Welcome to ACS TA Session 1

Yijian Liu, Li Quan, Rodrigo Laigner, and Ziming Luo

Department of Computer Science, University of Copenhagen

Academic year 2020 - 2021, Block 2

Overview of TA sessions

Schedule

* Weeks 47-53 of 2020 and 1-2 of 2021
* From 13:15 to 16:00
* Lundbeck Auditoriet, Ole Maaløes Vej 5 (Biocenter) and online (links on Absalon)

Agenda

* Exercises on the ACS topics covered in the week
* Questions and answers for clarifications on the topics
* Work on the assignment and get **unstuck** if needed
* Do you have something else on your mind?

Agenda for today

Agenda

* Clarification on the topics from the course - now it’s a great time for you to **ask questions**
* Java best practices - useful for successful assignments
* Testing using the **JUnit** framework
* Remote procedure calls using the **Jetty** framework
* **Optional**: work on a the **Bank Account** sample project
* **Optional**: deploy the **Book Store** project in the cloud

Best practices - data encapsulation

- Everything in Java is (ultimately) a reference
    - No pointer handling or (direct) memory allocation
- Encapsulation needs proper thought
    - What data should be encapsulated together
    - What (object) references are exposed through your abstractions
- Immutability
    - Preserve encapsulated mutable data are not shared
    - Using a copy constructor vs the `Cloneable` interface
- Thread safety raises further questions to encapsulation
    - Concurrent threads may harm data integrity? If so, how to protect it?

Best practices - APIs

- All-or-nothing semantics between clients and servers
- A possible strategy is to organize functions as **two-phase**

- Breaking a large request with all-or-nothing semantics into multiple underlying messages can hurt semantics
  - Can you discard changes made after failures?
- Performance depends on messaging overhead. Too many messages is not a good idea for ease of programming

- Use the **appropriate** algorithms and data structures. Think of algorithms in action and reflect about their performance!

The importance of testing

Quotes from a project with TDD at Lund University

* "*Look, I know it works and I can show you, see?*"
* "*Seriously, how am I supposed to test the main method?!*"
* *Testing code may perhaps not be the most fun activity but it's definitively better to do it first instead of trying to make the tests fit the code after the fact. It gives a really positive feedback to see the tests pass and turn green!*

Best practices - test-driven development

Recommended team organization

* Consider "pairwise" programming
* One person writes the tests, another person writes the codes
* Alternate roles for different systems under test
* For Java, use the JUnit test framework (covered today and in the assignments)

Recommended order of steps

* Both design and implement the API for the component
* Alice writes failing test cases for this API
* Bob implements the component step by step
* In effect of the implementation, all tests gradually pass

Best practices - JUnit testing

* Isolate the test cases from each other
* Isolate the component under test from dependencies (for unit tests)
* Do not isolate the component under test from dependencies (for integration tests)
* Test sunshine, dark and unexpected scenarios
* Test corner cases and values e.g.,
  * For numerics, test the maximum, minimum, zero, other thresholds as appropriate
  * For strings, test the empty string and null
  * For IDs, test both existing and missing IDs
  * For time or space complexities of functions, clamp f(n) for a sufficiently large, but tractable n
* Consider handling expected exceptions, e.g., using `@Test(expected=SomeException.class)`
* Consider using a test coverage tool, e.g., www.eclemma.org

JUnit examples

The Account interface

```
package com.mybank;

public interface Account {

    public int getBalance() throws
AccountException;

    public void deposit(int n) throws
AccountException;

    public void withdraw(int n) throws
AccountException;
}
```

The Bank Account class

```
package com.mybank;

public class BankAccount implements Account {
    private int balance;
    private int cpr;

    public BankAccount(int newCpr) {
        cpr = newCpr;
        balance = 0;
    }

    public synchronized int getBalance() {
        return balance;
    }

    public synchronized void deposit(int n) throws
AccountException {
        if (n < 0) {
            throw new AccountException("n less than 0");
        }

        balance = balance + n;
    }

    public synchronized void withdraw(int n) throws
AccountException {
        if (n < 0) {
            throw new AccountException("n less than 0");
        }

        if (n > balance) {
            throw new AccountException("n exceeds balance");
        }

        balance = balance - n;
    }
}
```

JUnit examples

The Bank Account class

```
package com.mybank;

public class BankAccount implements Account {

   /** The balance of the account. */
   private int balance;

   /** The CPR. */
   private int cpr;

   public BankAccount(int newCpr) { … }

   public synchronized int getBalance() { … }

   public synchronized void deposit(int n) throws AccountException { … }

   public synchronized void withdraw(int n) throws AccountException { … }
}
```

Tests

- `getBalance() == 0` after construction?
- `getBalance() == correctResult` after one call / series of calls of deposit and/or withdraw?
- `withdraw` raises `AccountException` if `n > getBalance()`?
- `deposit(n < 0)` or overflowing raises `AccountException`?
- `withdraw(n < 0)` or overflowing raises `AccountException`?

