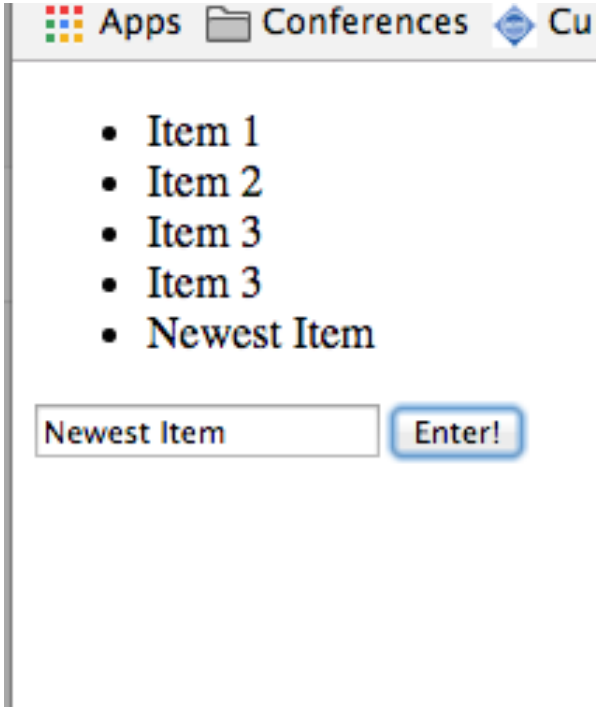
Objectives and Recap

- *Recap*
  - Last time we discussed web applications and REST
- Objectives
  - Reminders
  - Web applications
  - Regular Expression

The Simple Todo List

- In a todo list, the user enters some text that gets placed into an unordered list
- Such an application would be easy to implement in the client (browser) but the data would be lost every time the page was closed or refreshed
- Data should persist and be consistent with some external data store



Presentation and Data Layers

- Logically, the list items should reside on the server and the client-side should only act as an interface
- May think in terms of components and architecture
- *Todo Servlet*: Act as the controller and encapsulate the model (data) in the form of a simple list data structure
- *HTML*: Javascript to make an Ajax request when the page loads and automatically populate the html list with values from the request (client-side rendering)

Todo Servlet


```
// ...

private static List<String> list = new ArrayList<String>();

  public todoservlet() {
      super();
      list.add("Welcome, start adding items!");
  }

// ...
```

- `POST`: reads the data submitted by the user and add it to the internal representation of the todo list
- `GET`: return JSON string of the current state of the list as it resides on the server
- *Model*: A List of Strings

GET Behavior


```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws
                     ServletException, IOException {
    PrintWriter out = response.getWriter();
    response.setContentType("text/plain");
    Gson gson = new Gson();
    out.append(gson.toJson(list));
}
```

- GET request simply returns the data stored in our model
- Gson is a library for mapping back and forth between Java classes and JSON data

POST Behavior


```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
    String myText = request.getParameter("texttosend");
    list.add(myText);
}
```

- When the endpoint receives a POST request it adds the value in the parameter to the model

HTML


```
<body>
<h2>Todo List</h2>
<p>
<a href="todo">Todo Servlet</a>
</p>

<ul id="my-list"></ul>


Enter your text: <input type="text" id="texttosend"><br>
<button id="send-item" type="button" >Submit</button>

</body>
```

- Basic UI in HTML
- The list items will be added to the unordered list
- The input field and the button allow the user to send data

Front-end Script (1)


```
$(document).ready(function () {
  getListItems();

  $("#send-item").on('click', function () {
    sendListItem();
  });
});
```

- Use jQuery to register a click handler, callback function, on our button
- Only do this once we are sure the document as completely loaded

Front-end Script (2)


```
function getListItems() {
  $.ajax({
    url : 'todo',
    dataType : 'json',
    success : function(r) {
      console.log(r);
      items = r;
      console.log(items);
      updateList();
    }
    });
  }

function sendListItem() {
  var s = $("#texttosend").val();

  $.post("todo", { texttosend: s}, function() {
    getListItems();
  });
}
```

- These functions interact with the our servlet to retrieve and update the list items

Front-end Script (3)


```
items = [];
function updateList() {
  $("#my-list").html("");
  items.forEach(function (e) {
    $("#my-list").append("<li>" + e + "</li>");
  });
}
```

- This function displays the list items by adding them to the unordered list

Dealing with Requests

- When writing a web application, you may want to branch based on the request URL
  - Using the `@WebServlet` servlet annotation, one servlet can process requests for many different URLs
- RESTful apis are about accessing resources, we want to provide a sensible interface to those resources
- For example, for the to-do list we want our api to expose both the collection and the individual items
- One way to process a lot of string data is to use *regular expressions*

