel that allows for messages to be ordered within the channel based upon a priority.
- Priority is determined by
  - the priority header within each message.
  - For custom priority determination logic, a comparator of type Comparator<Message<?>> can be provided to the constructor.

DirectChannel

- Only one Subscriber without Queue
- Implements the SubscribableChannel interface instead of the PollableChannel interface, so it dispatches messages directly to a subscriber.

ExecutorChannel

- Point-to-point channel
- Similar to DirectChannel with additional configuration for TaskExecutor to perform the dispatch.
  - This means that the send method typically does not block, but it also means that the handler invocation may not occur in the sender's thread. It therefore does not support transactions that span the sender and receiving handler.

Scoped Channel

- Scope attribute to a channel.
  - The value of the attribute can be the name of a scope that is available within the context.
  - Difference scopes in different environment

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="thread" value="org.springframework.context.support.SimpleThreadScope" />
    </map>
  </property>
</bean>
```

Channel Interceptors

- Interceptors Concept
  - Servlet Filters
  - AOP
- Capture meaningful information about the messages passing through the system in a non-invasive way.
- Since the Message instances are sent to and received from MessageChannel instances, those channels provide an opportunity for intercepting the send and receive operations.

fands™

Channel Interceptors

```
public interface ChannelInterceptor {

    Message<?> preSend(Message<?> message, MessageChannel channel);

    void postSend(Message<?> message, MessageChannel channel, boolean sent);

    void afterSendCompletion(Message<?> message, MessageChannel channel, boolean
sent, Exception ex);

    boolean preReceive(MessageChannel channel);

    Message<?> postReceive(Message<?> message, MessageChannel channel);

    void afterReceiveCompletion(Message<?> message, MessageChannel channel,
Exception ex);
}
```

Message Bridge

- Simple endpoint that connects two message channels or channel Adapters
- Usage
  - Connect a PollableChannel to SubscribableChannel
  - Connect two different systems

```
@Bean
public PollableChannel polled() {
    return new QueueChannel();
}

@Bean
@BridgeFrom(value = "polled", poller = @Poller(fixedDelay = "5000",
maxMessagesPerPoll = "10"))
public SubscribableChannel direct() {
    return new DirectChannel();
}
```

Messaging Template

- Required when we need to invoke the messaging system from your application code.
- MessagingTemplate supports a variety of operations across the message channels, including request and reply scenarios

```
MessagingTemplate template = new MessagingTemplate();

Message reply = template.sendAndReceive(someChannel, new GenericMessage("test"));
```

Message Routing

- Crucial element in messaging architectures.
- Consume messages from a message channel and forward each consumed message to one or more different message channels depending on a set of conditions.
- Route messages based on
  - Header
  - Payload
  - Error
  - Recipient List

Message Routing

- Routers
- Filters
- Splitter
- Aggregation
- Message Handler Chain

Routers

- Consumes messages from a message channel and forwards each consumed message to one or more different message channels depending on a set of conditions.

