Messaging

Scenario

Video format conversion is slow: so when should it be performed?

Scenario

Video format conversion is slow: so when should it be performed?

- Before playback
  - *Problem: viewers will need to wait*
- At the end of an upload
  - *Problem: uploader will need to wait*

Scenario

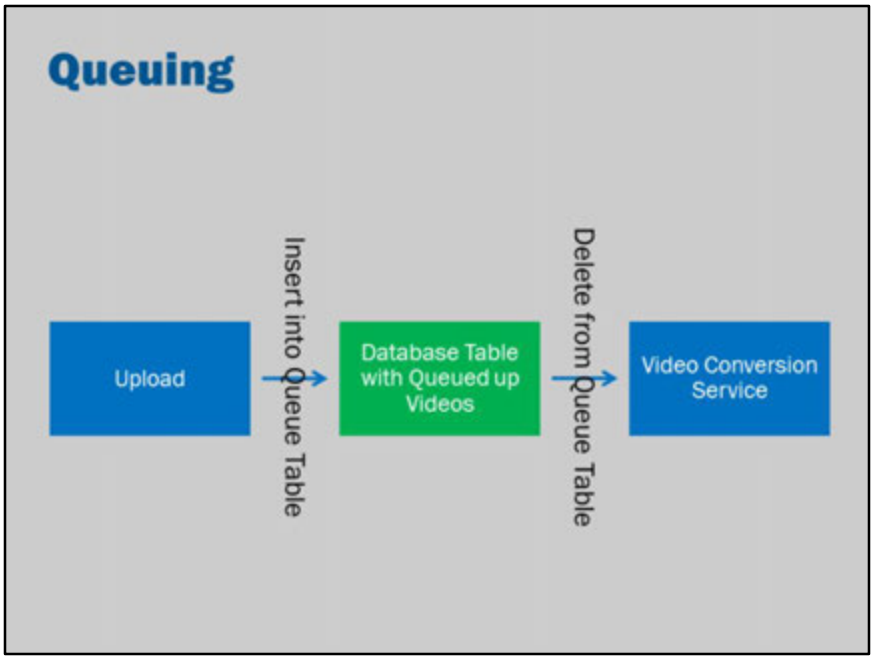
Video format conversion is slow: so when should it be performed?

- Before playback
  - *Problem: viewers will need to wait*
- At the end of an upload
  - *Problem: uploader will need to wait*
- On a separate thread after upload
  - *Problem: lots of threads could slow down the server*

Scenario

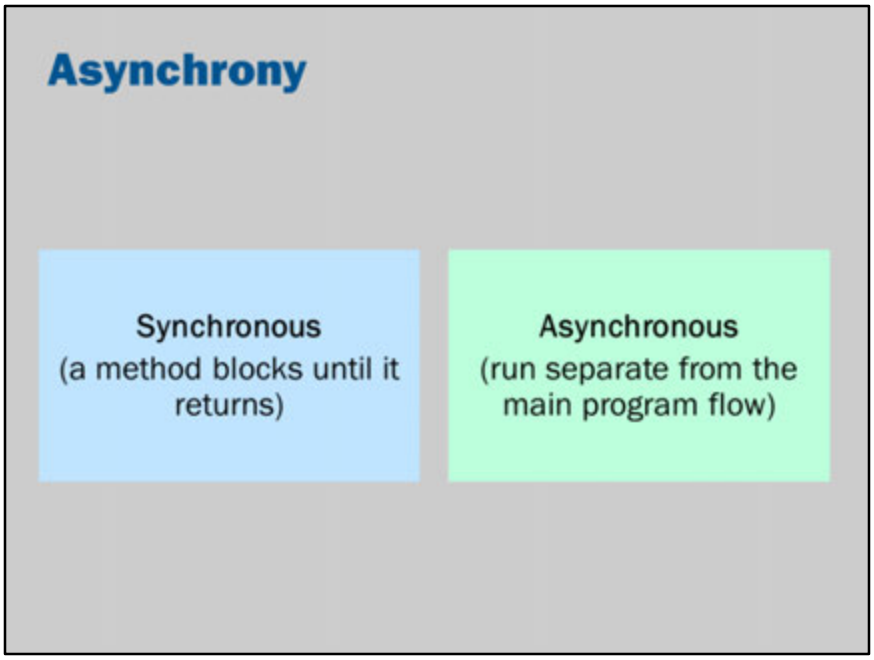
Video format conversion is slow: so when should it be performed?

- Before playback
  - *Problem: viewers will need to wait*
- At the end of an upload
  - *Problem: uploader will need to wait*
- On a separate thread after upload
  - *Problem: lots of threads could slow down the server*
- Keep a queue of videos to convert



A simple approach to building a queue would be to insert a task into a database table during from one process (e.g., after upload).
Then, have a separate process scan the table looking for entries, deleting from the table whenever it completes a job.

If there is a sudden rush of new items added to the table, then they will just queue up and the other process will gradually work through the backlog one-by-one.



In computer programming synchronous means that your code is executed within the ordinary flow of control.

Asynchronous means that your code is executed outside the ordinary flow of control. Control might be returned immediately.
The code will run at a later time, on a different thread.

Message driven beans are one way that this can be achieved in Java EE.

Sending

```
@Resource(lookup="jms/AIPConnectionFactory")
private ConnectionFactory factory;

@Resource(lookup="jms/aip")
private Queue queue;

public void send() throws Exception {
  // Set up
  Connection conn = factory.createConnection();
  Session sess = conn.createSession();
  MessageProducer prod = sess.createProducer(queue);

  // Send
  TextMessage message = session.createTextMessage();
  message.setText("Hello!");
  prod.send(message);
}
```

In this code, a connection to Java Message Service (JMS) is obtained, a session created and a message delivered to a queue.

Message driven bean

```
@MessageDriven(mappedName = "jms/aip")
public class MyMDBean implements MessageListener {

  @Override
  public void onMessage(Message message) {
    try {
      String body = message.getBody(String.class);
      System.out.println("Received: " + body);
    } catch (JMSException e) {
      // handle exception
    }
  }

}
```

Java EE will deliver the messages to the message driven bean, as the messages arrive.

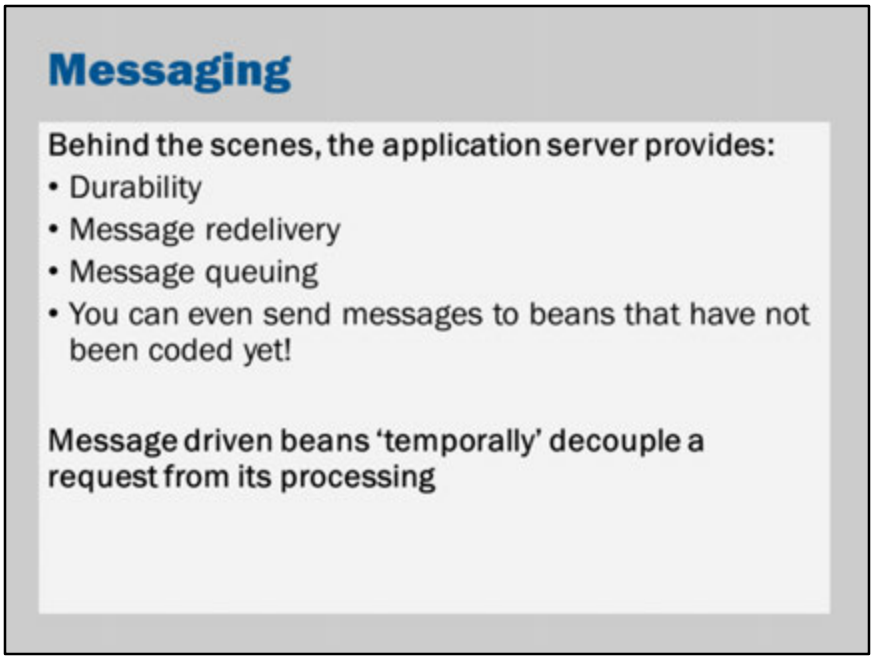
In the previous slide, the queue was injected like this:

```
@Resource(lookup="jms/aip")
private Queue queue;
```