Best practices - JUnit testing

* Test the exact equality of data structure contents

* Check for content, not only sizes or single attributes

* Test for invariants and not exceptions

JUnit examples

The Bank class

```
package com.mybank;

public class Bank {

   /** The accounts in a data structure. */
   private DataStructure<int, Account> accountsByCpr;

   public Bank(int totalNumberOfAccounts) { … }

   public synchronized Account findAccount(int cpr) { … }

   public synchronized void openAccount(int cpr) throws AccountException { … }

   public synchronized void closeAccount(int cpr) throws AccountException { … }
}
```

Test

`openAccount`

- test the right number of accounts before and after, as well as the right number and that the CPR is present
- what happens if the same CPR is added twice? *etc*.

`findAccount`

- test for both added account and for non-present account

`closeAccount`

- check the right number and that the CPR is removed
- what happens if we try with a non-present CPR?
- what happens if the removed CPR is added afterwards? *etc*.

JUnit tests file structure

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;

public class TestClass {

  @BeforeClass
  public void setUp() { … }          →  Before any test

  @AfterClass
  public void tearDown() { … }       →  After all tests

  @Before
  public void prepareTest() { … }    →  Before each test

  @After
  public void cleanUpTest() { … }    →  After each test

  @Test
  public void void testX() { … }

  @Test
  public void void testY() { … }
}
```

JUnit useful methods

JUnit

```
assertTrue(String message, boolean condition)
assertEquals(String message, TYPE expected, TYPE actual)
assertNotNull(String message, Object obj)
assertNull(String message, Object obj)
assertSame(String message, Object expected, Object actual)
fail(String message);
```

Java