```
@ServiceActivator(inputChannel = "routingChannel")
@Bean
public PayloadTypeRouter router() {
    PayloadTypeRouter router = new PayloadTypeRouter();
    router.setChannelMapping(String.class.getName(), "stringChannel");
    router.setChannelMapping(Integer.class.getName(), "integerChannel");
    return router;
}
```

Message Filters

- Message filters are used to decide whether a Message should be passed along or dropped based on some criteria, such as a message header value or message content itself.
- Similar to a router, except that, for each message received from the filter's input channel, that same message may or may not be sent to the filter's output channel.

Splitters and Aggregators

- The Splitter is a component whose role is to partition a message in several parts, and send the resulting messages to be processed independently.
- Aggregator is a type of Message Handler that receives multiple Messages and combines them into a single Message

Message Handler Chain

- Implementation of MessageHandler that can be configured as a single message endpoint while actually delegating to a chain of other handlers, such as filters, transformers, splitters, and so on. When several handlers need to be connected in a fixed, linear progression, this can lead to a much simpler configuration.

SOAP ← → REST

- SOAP
  - WSDL
  - Stubs generation using wsimport
  - Invoke using simple client
  - JAX-WS
- REST
  - Invoke SOAP Provider
  - JAX-RS

Http Inbound

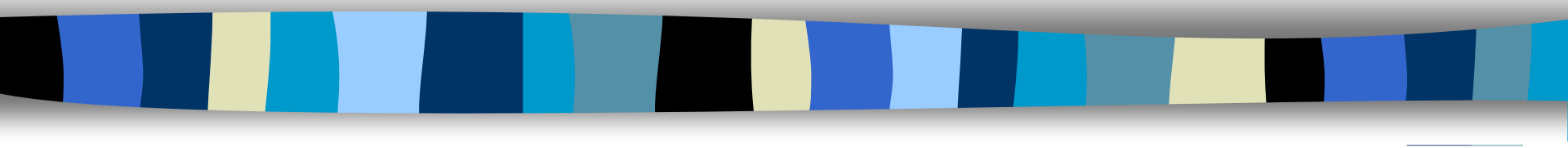
- Exposing Spring Integration workflow as REST API
  - Inbound Gateway
  - RequestMapping
  - Handler

```
@Bean
public HttpRequestHandlingMessagingGateway inbound() {
    HttpRequestHandlingMessagingGateway gateway =  new ..
    gateway.setRequestMapping(mapping());
    gateway.setRequestChannelName("httpRequest");
    gateway.setReplyChannelName("reply");
    gateway.setReplyTimeout(120000);
    return gateway;}

@Bean
public RequestMapping mapping() {
    RequestMapping requestMapping = new RequestMapping();
    requestMapping.setPathPatterns("/simple");
    requestMapping.setMethods(HttpMethod.GET);
    requestMapping.setProduces("text/plain");
    return requestMapping; }

@Bean
@ServiceActivator( inputChannel = "httpRequest",outputChannel
= "reply")
public Function<Message<?>, String> handler() {
    return new Function<Message<?>, String>() {
        public String apply(Message<?> message) {
return "Spring Integration as REST API";
```

Swagger



Swagger

- Why?
- How?
  - POM

```
<dependency>
	<groupId>org.springdoc</groupId>
	<artifactId>springdoc-openapi-ui</artifactId>
	<version>1.6.8</version>
</dependency>
```

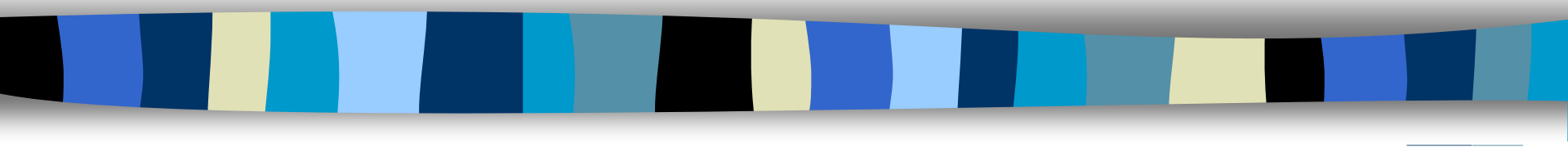
  - Properties