This message driven bean listens to the same queue:

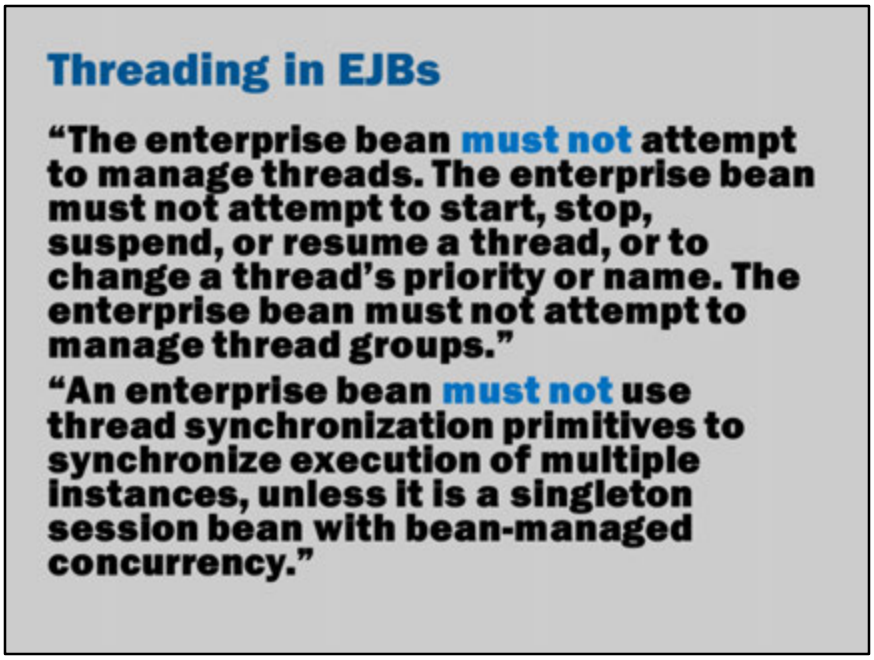
```
@MessageDriven(mappedName = "jms/aip")
```

Note, though that the queue also needs to be configured in your application server administration.



Durability: Once accepted, a message will not get lost when the Java EE server stops or loses power suddenly. It is only removed from the queues once it has been successfully processed.
Message redelivery: If the message driven bean fails during processing, the container will retry delivery so that the message driven bean can try again.
Message queuing: If the message processing is too slow, the messages will queue up until the message driven bean(s) are able to process it.

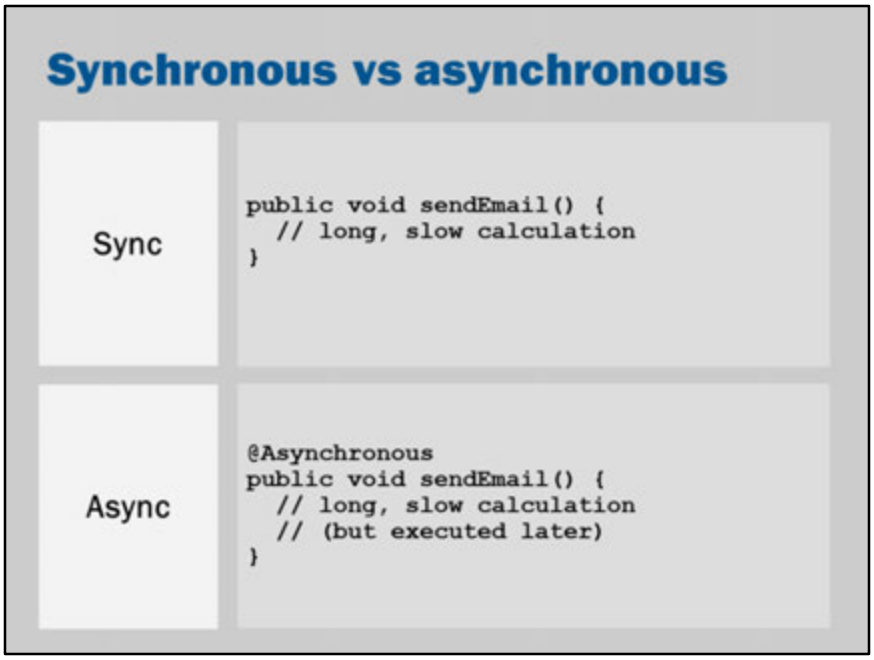
Asynchrony



In ordinary Java code, you might use threads when you want to execute two (or more) things at once.

In Java EE, you must not use threads.

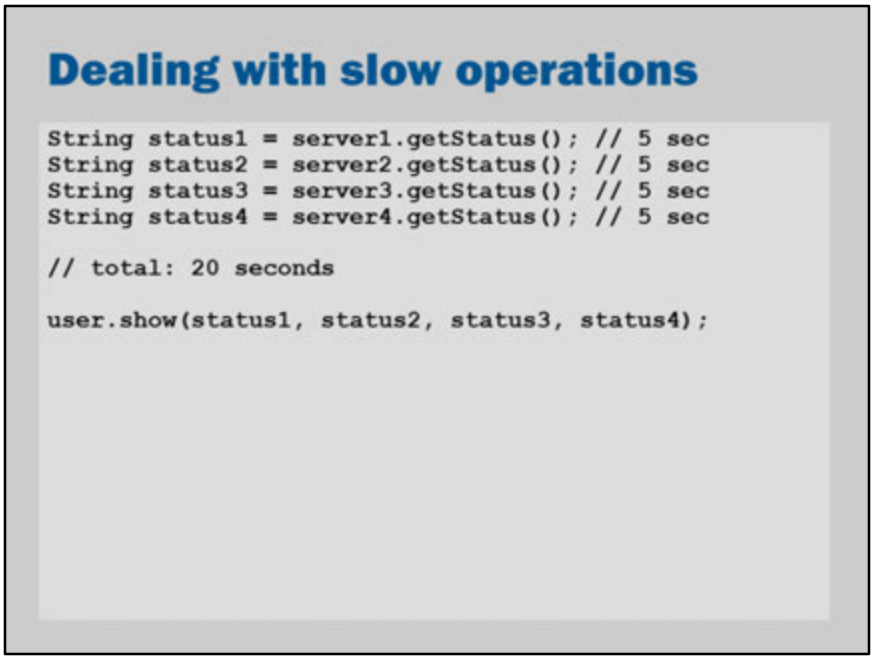
The quote in the slide comes from Section 16.2.2 of the EJB specification.



For information about Asynchronous methods, see this Wikipedia article:
http://en.wikipedia.org/wiki/Asynchronous_method_invocation

In Java EE, when you call a method that has been annotated with Asynchronous, it will return immediately.
However, the method will continue running in a separate thread.

Normally if you call a *synchronous* method, control does not return until that method is complete.
An *asynchronous* method will return immediately and it will continue processing separately.