```
equals, contains, containsAll
```

JUnit setup

Eclipse

* **Create tests**: right click in the package, select New: JUnit Test Case
* **Run tests**: right click test class, select Run As: JUnit Test
* **Run tests in non-local mode**: run the server and then run the tests

NetBeans

* **Create tests**: right click on the class to be tested, select Tools: Create Tests
* **Run tests**: go to the Main menu and select Run: Test (project name)

Local vs non-local flag

* Used extensively in the assignments to test directly or through RPC, respectively
* **Local run**: run the tests normally with local flag set to `true`
* **Run tests**: start the server and then run the tests with local flag set to `false`

Jetty servlet engine and web server

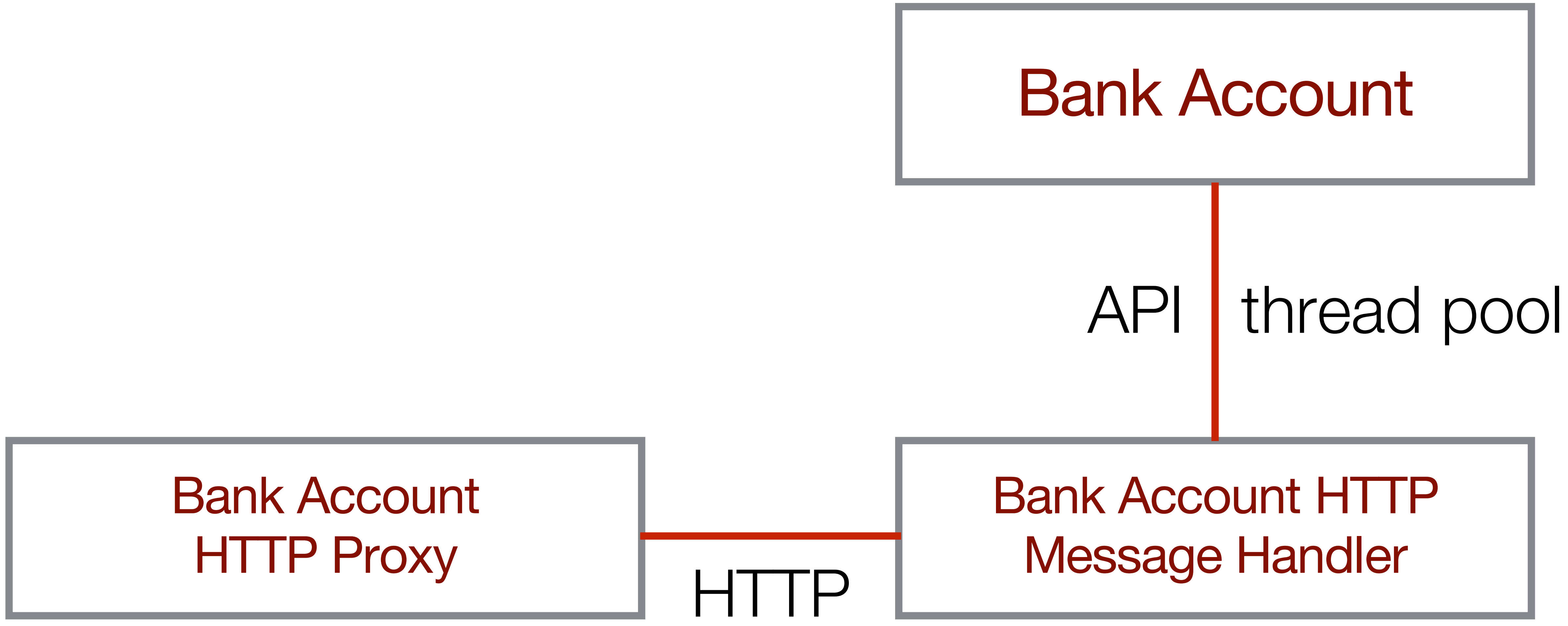
- Web server and servlet container with support for HTTP, web sockets etc.
- Used in a wide variety of projects and products
- Open source and available for commercial use and distribution
- www.eclipse.org/jetty

Jetty proxy and server code

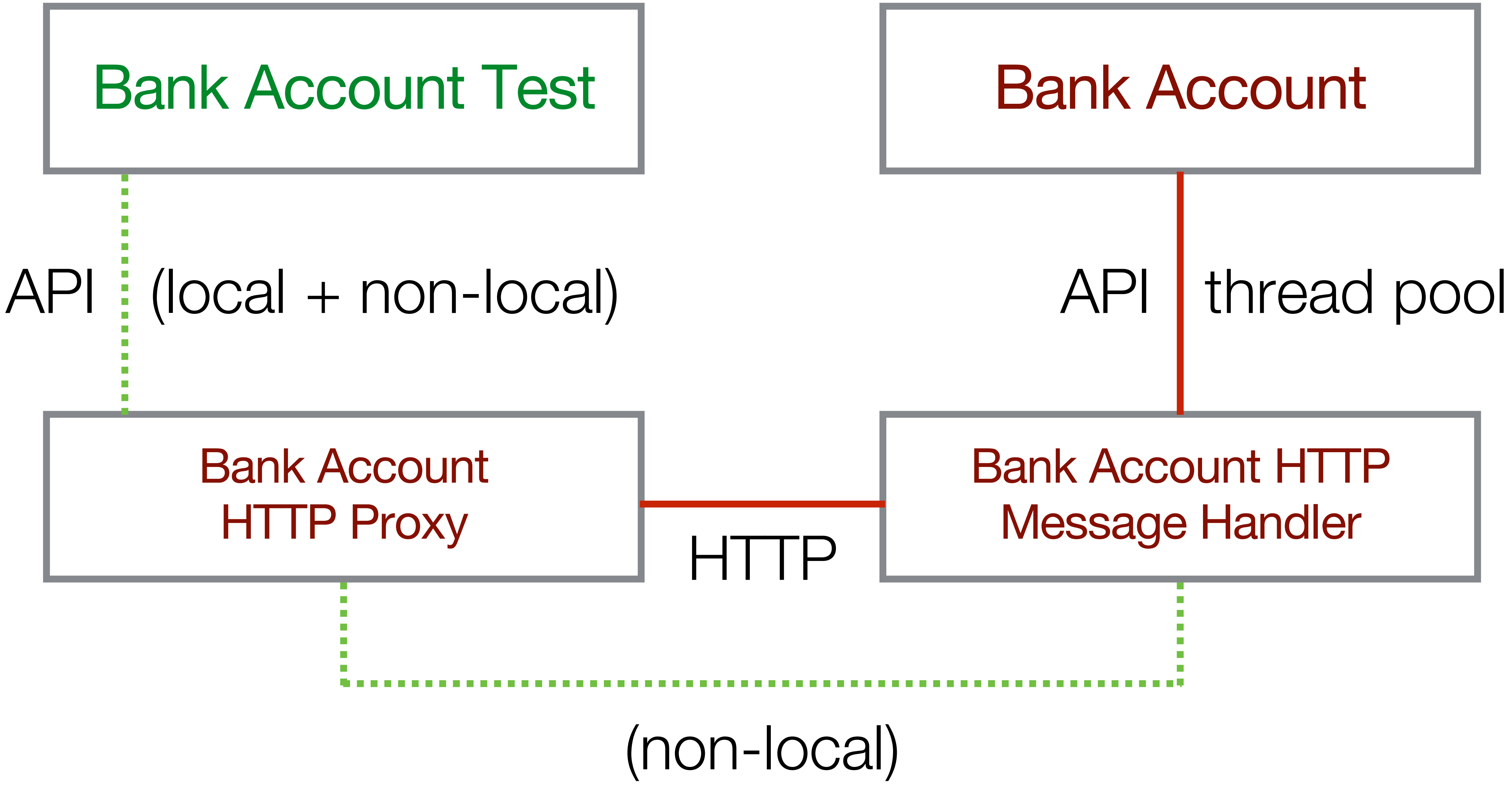
Proxy code

