JavaOne™

ORACLE®

The New HTTP Client API

including HTTP/2 and Websockets

Michael McMahon
Software Engineer
Java Core Libraries
Oracle

Safe Harbor Statement

Program Agenda

1. HTTP/2 (similarities and differences from 1.1)
2. CompletableFuture API
3. New HTTP API
4. Web sockets API

Program Agenda

1. HTTP/2 (similarities and differences from 1.1)
2. CompletableFuture API
3. New HTTP API
4. Web sockets

HTTP/2

What is similar?

- HTTP verbs/request methods
  - GET, POST, PUT, DELETE etc. have same meaning
- Request and response headers
  - Content-length, Cookie, Content-encoding etc (same meaning)
- Basic request/response structure
  - One request
  - Zero or more intermediate responses
  - One final response
- Request and response body data

HTTP/2

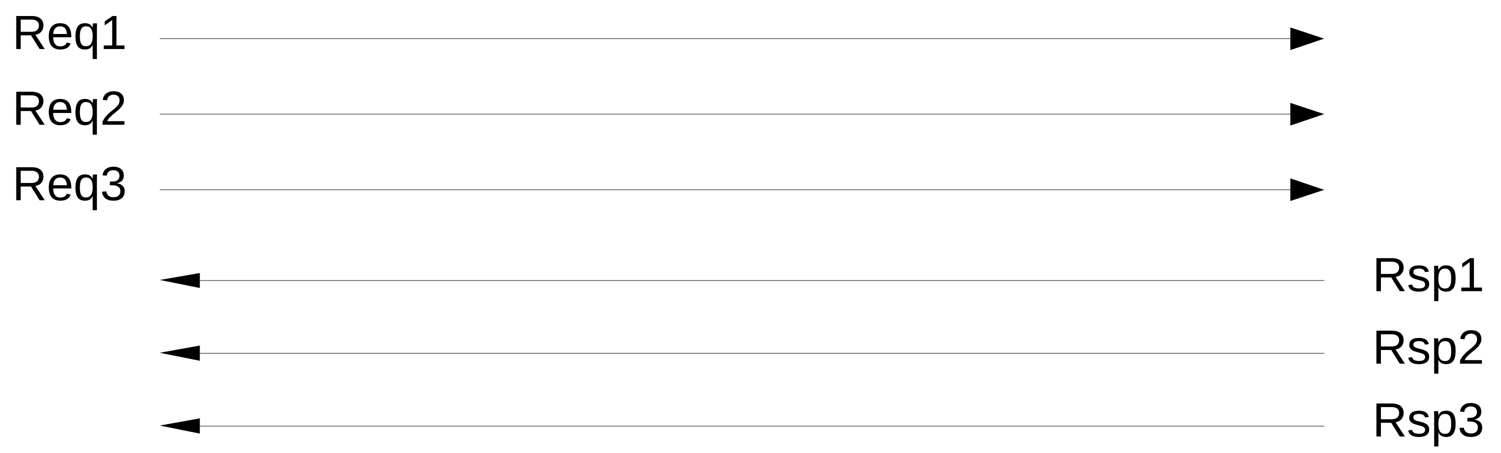
What is different?

- Multiplexes multiple requests on same TCP connection
  - without head of line blocking problem that 1.1 pipelined requests have

HTTP/2

What is different?

- Multiplexes multiple requests on same TCP connection
  - without head of line blocking problem that 1.1 pipelined requests have
- Illustrate problem with 1.1



Responses must be delivered in request order.