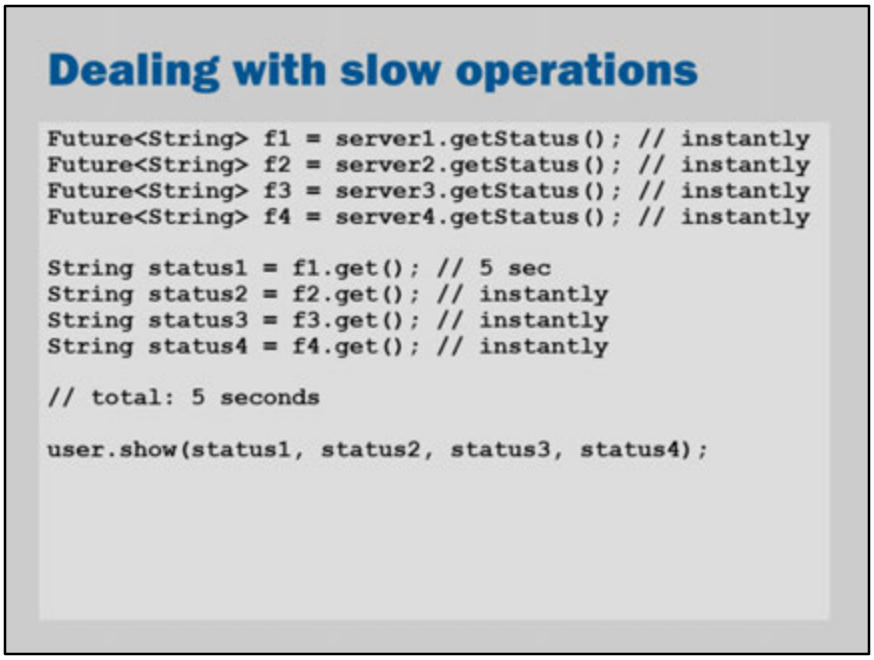


Suppose you have some ordinary (synchronous) Java code.

Each call to getStatus contacts a remote server and takes 5 seconds.

There are four servers, so executing them one-by-one will therefore take 20 seconds in total.

This would be a perfect situation for using threads.
You can't use threads in Java EE, so we use @Asynchronous.
However, in this case, we need to get the result.
So to do this we use futures…



…. a future is a "promise" to return a value at a future time.

https://en.wikipedia.org/wiki/Futures_and_promises

In this code, we get four promises the getStatus methods.
The method returns immediately but it doesn't return the real value.
Instead, it returns a "promise" to deliver a value in the future.
The promise is of type Future<String>, so this means that when we call .get() on the promise, it will return a String.

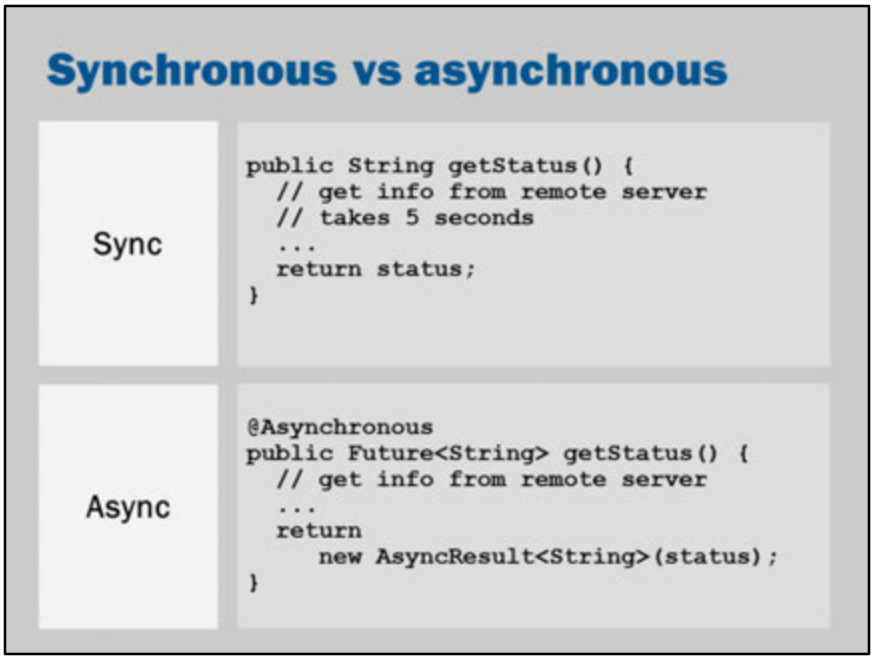
Calling each of the getStatus methods will cause four separate threads to start, each of them contacting the remote server.

Calling .get() on the Future will cause the code to block, awaiting the result from the thread that has been launched.

This code will get to the first .get() almost instantaneously.

The code will then block while f1.get() waits for the remote server to response (5 seconds).
Since five seconds has elapsed at that line of code, the other servers should also have responded in the meantime.
This means that the remaining calls to f2.get(), f3.get() and f4.get() should be almost instantaneous.

Thus, the total execution time is just 5 seconds.
Previously it was 20 seconds, so we've saved 15 seconds in total (note, however, that this caused 4 separate threads to be launched).

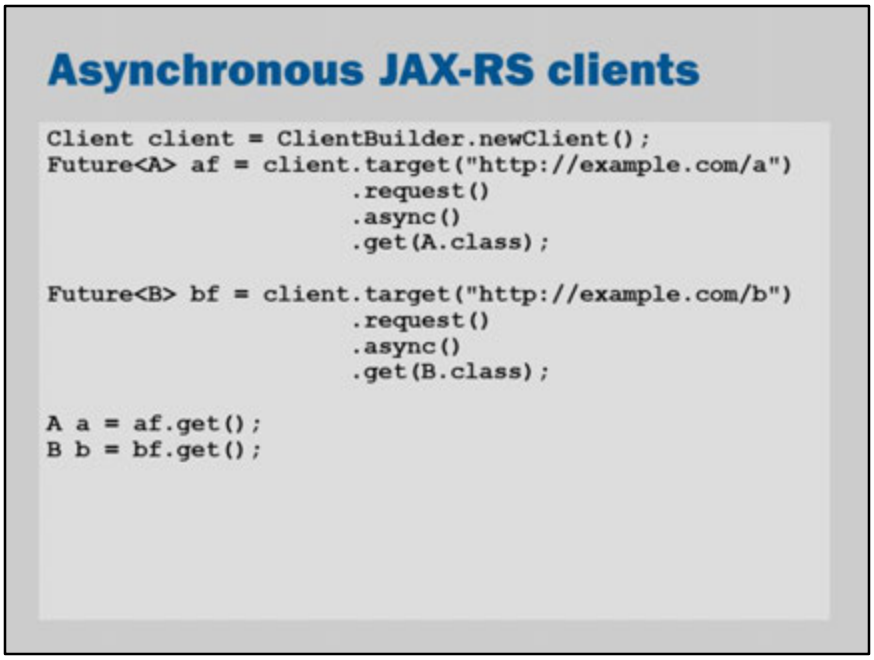


This code is what it looks like to get an asynchronous method to return a value.
That is, this code shows how to return a future.

A Future<> object is returned immediately, but the function continues to run separately.

The client can keep checking if the result is available (using Future.isDone()), or it can block until the task is complete (Future.get()).
That is, it can do some other work while waiting for the EJB to calculate the result.

See the following for further details:
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html
http://tomee.apache.org/examples-trunk/async-methods/README.html
http://docs.oracle.com/javaee/7/tutorial/doc/ejb-async001.htm



The Java EE specifications are gradually adding more-and-more support for asynchrony and non-blocking I/O.

In the example shown in the slide, we might have a web site that makes use of two external web services using JAX-RS.

This is what the synchronous code would look like:

```
Client client = ClientBuilder.newClient();
A a = client.target("http://example.com/a")
        .request()
        .get(A.class);

B b = client.target("http://example.com/b")
        .request()
        .get(B.class);
```

Suppose that each request takes 10 seconds.

The entire operation will then take 20 seconds (it does one, then the other).

If, instead, we use the asynchronous mode of the JAX-RS client, then the request returns an instance of Future.
The JAX-RS client will handle the request asynchronously, allowing other processing to be done in the mean time.

So, in the example of the slides, we will get *af* immediately and then also get *bf* immediately.
When we call af.get(), the Future will block for 10 seconds while the operation is processed.