```
springdoc.swagger-ui.path=/swagger-ui.html
springdoc.api-docs.path=/api-docs
```

Swagger

- Swagger Code gen
  - For
    - Server
    - Client
  - How
    - From online editor
    - Plugin

Security



Security

- Security is a crucial aspect of most applications
- Security is a concern that transcends an application's functionality
- An application should play no part in securing itself
- It is better to keep security concerns separate from application concerns

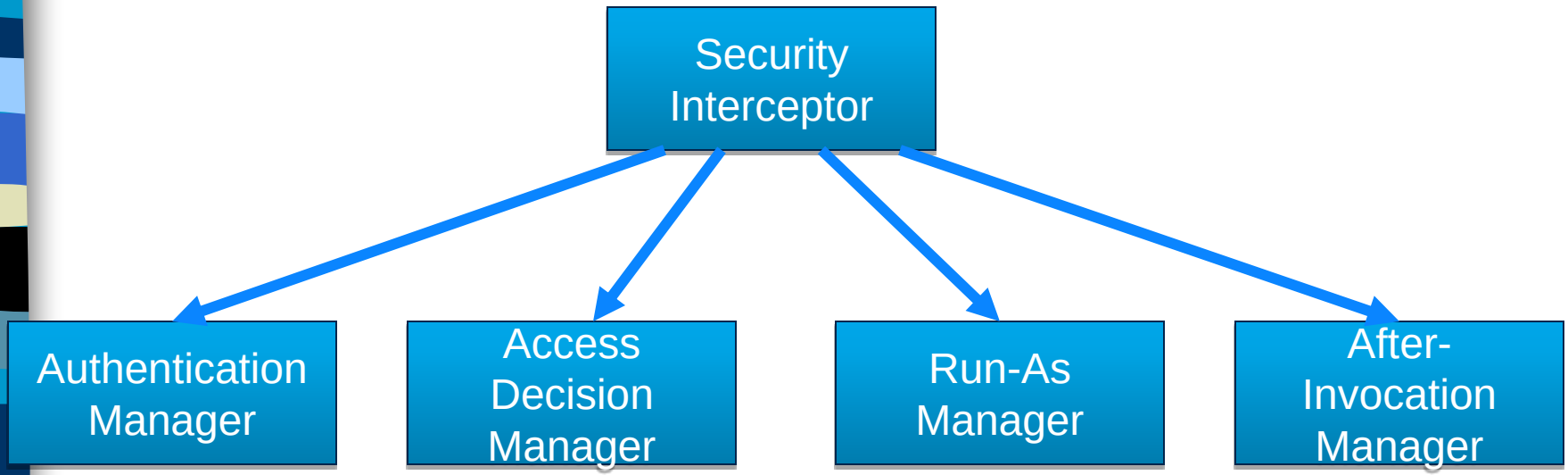
Acegi Security

- Started in 2003
- Became extremely popular
- Security Services for the Spring framework
- From version 1.1.0, Acegi becomes a Spring Module

Key concepts

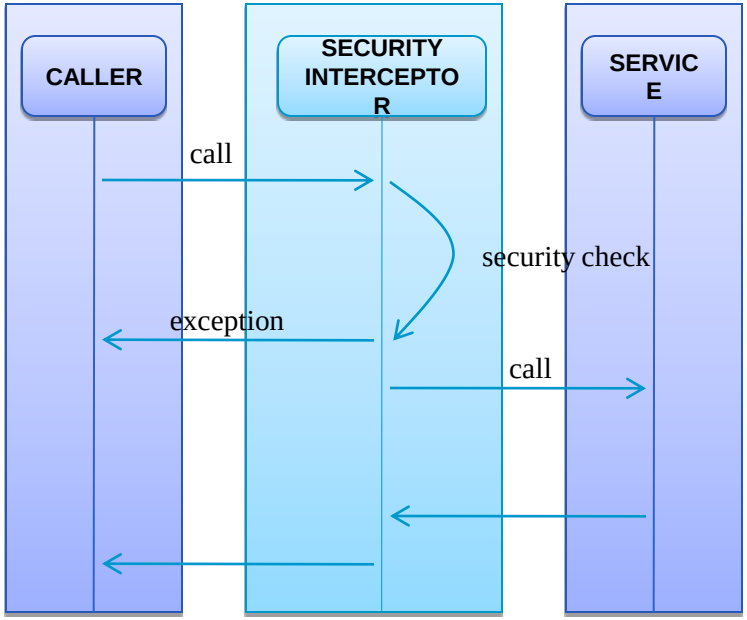
- Filters (Security Interceptor)
- Authentication
- Authorization
- Web authorization
- Method authorization

Fundamental Elements

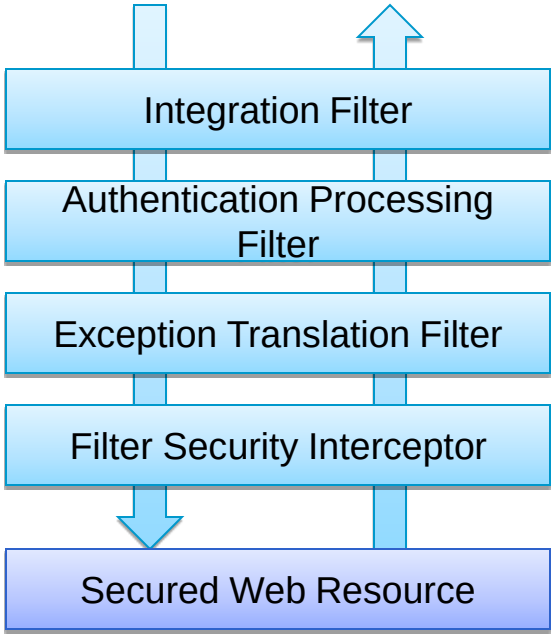


Security Interceptor