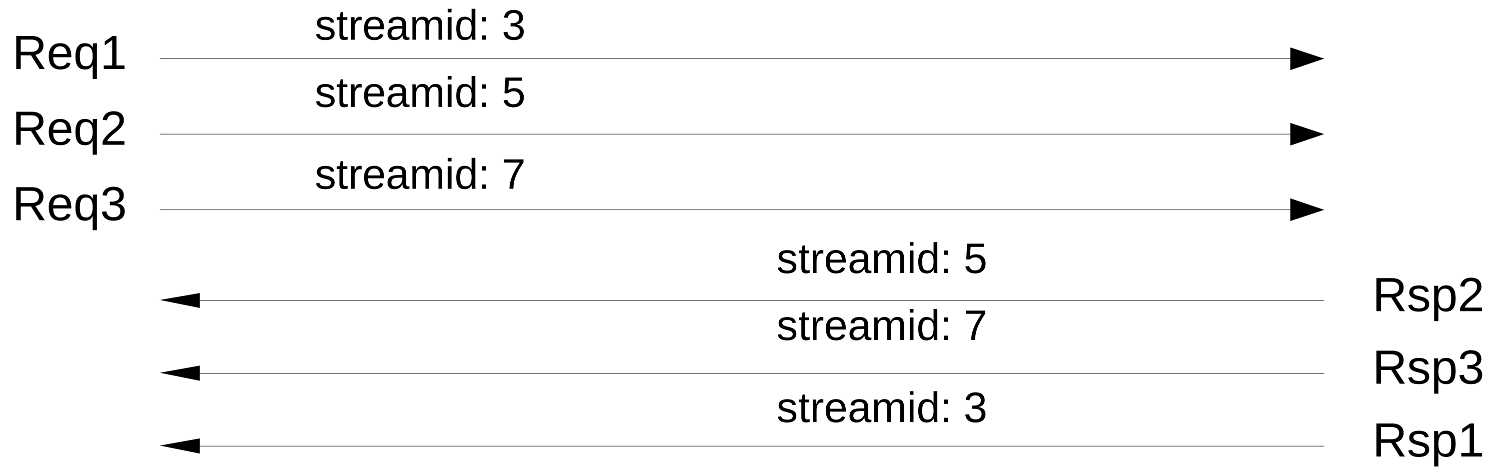
What if Rsp1 takes much longer to generate?

Rsp2, Rsp3 are blocked.

HTTP/2

What is different?

- Multiplexes multiple requests on same TCP connection
  - without head of line blocking problem that 1.1 pipelined requests have
- With HTTP/2: each request sent on own “stream”



Responses can be delivered in any order.

HTTP/2

What is different?

- Cancellation of requests
  - With 1.1 this is possible, but only at cost of closing TCP connection
  - explicit mechanism in 2 for resetting one stream without affecting others

HTTP/2

What is different?

- Cancellation of requests
  - With 1.1 this is possible, but only at cost of closing TCP connection
  - explicit mechanism in 2 for resetting one stream without affecting others
- Server push
  - only feature with noticeable API footprint
  - allows server to push resources directly into client's cache
  - pushes are always in context of a request initiated by client

HTTP/2

What else is different?

- Prioritisation of requests (relative to other requests)
- Ping
- Header compression (HPACK)
- Binary protocol. All information exchanged in Frames
  - HEADERS, DATA, PUSH_PROMISE, RST_STREAM, WINDOW_UPDATE

HTTP/2

Streams

- Multiplexes multiple requests on same TCP connection
- Stream id is 31 bit numeric identifier
  - odd numbered for client initiated, even for server initiated streams
- Each stream independently flow controlled (for data)
- Streams between any pair of endpoints assigned to same TCP connection
- Stream used for one request/response exchange only

HTTP/2

Protocol Example 1.1 vs 2

```
GET /resource HTTP/1.1                HEADERS
Host: example.org            ==>        + END_STREAM
Accept: image/jpeg                      + END_HEADERS
                                          :method = GET
                                          :scheme = https
                                          :path = /resource
                                          host = example.org
                                          accept = image/jpeg
```

HTTP/2

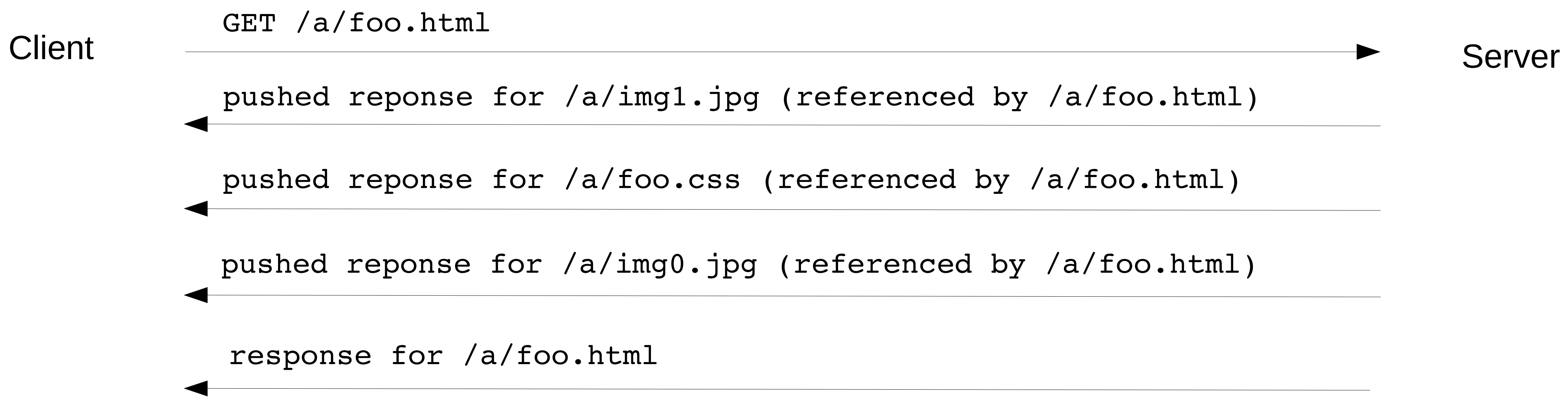
Protocol Example 1.1 vs 2

```
HTTP/1.1 200 OK                       HEADERS
Content-Type: image/jpeg   ==>          - END_STREAM
Content-Length: 123                     + END_HEADERS
                                          :status = 200
{binary data}                             content-type = image/jpeg
                                          content-length = 123

                                      DATA
                                        + END_STREAM
                                      {binary data}
```