Regular Expressions (Java Classes)

- Regular Expressions (or *regex*) are a type of syntax for defining pattern matchers
  - Almost all languages have some support for regular expressions
  - Notation is generally consistent
  - A regular expression defines a (potentially infinite) set of strings
- In Java we use the `java.util.regex.Pattern` and `java.util.regex.Matcher` classes to work with regular expressions

```
Pattern pattern = Pattern.compile("[abc]+");
Matcher m = pattern.matcher("aabba");

m.matches(); // returns true for match
```

Regular Expressions (Basic)

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- A simple string is the simplest regular expression, searching for a fixed pattern
- For example, if we want to find the occurrences of the word ‘hello‘ in a sentence then we define a regex `"hello"`
- There are two important methods on the matcher object
  - `.find()`: search the input for a pattern
  - `.matches()`: attempt to match the entire input against the pattern

```
pattern = Pattern.compile("hello");
Matcher m = pattern.matcher("Can you say hello?");

// prints From 12 to 17
if (m.find()) {
    System.out.println("From " +
                       m.start() +
                          " to " +
                          m.end());
}
```

Regular Expressions (Syntax)

- There is more to regex than finding a fixed pattern in a string
- Regex is almost an entire language unto itself (fairly consistent across implementations)
- Some of the most important regex features are the following
  - Alternatives, one thing *or* something else, indicated with vertical bar | or square brackets
  - Character classes, express common ranges of characters (lowercase letters, digits)
  - Quantification, express how many times a pattern should occur
  - Grouping, extract sub-patterns in a match

```
Pattern.compile("b[ioa]t"); // matches bit, bat, and bot
Pattern.compile("b(i|o|a)t"); // same as above
```

Regular Expressions (Character Classes)

- Set of characters enclosed in square brackets, match single character of input
- Some predefined ranges in Java
  - `[0-9]`: digits
  - `[a-z]`: lowercase letters
  - `[A-Z]`: uppercase letters
  - `[a-zA-Z]`: upper and lower case letters
  - `.`: matches any single character
- Character classes can be negated with the caret symbol `^`
- Java also supports some POSIX character classes, see example below

```
// match any single character except a,b, or c
Pattern.compile("[^abc]");

// pattern for posix punctuation class
pattern = Pattern.compile("\\p{Punct}");
```

Regular Expressions (Quantification)

- We can also specify how many times something should occur using the *quantification* meta-characters: ?,*,+
  - `A*`: Kleene star, 'A' occurs zero or more times
  - `A+`: 'A' occurs one or more times
  - `A?`: 'A' occurs zero or one times
  - `A{n}`: 'A' occurs *n* times

```
// match any character any number of times
pattern = Pattern.compile(".*");

// match one or more digits followed by three letters
// e.g. 01234tuv
pattern = Pattern.compile("[0-9]+[a-zA-Z]{3}");

// two sequences of digits
// possibly separated by a lowercase letter
// 11111a2222, a22222, 11111a all match
pattern = Pattern.compile("[0-9]*[a-z]?[0-9]*");
```

Regular Expressions (Grouping)

- *Capture groups* allow us to capture part of a match
  - For example, we are processing names and want to quickly separate the first and last names
  - Use parentheses to indicate a group and retrieve group with the `group(int a)` method on the matcher

```
pattern = Pattern.compile("/([a-z]+)/([0-9]+)");

Matcher m = pattern.matcher("/abc/001");
m.matches();

// prints abc
System.out.println(m.group(1));

// prints 001
System.out.println(m.group(2));
```

Regular Expressions (Caveats)

- Regular expressions are a powerful notation for expressing patterns, but it is important not to overdo them
- Can make your code more difficult for others to interpret
- Easy to write a regex that you think is doing one thing but that is actually doing something else
- Consider the date validator below
- Strictly speaking, regular expressions are equivalent to the least expressive models of computation
  - The set of strings specified by a regex is a formal language
  - Most restricted grammar in the Chomsky hierarchy (Type-3 Grammars)

```
pattern = Pattern.compile("[0-9]{2}/[0-9]{2}/[0-9]{2}");
Matcher m = pattern.matcher("10/11/12");

System.out.println(m.matches());
```

lodash and Templating


```
// Create a template function
var listTemplate = _.template('<p><%= name %></p>');

// ...
// Use the template function
$('#maindiv').append(listTemplate({name : e}));
```

- A *template* is a special function that performs string interpolation
  - String interpolation: create a string by substituting for placeholder values
- In the example, we use `lodash`[1] (a utility library for javascript) to create a template function
- The template function takes a javascript object as an argument