- A latch that protects secured resources, to get past users typically enter a username and password
- Implementation depends on resource being secured
  - URLs - Servlet Filter
  - Methods - Aspects
- Delegates the
- responsibilities to the
- various managers



Spring Security Filters



Filter	What it does
Integration Filter	responsible for retrieving a previously stored authentication (most likely stored in the HTTP session) so that it will be ready for Spring Security's other filters to Process
Authentication Processing Filter	determine if the request is an authentication request. If so, the user information (typically a username/ password pair) is retrieved from the request and passed on to the authentication manager
Exception Translation Filter	translates exceptions, for AuthenticationException request will be sent to a login screen, for AccessDeniedException returns HTTP 403 to the browser
Filter Security Interceptor	examine the request and determine whether the user has the necessary privileges to access the secured resource. It leans heavily on the authentication manager and the access decision manager

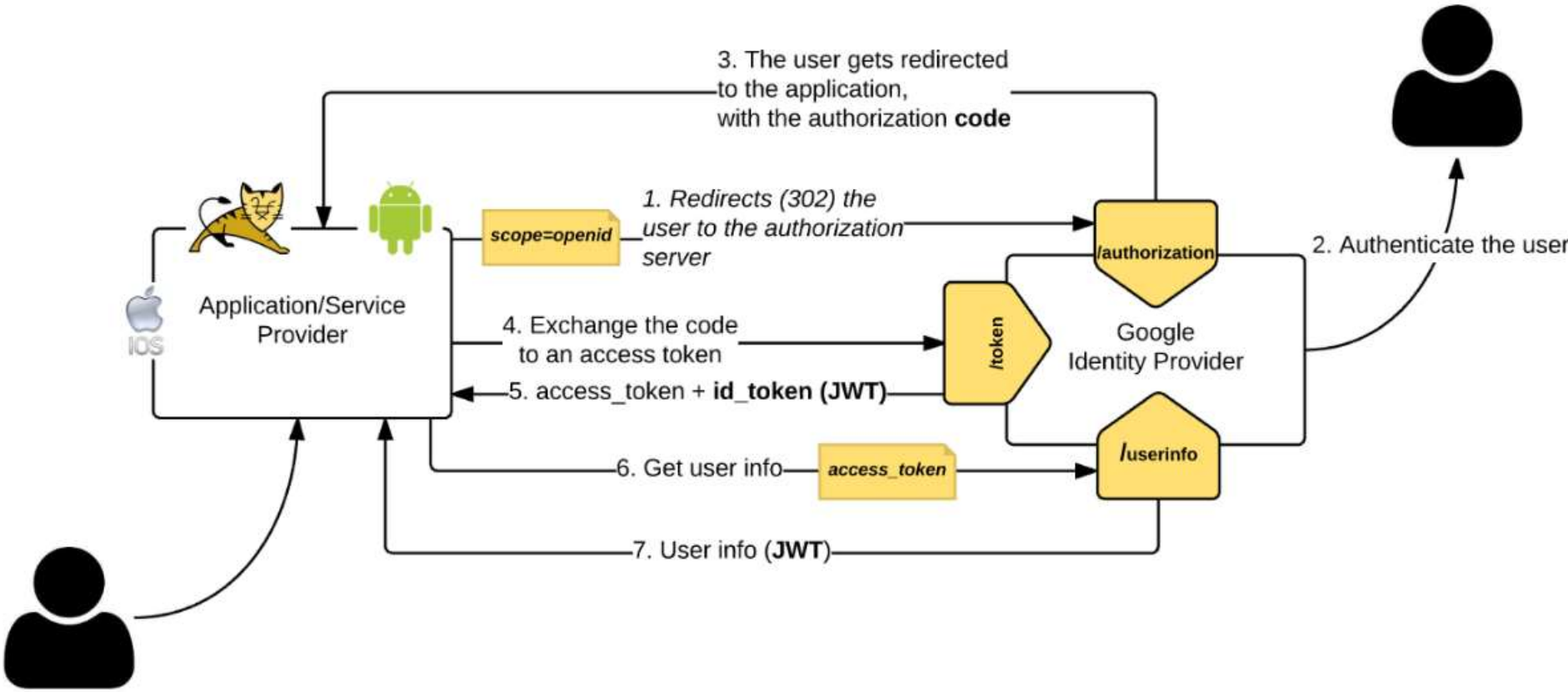
OpenID

- OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

JWT

- JSON Web Token (JWT) defines a container to transport data between interested parties. It became an IETF standard in May 2015 with the RFC 7519. There are multiple applications of JWT. The OpenID Connect is one of them. In OpenID Connect the id_token is represented as a JWT. Both in securing APIs and Microservices, the JWT is used as a way to propagate and verify end-user identity.

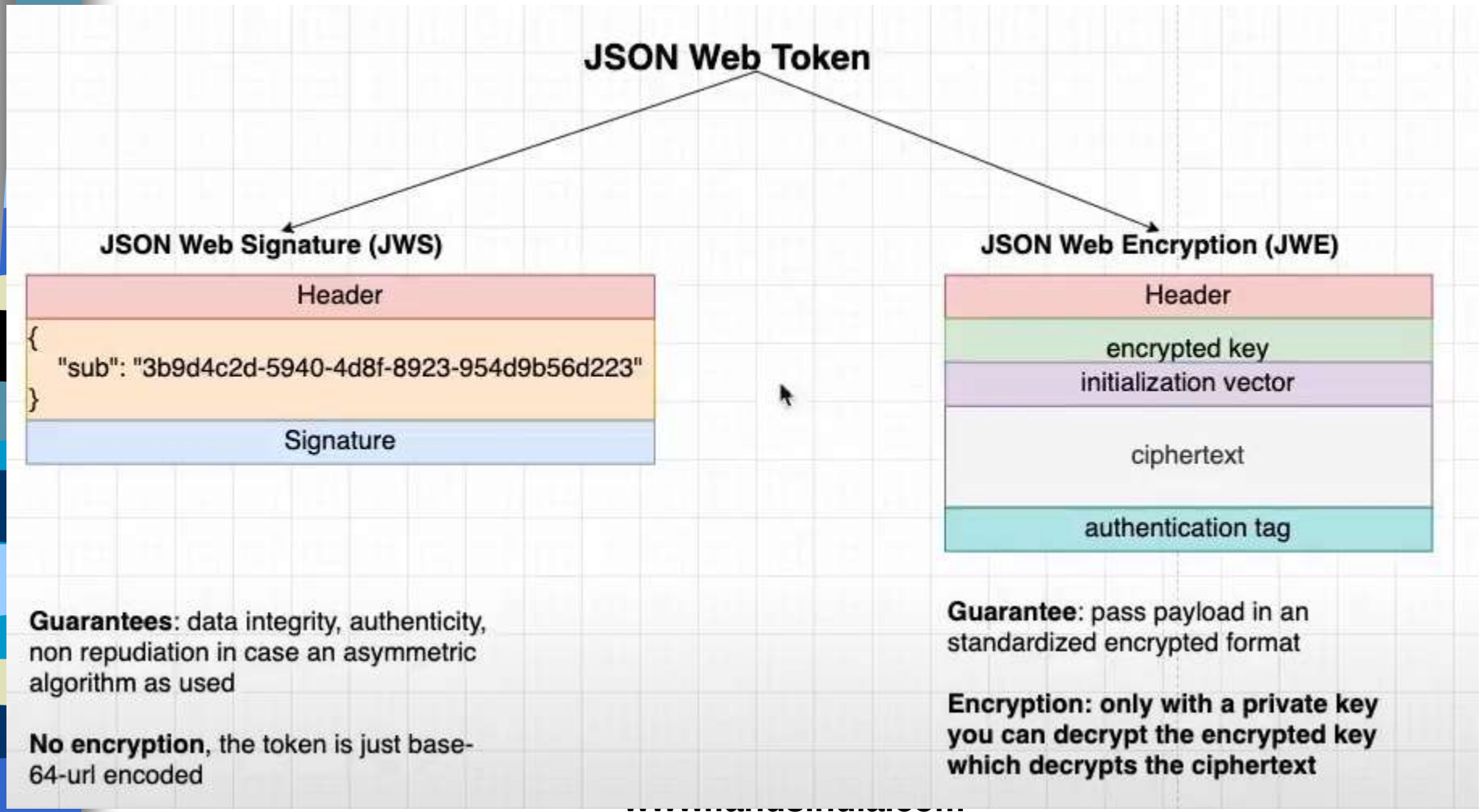
JWT


3. The user gets redirected
to the application,
with the authorization code
1. Redirects (302) the
user to the authorization
server
scope=openid
/authorization
2. Authenticate the user
Application/Service
Provider
IOS
4. Exchange the code
to an access token
5. access_token + id_token (JWT)
/token
Google
Identity Provider