HTTP/2

Server Push (high level)


Client
Server
GET /a/foo.html
pushed reponse for /a/img1.jpg (referenced by /a/foo.html)
pushed reponse for /a/foo.css (referenced by /a/foo.html)
pushed reponse for /a/img0.jpg (referenced by /a/foo.html)
response for /a/foo.html

HTTP/2

Starting HTTP/2: Upgrade from HTTP/1.1 (http urls)

```
GET /resource HTTP/1.1
Host: example.org
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url enc SETTINGS>
```

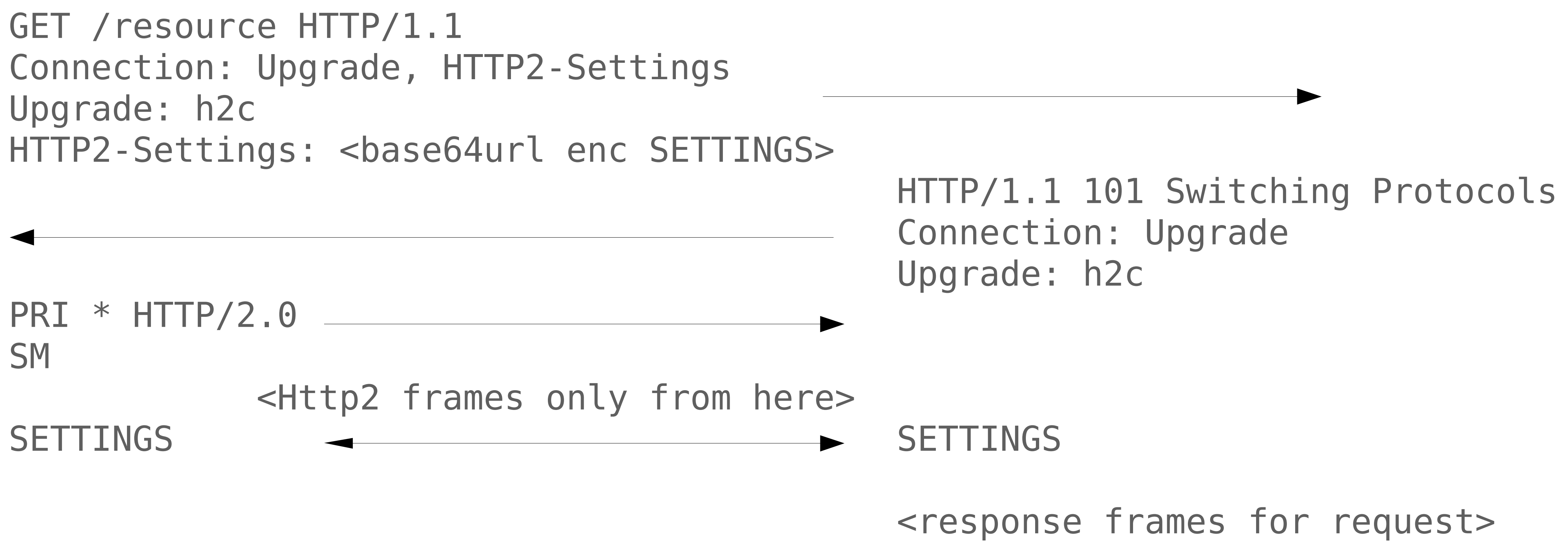
200 OK + response body for /resource

Server does not recognise HTTP/2: response in 1.1

:

HTTP/2

Starting HTTP/2: Upgrade from HTTP/1.1 (http urls)