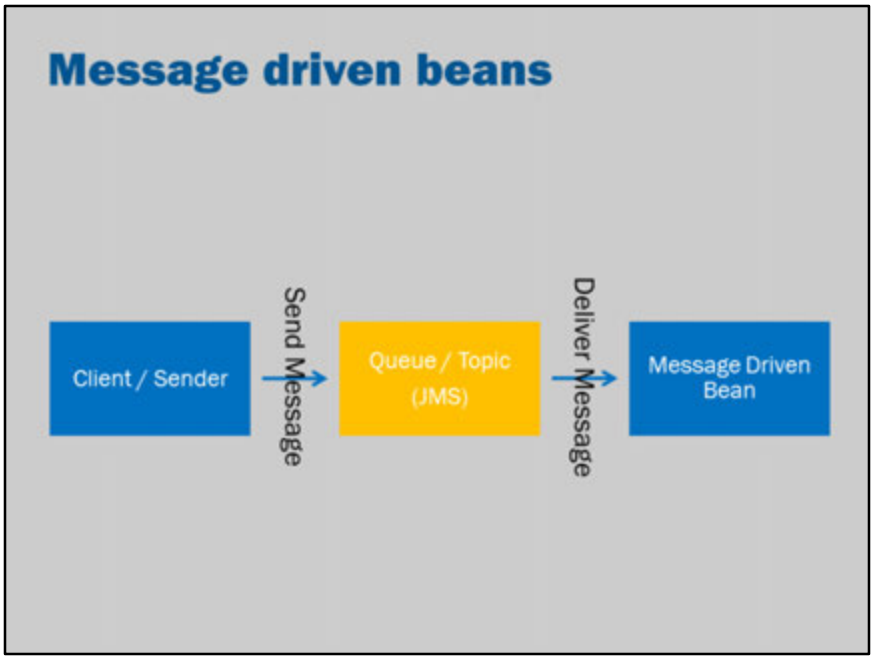
When we call bf.get(), the second response will already have been processed asynchronously for 10 seconds. So bf.get() should return more-or-less immediately.

Thus, the total amount of time taken to perform both operations is likely to be on the order of 10 seconds (i.e., half the time of performing the two operations synchronously).

Key points

- Do not use threads or `synchronized` in an Enterprise Java Bean
- Use `@Asynchronous` if you need concurrency (this will return a `Future` which you can use to retrieve the result)
- Many Java EE technologies are making increased use of asynchronous methods and interfaces

Bonus slides



Messaging decouples invocation from execution.
A client can send a message. It is queued by Java EE (i.e., JMS middleware). The message driven bean that processes the message can access the information at its own rate.

The queue will survive shutdown.
In fact, it is possible to send messages even if the components are incomplete.
Messaging makes it possible for the following hypothetical scenario to occure:

1. Write a client (but not the message driven bean)
2. Send a message (it gets queued up)
3. Shutdown GlassFish
4. Restart GlassFish
5. Write the message driven bean
6. Delete the client
7. Redeploy the application
8. The queued message will then be delivered to the message driven bean
9. If the message-driven bean fails, delivery will be reattempted until it succeeds.

This kind of technology can be very effective in large systems.
When systems are decoupled through messaging, the failure of one system does not affect the failure of another system.
For example, if the billing system is down, the website should still keep working.

As an aside: message oriented programming is also very effective in robotics. This is because robots are very prone to failure. If something fails, you don't want the entire robot to stop. If something is running slow, you don't want the entire robot to slow down.

Future

```
public interface Future<V> {

  // Attempt to cancel the task
  boolean cancel(boolean mayInterruptIfRunning);

  // Was the task cancelled before it completed?
  boolean isCancelled();

  // Is the task complete?
  boolean isDone();

  // Wait for the task to complete
  V get() throws InterruptedException,
   ExecutionException;

  // Wait, up to a given duration, for the task to
   complete
  V get(long timeout, TimeUnit unit) throws ...;
}
```

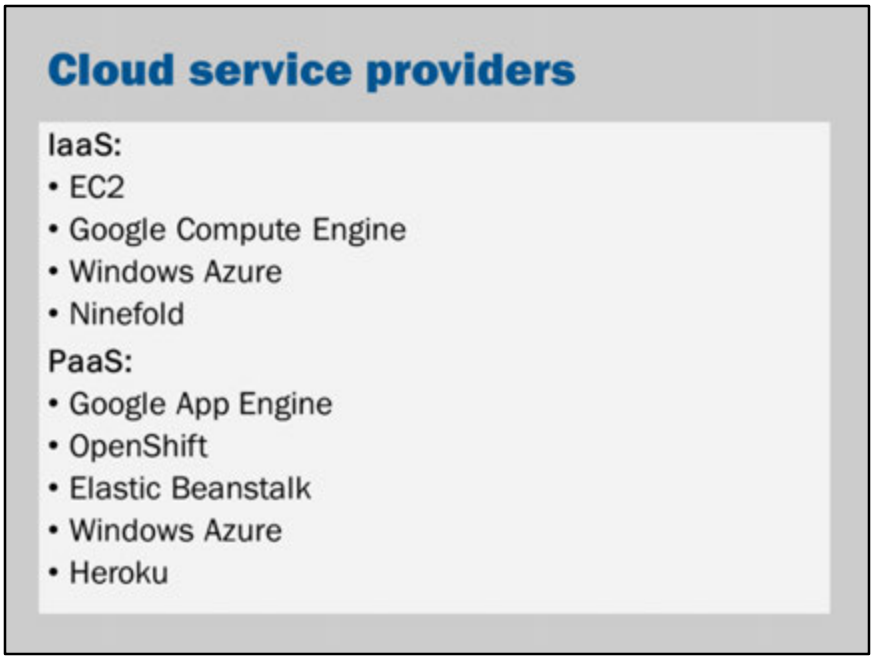
This is an abbreviated version of the Future interface.
It allows you to check on the status of an asynchronous task, get the result and wait for a result.

Trends

Outline

- Hosting
- NoSQL
- MVC and lightweight frameworks
- Asynchrony

Hosting



There are a number of hosting options.
Some provide managed hosting where you can upload a WAR/EAR file.
Others provide a more basic computer-by-the-hour service.

Here's how you might use Amazon's EC2:

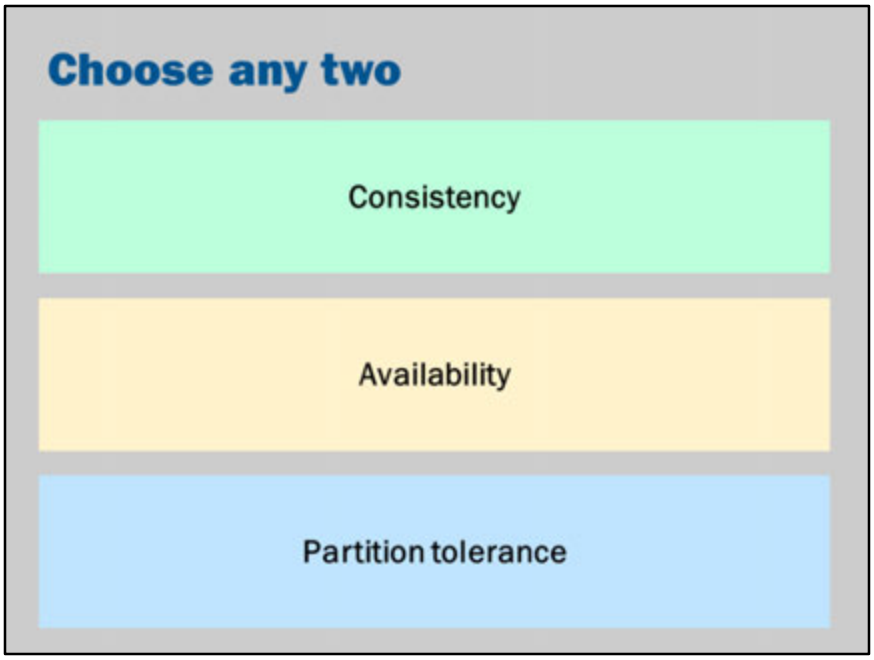
1. Log into the Amazon Web Services Console and enter the EC2 section
2. Launch a new instance
3. Get a security key (private key)
4. Convert the key to a format usable by Putty (only needed under Windows)
5. SSH into the new server
6. Follow the steps here:
   https://www.digitalocean.com/community/tutorials/how-to-install-glassfish-4-0-on-ubuntu-12-04-3
7. Go back to the Amazon Console and enable port 8080 on the Firewall

The entire process can be done in less than 10 minutes (though, it will take longer the first time you do it).