/userinfo
6. Get user info
access_token
7. User info (JWT)

JWS and JWE

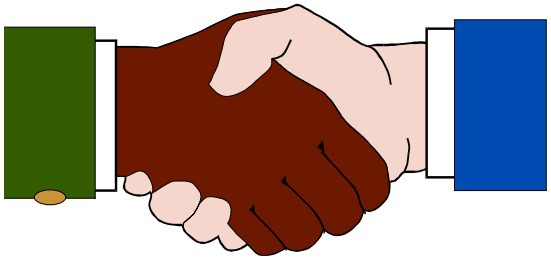
- A signed JWT is known as a JWS (JSON Web Signature).
- An encrypted JWT is known as JWE (JSON Web Encryption)
- In fact a JWT does not exist itself - either it has to be a JWS or a JWE
- More like an abstract class - the JWS and JWE are the concrete implementations.

JWT


JSON Web Token
JSON Web Signature (JWS)
Header
{
"sub": "3b9d4c2d-5940-4d8f-8923-954d9b56d223"
}
Signature
JSON Web Encryption (JWE)
Header
encrypted key
initialization vector
ciphertext
authentication tag
Guarantees: data integrity, authenticity, non repudiation in case an asymmetric algorithm as used
No encryption, the token is just base-64-url encoded
Guarantee: pass payload in an standardized encrypted format
Encryption: only with a private key you can decrypt the encrypted key which decrypts the ciphertext

QUESTION / ANSWERS

THANKING YOU !



www.fandsindia.com