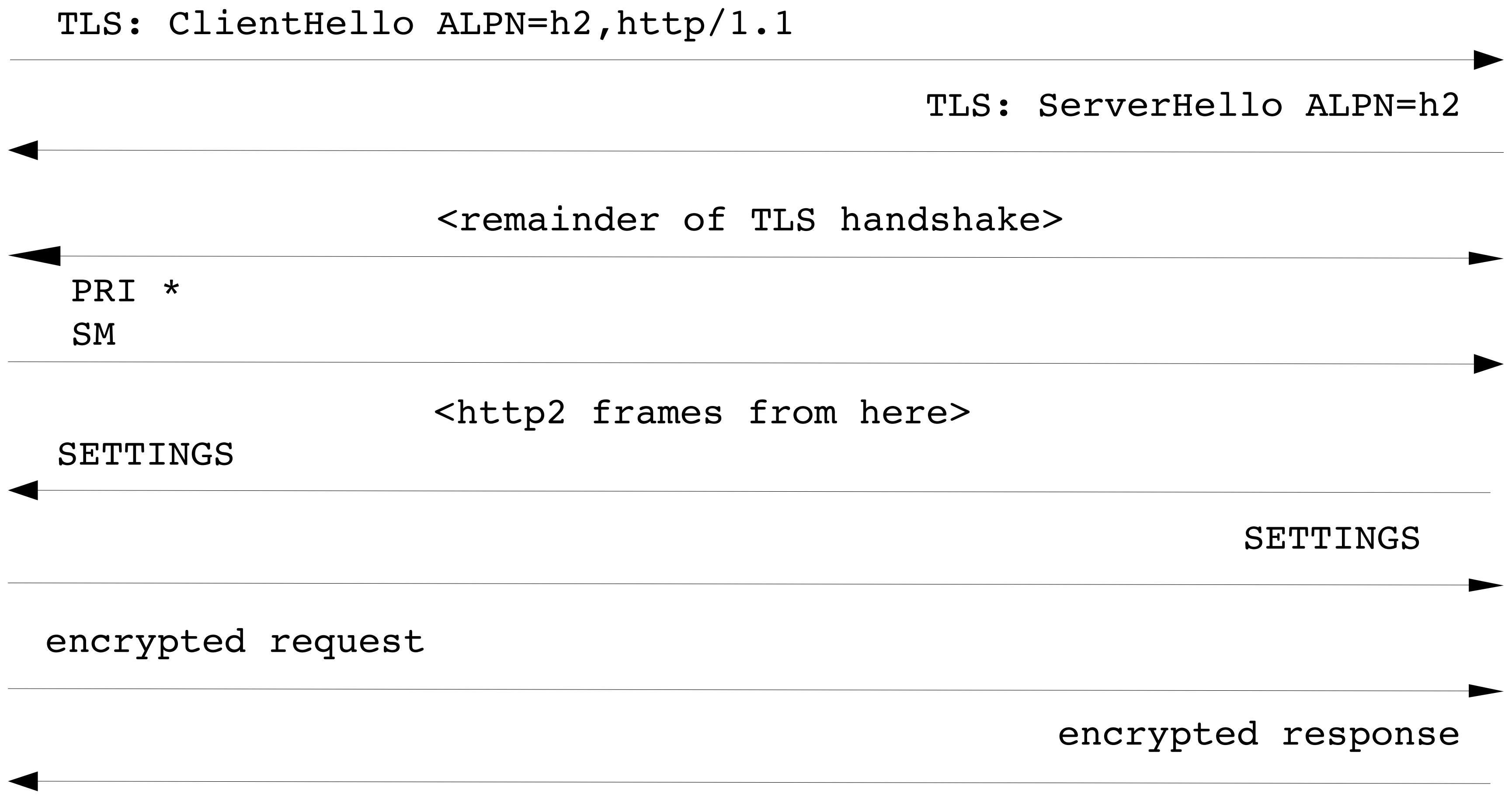
```
GET /resource HTTP/1.1
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url enc SETTINGS>
                                         HTTP/1.1 101 Switching Protocols
                                         Connection: Upgrade
                                         Upgrade: h2c
PRI * HTTP/2.0
SM
            <Http2 frames only from here>
SETTINGS                                 SETTINGS

                                         <response frames for request>
```