GlassFish can be reconfigured to host directly on port 80. It is also possible to use a

reverse-proxy such as Nginx as a front-end.

NoSQL



While a database on a single computer may be appropriate for small websites, eventually a system with enough users will need a database running on multiple computers.

Ideally, a distributed system will:
- Remain consistent: it will work just like a single system
- Have high availability: it should keep working even if parts of the system fail
- Be able to keep working or remain consistent, even if the network splits in two (e.g., if you have 10 computers in Australia and 10 in the USA, it should still keep working even if the connection between Australian and USA stopped working)

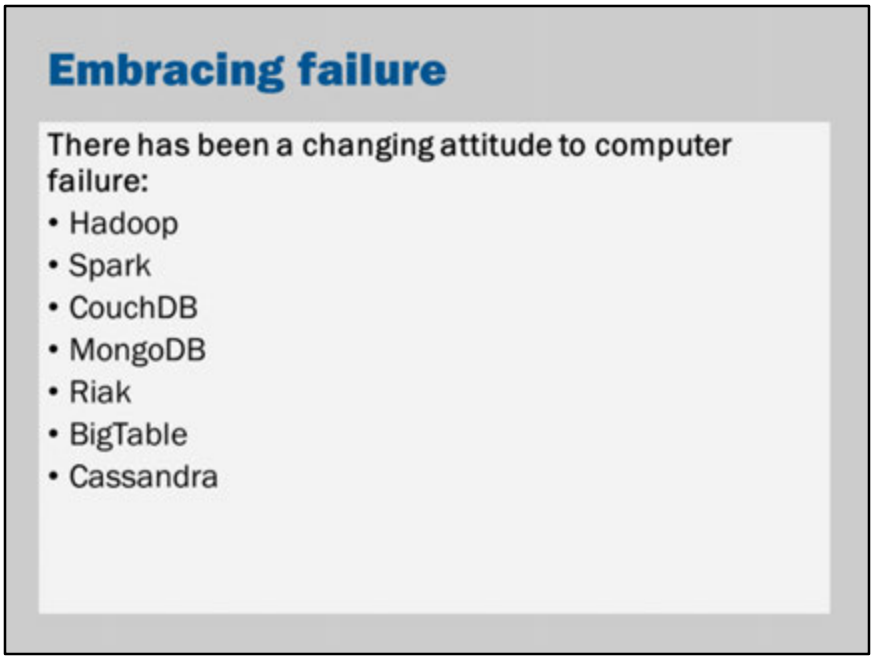
An easy way to ensure consistency and partition tolerance is to just have one computer. However, if that one computer crashes, it will no longer remain available.

Conversely, an easy way to ensure availability, is to run the same database on multiple computers and perform the same operations on every computer. If any computer crashes, you can keep communicating with the other databases. However, this may result in inconsistencies if the same update isn't sent to every machine.

Is it possible to have all three at once?

In 2002, the "CAP theorem" was proven:
http://en.wikipedia.org/wiki/CAP_theorem

The CAP theorem says that you can't have all three. You can only choose two.



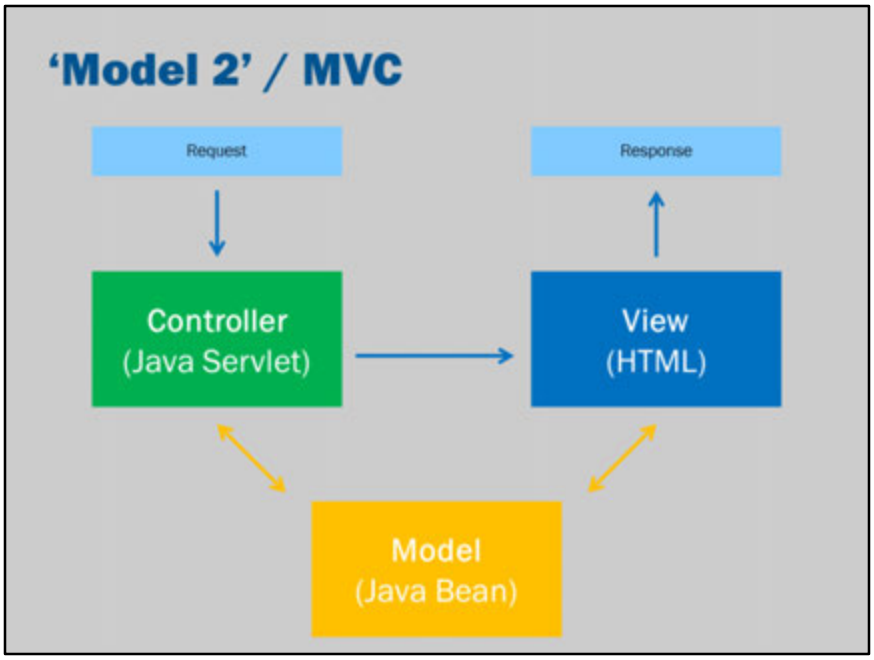
SQL databases have traditionally prioritized consistency.
NoSQL databases have, instead, focused on performance and availability at the price of consistency.

In recent years, there have been a range of technologies that 'embrace' failure.

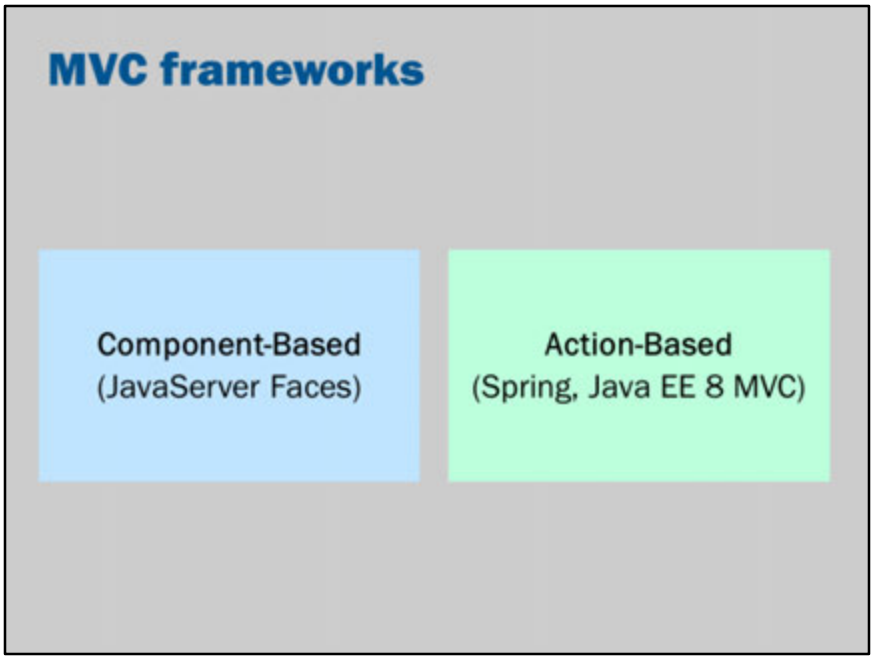
Many of these databases are either key-value stores or document-based stores. They're designed for use in web development and so often have JavaScript-based representations, protocols and/or data-types.

Hadoop and Spark probably shouldn't be on this list… but they are examples of other systems that embrace failure. They aren't database systems but they're designed for robust, high speed computation. A large computation is spread over a network. If something fails, the system automatically recovers and re-does the computation on another node.