```
public class BankAccountHTTPProxy implements Account {
   protected HttpClient client;
   protected String serverAddress;

   public BankAccountHTTPProxy(String serverAddress) throws Exception {
      serverAddress = serverAddress;
      client = new HttpClient();

      // Max concurrent connections to every address.
      client.setMaxConnectionsPerDestination(CLIENT_MAX_CONNECTION_ADDRESS);

      // Max number of threads.
      client.setExecutor(new QueuedThreadPool(CLIENT_MAX_THREADSPOOL_THREADS));

      // Seconds timeout; if no server reply, the request expires.
      client.setConnectTimeout(CLIENT_MAX_TIMEOUT_MILLISECS);

      client.start();
   }
   …
}
```

Jetty with RPC



Jetty with RPC ...and tests



Jetty proxy and server code

Server code

```
public class BankAccountHTTPServerUtility {
   public static boolean createServer(int port, AbstractHandler handler) {
      Server server = new Server(port);

      if (handler != null) {
         server.setHandler(handler);
      }

      try {
         server.start();
         server.join();
      } catch (Exception ex) {
         ex.printStackTrace();
      }

      return true;
   }
   …
}
```

Jetty proxy and server code

Server code

```
public class BankAccountHTTPMessageHandler extends AbstractHandler {
  private final Account account; // Initialized in the constructor, omitted from this listing for brevity.

  public void handle(String target, Request baseRequest, HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {
    MessageTag messageTag = null;
    String requestURI;
    response.setContentType("text/html;charset=utf-8");
    response.setStatus(HttpServletResponse.SC_OK);
    requestURI = request.getRequestURI();

    if (!BankAccountUtility.isEmpty(requestURI)) {
      messageTag = BankAccountUtility.convertURItoMessageTag(requestURI);
    }

    if (messageTag == null) {
      System.err.println("Unknown message tag");
    } else {
      switch (messageTag) {
      case GETBALANCE:
        getBalance(response); break;
      case DEPOSIT:
        // Write your implementation here!
        break;
      case WITHDRAW:
        withdraw(request, response); break;
      default:
        System.err.println("Unhandled message tag”); break;
      }
    }

    baseRequest.setHandled(true);
  }
}
```

JUnit exercise

1. Design the tests for the `deposit(int n)` method
2. Part of the team writes failing tests in `BankAccountTest`
3. Part of the team implements `deposit` in `BankAccountHTTPProxy` and `BankAccountHTTPMessageHandler`
4. Make the tests pass

- Optional: complete the tests for the other methods

Cloud platform deployment

Cloud platforms

- Google Cloud
- Microsoft Azure
- Amazon Web Services
- …and many others

Amazon Web Services

- Deployment on the AWS EC2
- absalon.ku.dk/courses/2576/pages/aws-wiki

Thank you

Yijian Liu, Li Quan, Rodrigo Laigner, and Ziming Luo