Server accepts HTTP/2

HTTP/2

Starting HTTP/2: for https urls



Program Agenda

1. HTTP/2 (similarities and differences from 1.1)
2. CompletableFuture API
3. New HTTP API
4. Web sockets

CompletableFuture API

- Framework for asynchronous programming:
  - in “Continuation” style
- Implements j.u.c.Future (from JDK 1.5) with:
  - standard API for completion (normally and exceptionally)
  - implementation of CompletionStage (provides continuation capability)
  - various static methods to:
    - create CF instances from various sources (Suppliers, Runnables, object instances)
    - create aggregate CF objects from multiple individual objects

CompletableFuture

Examples

```
CompletableFuture<Document> cf = load(new URI("http://foo.com/"));
```

CompletableFuture

Examples

```
CompletableFuture<Document> cf = load(new URI("http://foo.com/"));

Document doc = cf.get();
```

CompletableFuture

Example: composing multiple async operations

```
URI uri = URI.create("http://foo.com/");

CompletableFuture<Document> cf =
         load(uri);
```

CompletableFuture

Example: composing multiple async operations

```
URI uri = URI.create("http://foo.com/");

CompletableFuture<Image> cf =
         load(uri)
        .thenApplyAsync((Document doc) -> {
            return new Image(doc);
        });
```

CompletableFuture

Example: composing multiple async operations

```
URI uri = URI.create("http://foo.com/");

CompletableFuture<Void> cf =
         load(uri)
        .thenApplyAsync((Document doc) -> {
            return new Image(doc);
        })
        .thenAcceptAsync((Image image) -> {
            image.display();
        });
```

CompletableFuture

Example: composing multiple async operations

```
URI uri = URI.create("http://foo.com/");

CompletableFuture<Void> cf =
         load(uri)
        .thenApplyAsync((Document doc) -> {
            return new Image(doc);
        })
        .thenAcceptAsync((Image image) -> {
            image.display();
        });

cf.join();
```

CompletableFuture

Example implementation

```
ExecutorService executor;
      :
CompletableFuture<Document> load(URI uri) {
    CompletableFuture<Document> cf = new CompletableFuture<>();
    executor.execute(() → {

          Resource resource = new Resource(uri);
          Document doc = new Document(resource);
          cf.complete(doc);

    });
    return cf;
}
```

CompletableFuture

Example implementation

```
ExecutorService executor;
     :
CompletableFuture<Document> load(URI uri) {
    CompletableFuture<Document> cf = new CompletableFuture<>();
    executor.execute(() → {
      try {
          Resource resource = new Resource(uri);
          Document doc = new Document(resource);
          cf.complete(doc);
      } catch (Throwable t) {
          cf.completeExceptionally(t);
      }
    });
    return cf;
}
```

CompletableFuture

Example: wait for two concurrent loads to complete

```
CompletableFuture<Document> cf1 = load(new URI("http://foo.com/doc1"));
CompletableFuture<Document> cf2 = load(new URI("http://foo.com/doc2"));

CompletableFuture<Void> both = CompletableFuture.allOf(cf1, cf2);

both.join(); // Completes after both dependent CF's complete

Document doc1, doc1;
doc1 = cf1.get();
doc2 = cf2.get();
```

CompletableFuture

Example: wait for two loads then trigger dependent operation

```
// wait for two concurrent async operations, then merge the two results
:
CompletableFuture<Document> cf1 = load(new URI("http://foo.com/doc1"));
CompletableFuture<Document> cf2 = load(new URI("http://foo.com/doc2"));

CompletableFuture<Document> both =
    cf1.thenCombine(cf2, (doc1, doc2) -> {
        return doc1.merge(doc2);
    });

Document mergedDocument = both.join();
```

CompletableFuture

How are errors handled?

```
CompletableFuture<Document> cf1 = load(new URI("http://foo.com/doc1"));
CompletableFuture<Document> cf2 = load(new URI("http://foo.com/doc2"));

CompletableFuture<Document> both =
    cf1.thenCombine(cf2, (doc1, doc2) -> {
        return doc1.merge(doc2);
    });

Document mergedDocument = both.join(); // ←-- CompletionException thrown here
```

CompletableFuture

Example: return only first image to be loaded

```
CompletableFuture<Document> cf1 = load(new URI("http://foo.com/doc1"));
CompletableFuture<Document> cf2 = load(new URI("http://foo.com/doc2"));

CompletableFuture<Image> either =
    cf1.applyToEither(cf2, doc -> new Image(doc));

Image image = either.join();
```

CompletableFuture

Recap

- provides convenient single waiting point for multiple async operations
- provides multiple ways to combine/compose async operations