MVC frameworks



Recall the MVC architecture from the Week 3 lecture. If you can't explain how the MVC framework relates to JavaServer Faces, I encourage you to revisit the lecture.



There are two approaches to MVC architectures.

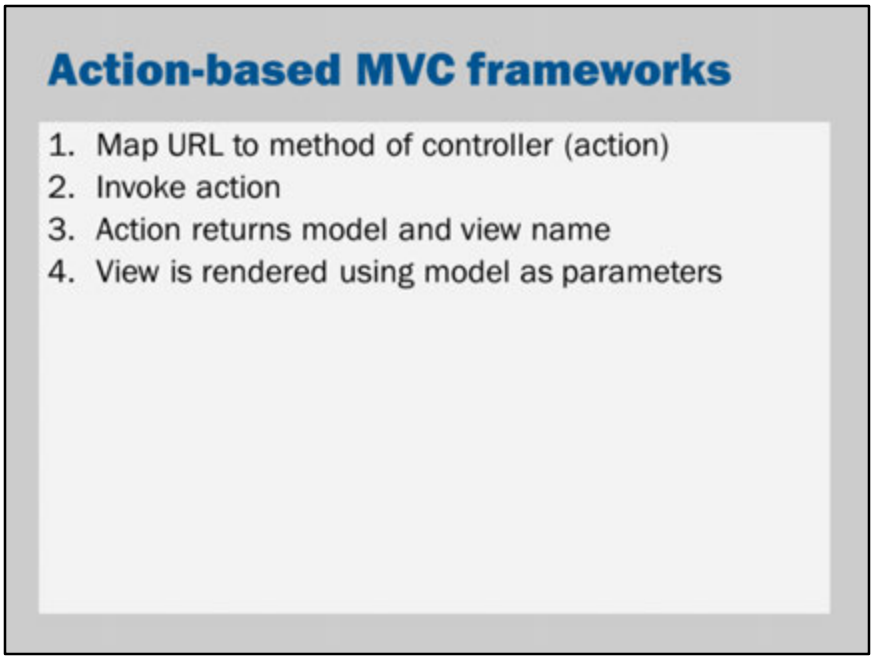
JavaServer Faces uses a component-based architecture. This is a very powerful approach. It attempts to simulate traditional application development when creating websites. It uses powerful components that are responsible for generating user interface and processing the input. This power and complexity makes it possible to create highly functional websites in little time. However, the downside is that the complexity can be difficult to use, especially when something goes wrong.

Action-based frameworks, in contrast, emphasize simplicity. An incoming request is mapped to an action and then that action is responsible for doing all processing and selecting an appropriate view.

We covered JSF during this course because JSF is recommended by the Java EE specifications.
In current practice, Spring is more widely used than JSF.
With the official endorsement of JSF, it is likely to grow in popularity in future.
In addition, there is the expectation that Java EE 8 will incorporate a new MVC

framework that is likely to be more close to Spring and so in the very long term, I anticipate that they will grow in popularity and Spring will slowly decline.

In any case, JSF is perhaps one of the most difficult MVC frameworks. If you understand how to use JSF and JAX-RS, you will not have any difficulty picking up Spring or any other Java-based MVC framework.



Action-based MVC frameworks typically follow the four steps listed in the slide.

Spring model

```
package au.edu.uts.aip.spring;

import javax.validation.constraints.*;

public class Task {

  private String title;
  private String description;

  @NotNull @Size(min = 1)
  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  @NotNull @Size(min = 1)
  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }

}
```

Spring is, at the current moment, perhaps the most popular MVC framework in Java.

The Model in Spring is exactly the same as the models we have been using in JSF.

Spring view

```
<h1>To-do List</h1>
<ul>
    <c:forEach var="item" items="${items}">
        <li>
            <c:out value="${item.title}"/>
            <i><c:out value="${item.description}"/></i>
        </li>
    </c:forEach>
</ul>

<form:form action="create" commandName="task">
    <p>
        <label>Title:
            <form:input path="title"/>
            <form:errors path="title" cssClass="error"/>
        </label>

    </p>
    <p>
        <label>Description:
            <form:input path="description"/>
            <form:errors path="description" cssClass="error"/>
        </label>
    </p>
    <p>
        <input type="submit" value="Add"/>
    </p>
</form:form>
```

The View in Spring is standard JSP with the JSTL (JavaServer Pages Standard Tag Library).
Spring also provides some additional tags. In the example above, we see a form:form, form:input and form:errors that work similar to h:form, h:inputText and h:messages in JSF.

Spring controller

```
@Controller
public class TodoListController {

  private List<Task> tasks = new ArrayList<>();

  @RequestMapping("/list")
  public String list(Model model) {
    model.addAttribute("items", tasks);
    model.addAttribute("task", new Task());
    return "list";
  }

  @RequestMapping("/create")
  public String create(Model model, @Valid Task task,
   BindingResult result) {
    if (result.hasErrors()) {
      model.addAttribute("items", tasks);
      return "list";
    } else {
      tasks.add(task);
      return "redirect:list";
    }
  }

}
```

The Spring controller is a Java class with annotations that handle configuration.

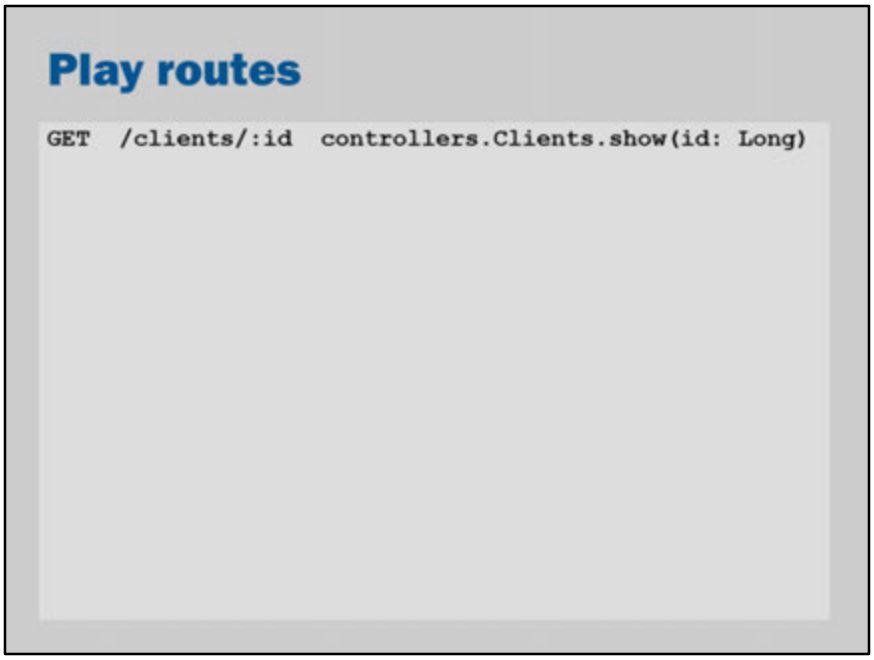
Note that the Spring controller is similar to JAX-RS in that annotations are used to define the path.
The model is passed in as a parameter, and the action/controller sets properties of the controller to use in the view.
The view is selected by the return values of the action.
Spring will map the view name into a JSP page and then view will be rendered with the model.

Play controller

```
package controllers;

import play.*;
import play.mvc.*;

public class Clients extends Controller {

    public static Result show(int id) {
        return ok(getClient(id));
    }

}
```

Play is an up-and-coming MVC framework that is perhaps even simpler.

In Play, the actions of methods are handled using static methods.