[1] `https://lodash.com/docs/#template`

JAX-RS

- The servlet API can be used to implement a RESTful service (as shown in the previous exercise)
  - Sometimes you can be more effective by leveraging an existing solution
- JAX-RS is a framework designed to make the creation of RESTful web services easier
- Package is named `javax.ws.rs`
- Based around annotations of plain Java objects

RESTful Services

- Architectural style where we think of data and functionality as resources that accessed by URI
- Transfer representations of resources through requests and responses (Client-Server)
- Uniform Interface for CRUD operations
  - GET: retrieve current state
  - POST: change the state of a resource
  - PUT: create a new resource
  - DELETE: delete a resource

Features of JAX-RS

- JAX-RS[2] methods bind and map HTTP requests to Java Methods
- Parameter Injection annotations
  - Makes it simple to get information the request
- Annotations can be applied to either Java classes or interfaces
  - Separate the implementation of the service from the business logic
- Services can either be *singletons* or *per-request*

[2] Described in Ch. 29 of Eric Jendrock et al. *The Java EE Tutorial, Release 7*. Oracle, 2014

Basic Service (JAX-RS)

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
@Path("/items")
public class TestJaxRsService {

    @GET
    public Response getItems() {
        String r = "JAX-RS: Hello, world";
        return Response.ok(r).build();
    }
}
```

- Each request corresponds to an annotated method
- Handles all requests with path <app>/items
- Can also place path annotations on methods, using regex

Retrieving the collection


```
@GET
@Path("{id: [a-zA-Z]+}")
public Response getById(@PathParam("id") String id) {
    String r = "Requested for item with id \"" + id + "\"";
    return Response.ok(r).build();
}
```

- Looks for a particular item, `item/aaa`, `item/bbb`
- Match with a regular expression and pass the value to our method

Creating a New Resource


```
@POST
@Consumes("application/json")
public void postCity(String req) {
    d.addRow(req);
}
```

- Respond to a post request
- The consumes annotation indicates that our API is expecting to receive json data

Removing a Resource


```
@DELETE
@Path("{myId : [0-9]+}")
public void removeCity(@PathParam("myId") String myId) {
  d.deleteRow(Integer.parseUnsignedInt(myId));
}
```

- Make a request to a particular resource for deletion
- Same regular expression to match the request URL but we are implementing a different http verb than before

Using JAX-RS with Tomcat and Maven


```
<dependency>
    <groupId>org.glassfish.jersey.bundles</groupId>
    <artifactId>jaxrs-ri</artifactId>
    <version>2.26</version>
</dependency>
```

- This is the only dependency you should need to make JAX-RS work in a servlet container
- *Jersey* is the reference implementation of JAX-RS

Application class

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
@ApplicationPath("/services")
public class ExampleApplication extends Application {
    private Set<Object> singletons = new HashSet<Object>();
    private Set<Class<?>> empty = new HashSet<Class<?>>();

    public ExampleApplication() {
        singletons.add(new TestJaxRsService());
        singletons.add(new NewService());
    }

    @Override
    public Set<Class<?>> getClasses() {
        return empty;
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

- The `javax.ws.rs.core.ApplicationPath` is the simplest way to configure your restful service (saves having to write extra configuration in `web.xml`)

Response Codes

- HTTP response codes cover different success/error conditions when making requests
- Exceptions and errors may require specific response headers
- *Success Codes*: in the range 200-399
  - 200, OK: Successful Request
  - 204, No Content: Successful request but no message body
- *Error Codes*: in the range 400-599
  - 405, Method not Allowed: Valid URI but unsupported HTTP method
  - 406, Not Acceptable: No resource for requested content type

Returning an Error Response (JAX-RS)


```
@Path("/items")
public class TestJaxRsService {

    @DELETE
    public Response getItems() {
      return Response.status(Status.GONE).build();
    }
}
```

- In this case we are returning a 410, trying to delete a resource that is already gone
- The `Status` enum contains HTTP response codes in a convenient format

Returning a Response (JAX-RS)


```
@Path("/items")
public class TestJaxRsService {

    @GET
    public Response getItems() {
        String r = "JAX-RS: Hello, world";

        ResponseBuilder b = Response.ok(r);
        b.header("header-name", "value");

        return b.build();
    }
}
```

- Returning a `Response` object in JAX-RS
- Set the body of the response in the `ok()` method
- Can add custom headers using the fluent interface

Query Parameters

- Query Parameters are the values that appear after a '?' in the uri
- Query Parameters can be accessed in a JAX-RS method by annotating local variables with the `@QueryParam(<name>)` annotation
- For example we can retrieve a range of items from our resource by setting `start` and `end` Query Parameters
- This type of functionality is best communicated to the client via links (HATEOAS)