Program Agenda

1. HTTP/2 (similarities and differences from 1.1)
2. CompletableFuture API
3. **New HTTP API**
4. Web sockets

HTTP API

Overview

- Coming in Java SE 9
- In java.net.http package
- Simple API. Minimal classes are
  - HttpRequest(Builder), HttpClient(Builder), HttpResponse
- Supports HTTP/1.1 and HTTP/2
  - API is mostly agnostic about HTTP version

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET()
        .response();

if (resp.statusCode() == 200) {
    String responseBody = resp.body(HttpResponse.asString());
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/)) // → HttpRequest.Builder
        .GET()
        .response();

if (resp.statusCode() == 200) {
    String responseBody = resp.body(HttpResponse.asString());
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET() // → HttpRequest
        .response();

if (resp.statusCode() == 200) {
    String responseBody = resp.body(HttpResponse.asString());
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET()
        .response(); // → HttpResponse

if (resp.statusCode() == 200) {
    String responseBody = resp.body(HttpResponse.asString());
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET()
        .response();

if (resp.statusCode() == 200) {
    String responseBody = resp.body(HttpResponse.asString());
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET()
        .response();

if (resp.statusCode() == 200) {
    Path path = Paths.get("/path/file");
    Path responseBody = resp.body(HttpResponse.asFile(path));
    System.out.println(responseBody);
}
```

HTTP API

Example

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .GET()
        .response();

if (resp.statusCode() == 200) {
    byte[] responseBody = resp.body(HttpResponse.asByteArray());
    System.out.println(responseBody.length);
}
```

HTTP API

Example

```
try {
    HttpResponse resp = HttpRequest
            .create(new URI(“http://www.foo.com/))
            .GET()
            .response();

    if (resp.statusCode() == 200) {
        byte[] responseBody = resp.body(HttpResponse.asByteArray());
        System.out.println(responseBody.length);
    }
} catch (IOException | InterruptedException e) {
    ...
}
```

HTTP API

Example POST with a request body

```
HttpResponse resp = HttpRequest
        .create(new URI(“http://www.foo.com/))
        .body(HttpRequest.fromString(“param1=1,param2=2”))
        .POST()
        .response();
```

HTTP API

Asynchronous example

```
HttpResponse resp =
           HttpRequest.create(uri)
              .GET()
              .response();
```

HTTP API

Asynchronous example

```
CompletableFuture<HttpResponse> cf =
          HttpRequest.create(uri)
             .GET()
             .responseAsync();

HttpResponse resp = cf.join();
```

HTTP API

Asynchronous example

```
HttpRequest.create(uri)
   .GET()
   .responseAsync()
   .thenApplyAsync((HttpResponse resp) -> {
       if (resp.statusCode() == 200) {
           return resp.body(asFile(path));
       } else {
           throw new UncheckedIOException(new IOException());
       }
   });
```

HTTP API

Asynchronous example

```
HttpRequest.create(uri)
    .GET()
    .responseAsync()
    .thenApplyAsync((HttpResponse resp) -> {
        if (resp.statusCode() == 200) {
            return resp.body(asFile(path));
        } else {
            throw new UncheckedIOException(new IOException());
        }
    }); // output is CompletableFuture<Path>
```

HTTP API

Asynchronous example

```
CompletableFuture<Path> fetchAsync(URI uri) {
    Path p = Paths.get(“/someroot”, uri.getPath());
    return HttpRequest.create(uri)
              .GET()
              .responseAsync()
              .thenApplyAsync((HttpResponse resp) -> {
                  if (resp.statusCode() == 200) {
                      return resp.body(asFile(path));
                  } else {
                      throw new UncheckedIOException(new IOException());
                  }
              });
}
```

HTTP API

Asynchronous example: fetch list of URIs concurrently

```
List<URI> uris = ...;
List<CompletableFuture<Path>> cfs = new LinkedList<>();

uris.stream()
    .forEach((URI uri) -> {
        cfs.add(fetchAsync(uri));
     });
```

HTTP API

Asynchronous example: fetch list of URIs concurrently

```
List<URI> uris = ...;
List<CompletableFuture<Path>> cfs = new LinkedList<>();

uris.stream()
    .forEach((URI uri) -> {
        cfs.add(fetchAsync(uri));
     });

// all requests initiated here. Wait for all to complete

CompletableFuture.allOf(cfs.toArray(emptyArray))
        .join();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParams)
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParams)
        .proxy(ProxySelector.of(new InetSocketAddress("proxy", 80)))
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParams)
        .proxy(ProxySelector.of(new InetSocketAddress("proxy", 80)))
        .executorService(Executors.newCachedThreadPool())
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParams)
        .proxy(ProxySelector.of(new InetSocketAddress("proxy", 80)))
        .executorService(Executors.newCachedThreadPool())
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .build();
```