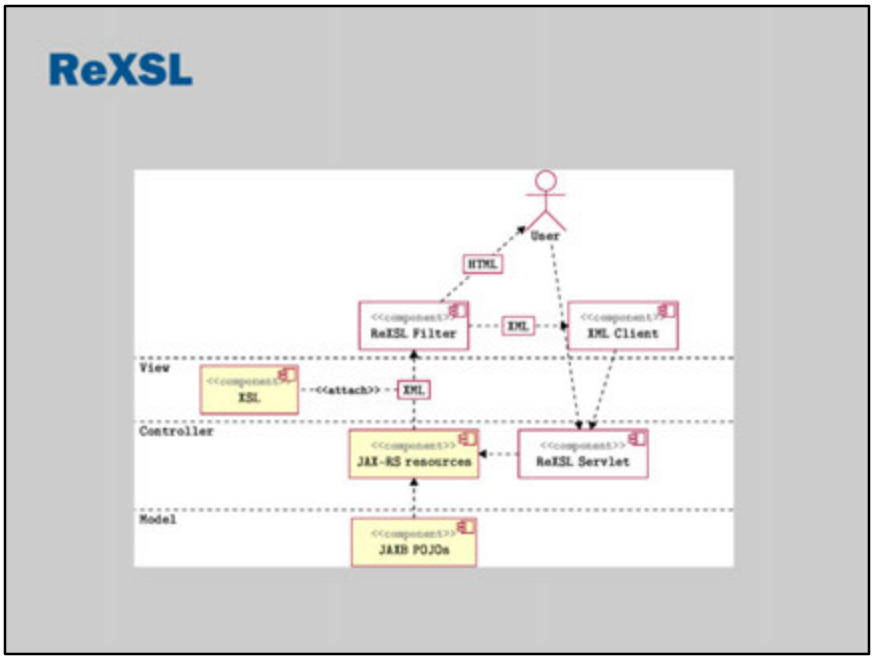
The mapping between URLs and the controllers/actions is not achieved with annotations. Instead a route mapping file is used to configure the mappings.

Play views

```
@(client: Client)

<h1>Welcome @client.name!</h1>

<ul>
@for(order <- client.getOrders()) {
  <li>@order.getDescription()</li>
}
</ul>
```

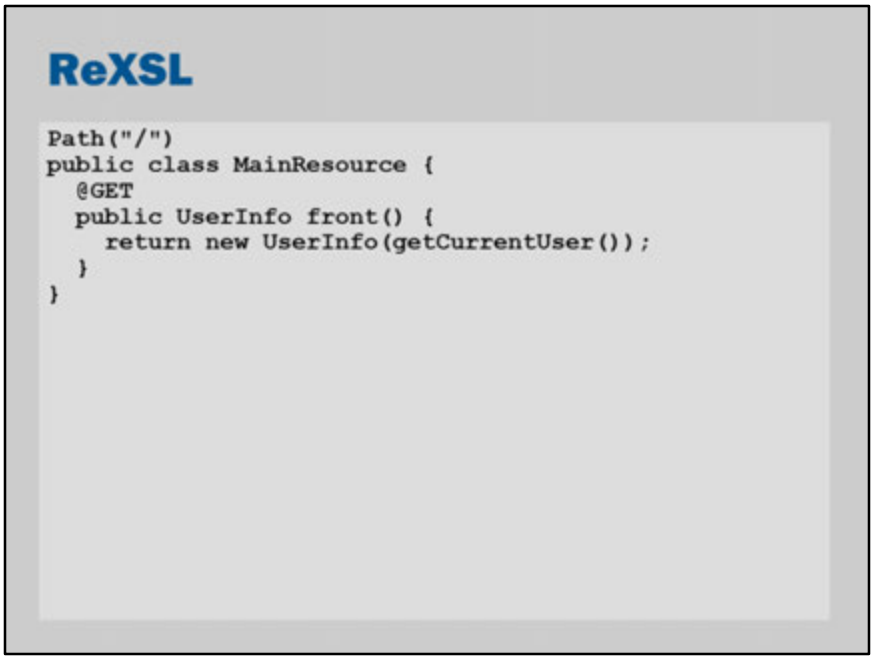
Play uses its own view language (based on the Scala programming language). However, the general principles are the same as in JSP. There are special escapes to insert values from the model.

Note that the first line of the view defines the data-type of the model that gets rendered. This allows for "strong" typing, even in the view.



ReXSL is not a major MVC framework (to my knowledge). However, it has an interesting approach and it is one alternative that comes up when you search for Java MVC frameworks.

The basic idea of ReXSL is to use JAX-RS for the model and controller. ReXSL uses JAX-RS to generate an XML representation of the resulting data. This is then transformed into a user-friendly HTML document by processing the XML document with an XSL stylesheet.



So… in ReXSL, you would use a standard JAX-RS implementation of a RESTful web service / resource.

ReXSL UserInfo.xsl

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml" version="1.0">
  <xsl:template match="/user">
    <html xml:lang="en">
      <body>
        <p>
          Welcome, <xsl:value-of name="fullname" />
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

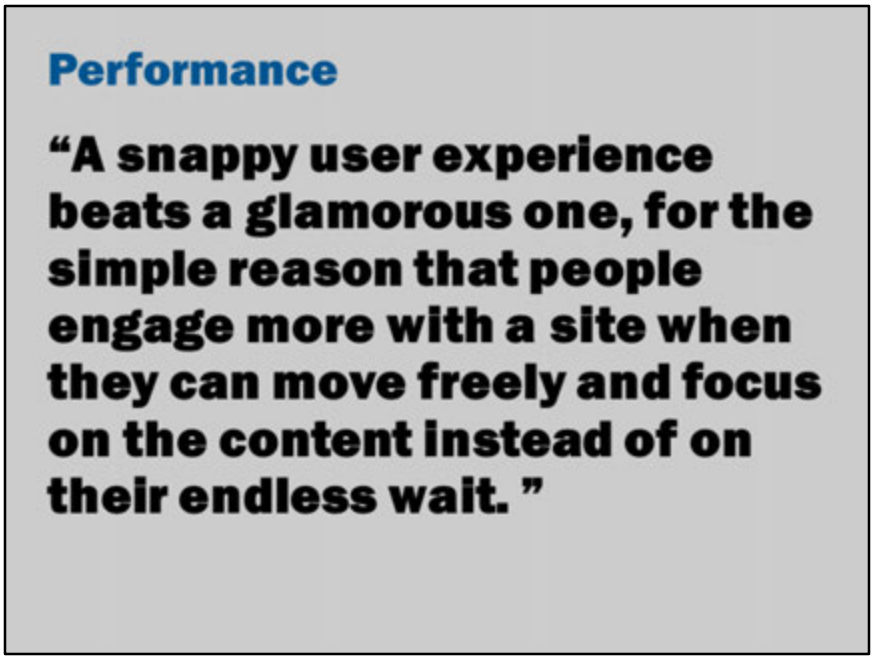
Then ReXSL would transform the XML output of the JAX-RS API into a HTML document.
The view language is XSL / XSLT (eXtensible Stylesheet Language / XSL Transformations).
The view in the slide renders a HTML document and extracts the fullname from the XML document returned by JAX-RS.

Java EE 8 MVC?

```
@Path("user")
public class UserController {

  @GET
  public Viewable get() {
    User user = getCurrentUser();
    return new Viewable("user_details", user);
  }

}
```

A new MVC framework is planned for Java EE 8.
The details of it are not yet available.
However, it is expected that it will be based on JAX-RS.
This means that we might expect a controller to look something like a JAX-RS resource.