HTTP API

HttpClient configuration

```
HttpClient client = HttpClient.create()
        .authenticator(someAuthenticator)
        .sslContext(someSSLContext)
        .sslParameters(someSSLParams)
        .proxy(ProxySelector.of(new InetSocketAddress("proxy", 80)))
        .executorService(Executors.newCachedThreadPool())
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .cookieManager(someCookieManager)
        .build();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromString("param1=val1,param2=val2"))
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromFile(Paths.get("/path/file")))
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromInputStream(someInputStream)))
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromByteArray(new Byte[] {1,2,3}))
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromString("param1=val1,param2=val2"))
        .header("X-Foo", "foo")
        .header("X-Bar", "bar")
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromString("param1=val1,param2=val2"))
        .header("X-Foo", "foo")
        .header("X-Bar", "bar")
        .timeout(TimeUnit.SECONDS, 5)
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromString("param1=val1,param2=val2"))
        .header("X-Foo", "foo")
        .header("X-Bar", "bar")
        .timeout(TimeUnit.SECONDS, 5)
        .version(HttpClient.Version.HTTP_2)
        .POST();

HttpResponse response = request.response();
```

HTTP API

Building HttpRequests

```
HttpClient client = HttpClient.create()
        .build();

HttpRequest request = client.request(URI.create("https://www.foo.com/"))
        .body(fromString("param1=val1,param2=val2"))
        .header("X-Foo", "foo")
        .header("X-Bar", "bar")
        .timeout(TimeUnit.SECONDS, 5)
        .version(HttpClient.Version.HTTP_2)
        .POST();

HttpResponse response = request.response();
```

HTTP API

Examining HttpResponses

```
public abstract class HttpResponse {
    int statusCode();
    URI uri();
    HttpRequest request();
    SSLParameters sslParameters();
    HttpClient.Version version();
    HttpHeaders headers();
    HttpHeaders trailers();
    <T> T body(BodyProcessor<T> processor);
    <T> CompletableFuture<T> bodyAsync(BodyProcessor<T> processor);
}
```

HTTP API

Examining HttpResponses

```
public abstract class HttpResponse {
        :
    HttpHeaders headers();
    HttpHeaders trailers();
}

public interface HttpHeaders {
    Optional<String> firstValue(String name);
    Optional<Long> firstValueAsLong(String name);
    List<String> allValues(String name);
    Map<String,List<String>> map();
}
```

HTTP API

Examining HttpResponses headers (Optional usage)

```
public interface HttpHeaders {
    Optional<String> firstValue(String name);
    Optional<Long> firstValueAsLong(String name);
    List<String> allValues(String name);
    Map<String,List<String>> map();
}

{
    String contentType = headers
                .firstValue("Content-type")
                .orElseThrow(() -> new IOException("Expected content-type"));
}
```

HTTP API

Examining HttpResponse headers (Optional usage)

```
public interface HttpHeaders {
    Optional<String> firstValue(String name);
    Optional<Long> firstValueAsLong(String name);
    List<String> allValues(String name);
    Map<String,List<String>> map();
}

{
    String contentType = headers
                .firstValue("Content-type")
                .orElseThrow(() -> new IOException("Expected content-type"));

    String y = headers.firstValue("X-Foo").orElse("defaultfooValue");
}
```

HTTP API

List of standard HttpRequest.BodyProcessor

```
HttpRequest.BodyProcessor fromString(String body);

HttpRequest.BodyProcessor fromFile(Path path);

HttpRequest.BodyProcessor fromString(String s, Charset charset);

HttpRequest.BodyProcessor fromByteArray(byte[] buf);

HttpRequest.BodyProcessor fromByteArray(byte[] buf, int offset, int length);

HttpRequest.BodyProcessor fromByteArrays(Iterator<byte[]> iter);

HttpRequest.BodyProcessor fromInputStream(InputStream stream);

HttpRequest.BodyProcessor noBody();
```

HTTP API

List of standard response body processors

```
BodyProcessor<Path> asFile(Path file)

BodyProcessor<Path> asFileDownload(Path directory, OpenOption... openOptions)

BodyProcessor<Path> asFile(Path file, OpenOption... openOptions)

BodyProcessor<Void> asByteArrayConsumer(Consumer<byte[]> consumer)

BodyProcessor<InputStream> asInputStream()

BodyProcessor<byte[]> asByteArray()

BodyProcessor<String> asString()

BodyProcessor<String> asString(Charset charset)
```

HTTP API

Server Push API: Multi responses

```
public abstract class HttpRequest {

    HttpResponse response();

    CompletableFuture<HttpResponse> responseAsync();

    <U> CompletableFuture<U>
             multiResponseAsync(HttpResponse.MultiProcessor<U> rspproc);
}
```

HTTP API

Server Push API: Multi responses

```
    public abstract class HttpRequest {

        HttpResponse response();

        CompletableFuture<HttpResponse> responseAsync();

        <U> CompletableFuture<U>
                  multiResponseAsync(HttpResponse.MultiProcessor<U> rspproc);
    }

    public abstract class HttpResponse {
           :
         MultiProcessor<Map<URI,Path>> multiFile(Path destination);
    }
```

HTTP API

Server Push API: Multi responses

```
CompletableFuture<Map<URI,Path>> cf =
    HttpRequest.create(new URI("https://www.foo.com/"))
           .version(Version.HTTP_2)
           .GET()
           .multiResponseAsync(HttpResponse.multiFile("/usr/destination"));

Map<URI,Path> results = cf.join();
```

HTTP API

Server Push API: Multi responses

```
CompletableFuture<Map<URI,Path>> cf =
    HttpRequest.create(new URI("https://www.foo.com/"))
           .version(Version.HTTP_2)
           .GET()
           .multiResponseAsync(HttpResponse.multiFile("/usr/destination"));

Map<URI,Path> results = cf.join();
```

Program Agenda

1. HTTP/2 (similarities and differences from 1.1)
2. CompletableFuture API
3. New HTTP API
4. **Web sockets API**

Websocket API

Overview

- Coming in Java SE 9
- Simple API.
  - in java.net.http package
  - WebSocket, WebSocket.Builder, WebSocket.Listener, WebSocketException
- API still evolving

Websocket

Builder class

```
public static class WebSocket.Builder {
    public Builder(String uri);
    public Builder(String uri, HttpClient client);
    public Builder header(String name, String value);
    public Builder subprotocols(String mostPreferred,
                                String... lesserPreferred)
    public Builder connectTimeout(long timeout, TimeUnit unit);
    public Builder listenWith(Listener listener);
    public Builder listenWith(DecodedTextListener listener);
    public CompletableFuture<WebSocket> buildAsync();
}
```

Websocket

Listener interface

```
public interface Listener {
    default void onOpen(WebSocket webSocket);
    default CompletableFuture<?> onText(WebSocket webSocket,
                                        ByteBuffer payload,
                                        boolean isLast);
    default CompletableFuture<?> onBinary(WebSocket webSocket,
                                          ByteBuffer payload,
                                          boolean isLast);
    default CompletableFuture<?> onPing(WebSocket webSocket,
                                        ByteBuffer payload);
    default CompletableFuture<?> onPong(WebSocket webSocket,
                                        ByteBuffer payload);
    default void onClose(WebSocket webSocket, Optional<ClosureCode> code,
                         String reason);
    default void onError(WebSocket webSocket, Throwable error);
}
```

Websocket

Websocket class

```
class WebSocket {
    CompletableFuture<Void> sendText(ByteBuffer payload, boolean isLast);
    CompletableFuture<Void> sendText(CharSequence payload, boolean isLast);
    CompletableFuture<Void> sendText(CharSequence payload);

    CompletableFuture<Void> sendText(Stream<? extends CharSequence> payloadSource);
    CompletableFuture<Void> sendBinary(ByteBuffer payload, boolean isLast);
    CompletableFuture<Void> sendBinary(byte[] payload, boolean isLast);
    CompletableFuture<Void> ping(ByteBuffer payload);
    CompletableFuture<Void> pong(ByteBuffer payload);
    CompletableFuture<Void> sendClose(ClosureCode code, String reason);
    CompletableFuture<Void> sendClose();
    void request(long n);
    String getSubprotocol();
    boolean isClosed();
    void abort() throws IOException;
}
```

Websocket

Example creation with HttpClient

```
HttpClient client = HttpClient.create()
        .proxy(ProxySelector.of(new InetSocketAddress("proxy.example.com", 80)))
        .build();

CompletableFuture<WebSocket> f =
    new WebSocket.Builder("ws://websocket.example.com", client)
        .subprotocols("proto")
        .header("Foo", "foovalue")
        .header("Bar", "barvalue")
        .buildAsync();

f.join();
```

Q & A

JavaOne™

ORACLE®