Asynchrony



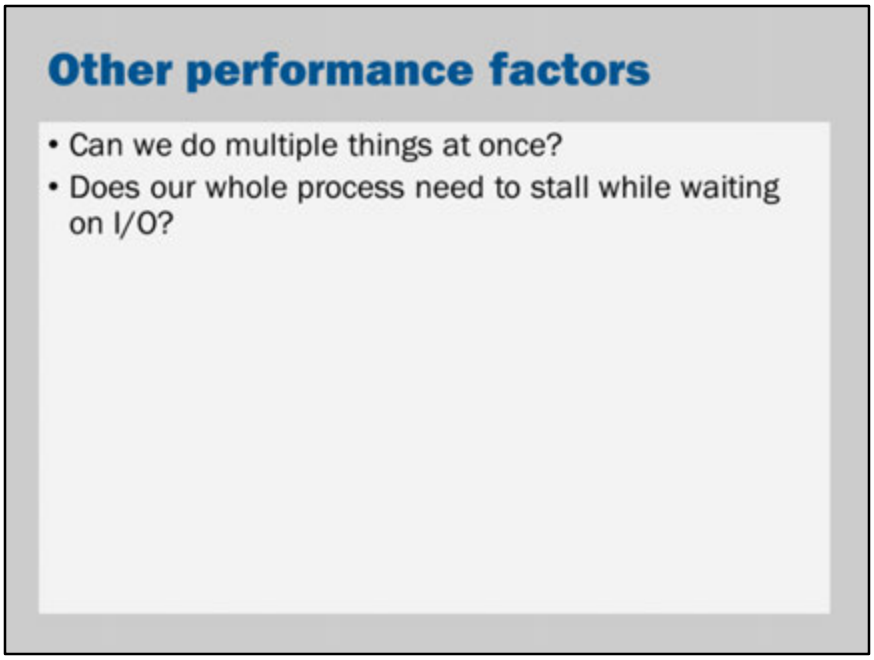
-- http://www.nngroup.com/articles/website-response-times/

The most important feature of an application is that it works.
If an application becomes unusable as soon as it has more than a few users, it isn't going to be very successful.

So, what are some strategies for dealing with many users at once?

The first, and easiest, step would be to buy a faster server. This is a perfectly legitimate approach.
You might also install reverse proxies and caches to bypass content generation for normally static (or slow changing) pages.
You could also install additional servers.

However, if scalability is an objective then it is important to also minimize overheads.



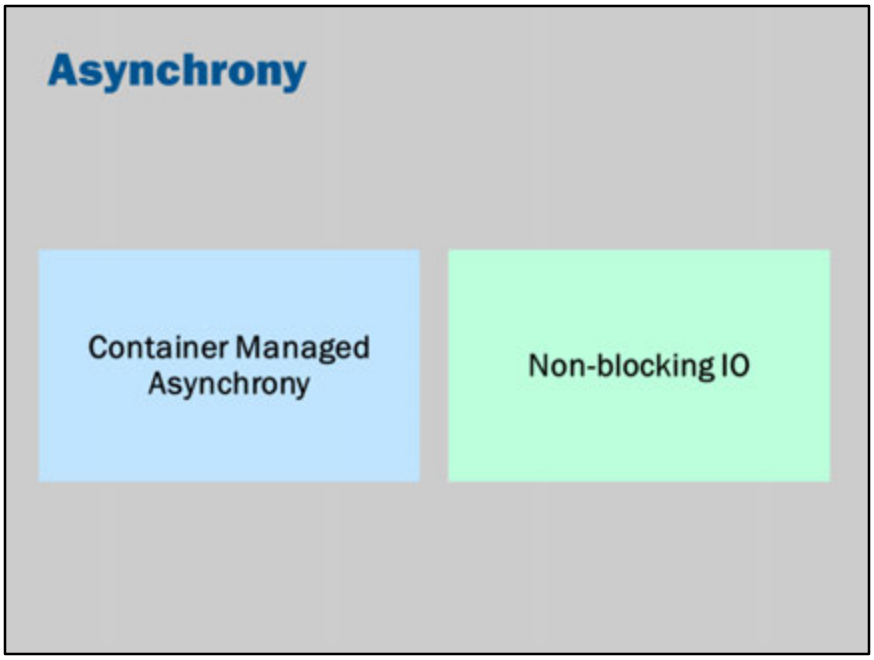
In minimizing overheads, our objective is to eliminate the things that can slow down our server.

1 CPU Cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50 - 150 us	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1-3 s	105-317 years
OS virtualization system reboot	4	423 years
SCSI command timeout	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 min	32 millennia

This table comes from the book Enterprise and the Cloud by Brendan Gregg.

It provides a dramatic demonstration of two things:
1. The phenomenal power of a modern CPU
2. How much time is wasted waiting on disk drives, waiting on network communication and even just waiting on main memory.

Asynchronous processing is a way to better manage waiting.



There are two main features for asynchronous processing in Java EE.

We have already seen container managed asynchrony before. This is where we allow the application server to automatically deal with asynchronous processing. The container will start and manage separate threads for us.

Non-blocking I/O is a technology available to Servlets (and other parts of the Java, such as the filesystem) that enables concurrently requests to be processed synchronously.

Asynchronous Servlets

```
@WebServlet(
    urlPatterns = "/AsyncServlet",
    asyncSupported = true)
public class AsyncServlet extends HttpServlet {

  @Override
  protected void doGet(...) throws ... {
    AsyncContext context = request.startAsync();
    processor.addToQueue(context);
  }

}
```

To use asynchrony in servlets, just add asyncSupported = true to the @WebServlet annotation.

Then, to switch the request to asynch mode, you call request.startAsync();

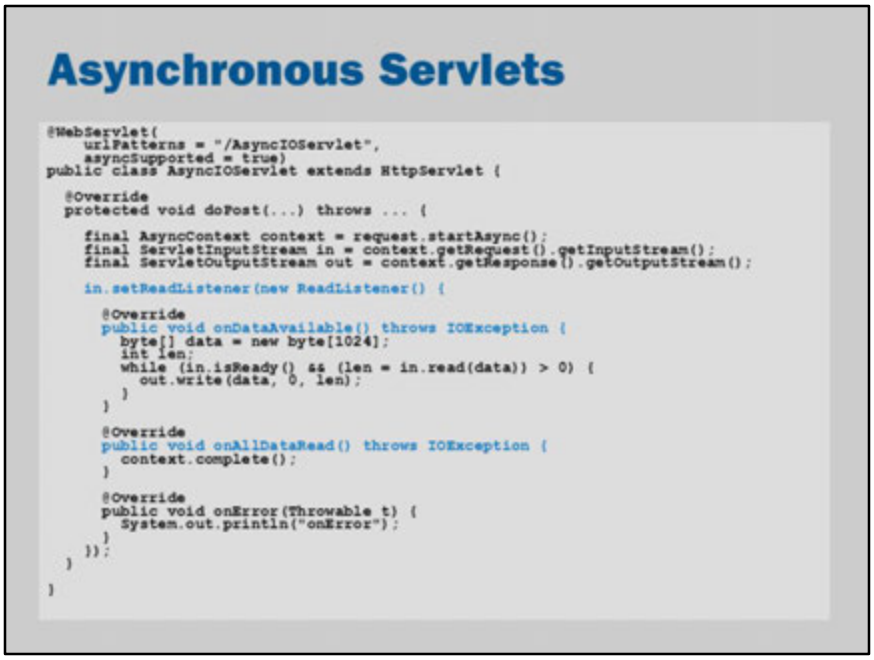
The AsyncContext that is returned can be used in other threads, queued up or otherwise used later.

Asynchronous Servlets

```
AsyncContext context = contexts.poll();
PrintWriter out = context.getResponse().getWriter();
out.println("Hello, World!");
context.complete();
```

Once you have an async context you can access it at a later time (perhaps from another thread).

When the response is complete, you communicate it to the server by calling context.complete();



The previous example makes sense if you don't have the response data immediately available.

Another possibility is that the request data is not immediately available (or may be very large).
You can configure the Servlet container so that it doesn't block waiting for data from the client.
Instead, you can use call backs. The container will call methods of the ReadListener when data is available, rather than requiring that your code block waiting for the data.

This can be a more efficient use of threads. It may also be useful if you have other processing that could be done while waiting for the complete request from the client.