Chapter 4: Build and test the network

Learning Bluemix & Blockchain

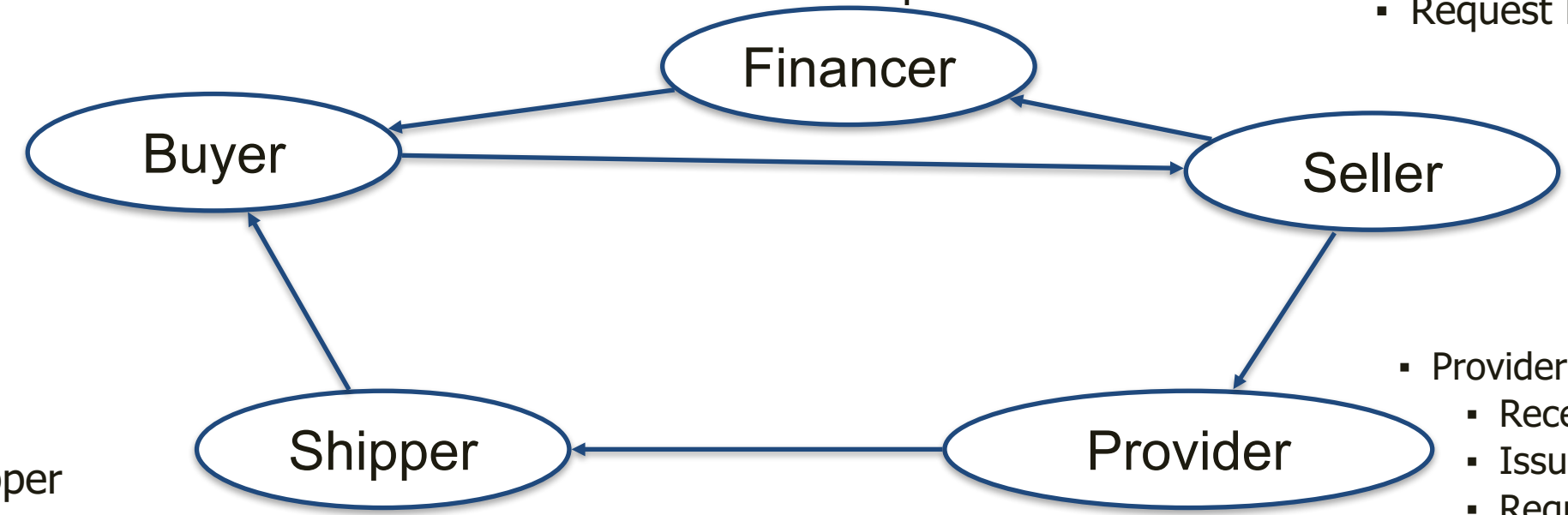
Bob Dill, IBM Distinguished Engineer, CTO Global Technical Sales
David Smits, Senior Certified Architect, IBM Blockchain

The Plan: 30 minute Chapters with an hour or two of practice

Chapter 1:	What is Blockchain? Concept and Architecture overview
Chapter 2:	What's the story we're going to build
Chapter 2.1:	Architecture for the Story
Chapter 3:	Set up local HyperLedger Fabric V1 development environment
Chapter 4:	**Build and test the network**
Chapter 5:	Administration User Experience
Chapter 6:	Buyer Support and User Experience
Chapter 7:	Seller Support and User Experience
Chapter 8:	Shipper Support and User Experience
Chapter 9:	Provider Support and User Experience
Chapter 10:	Finance Company Support and User Experience
Chapter 11:	Combining for Demonstration
Chapter 12:	Events and Automating for Demonstration

Who are the participants and what can they do?
This defines the members of the network, the transactions, and who should approve transactions

- Buyer
  - Create an Order
  - Submit an Order
  - Receive a Shipment
  - Dispute an Order
  - Approve Payment on an Order
- Finance Company
  - Receive Request for Payment
  - Approve and Pay
  - Receive notification of dispute
  - Resolve Dispute
- Seller
  - Accept an Order
  - Submit an Order to a Provider
  - Receive Notification of Delivery
  - Request Payment



- Shipper
  - Receive Request for Delivery
  - Post notification of delivery
- Provider
  - Receive an item request
  - Issue, resolve backorder
  - Request Delivery
  - Receive Notification of Delivery

Defining Members

- In our simple network, members have:
  - A company name
  - An identifier, which we'll implement as an email address
- We define an abstract type of member with a single field called "companyName"

```
18  namespace composer.base
19
20  abstract participant Member {
21    o String companyName
22  }
```

- We then extend this abstract type for each of the types of members.

```
participant Buyer identified by buyerID extends Member{
    o String buyerID
}
```

- The objective is to introduce abstract types and our ability to separate different types of definitions into separate files, for maintenance purposes while still easily combining everything when we're done.

Defining Assets

- The single asset which we'll deal with in this tutorial is an "Order", which is defined as shown here:
- Many of the fields are for storing dates, so we can tell what happened when. Dates and reasons are updated via transactions, which will be limited by participant.
- Note the square brackets, these denote arrays
- Note the arrows (- ->), which denote references to other, previously defined, Network classes.

```
asset Order identified by orderNumber {
    o String orderNumber
    o String[] items
    o String status
    o Integer amount
    o String created
    o String bought
    o String ordered
    o String dateBackordered
    o String requestShipment
    o String delivered
    o String disputeOpened
    o String disputeResolved
    o String paymentRequested
    o String orderRefunded
    o String paid
    o String[] vendors
    o String dispute
    o String resolve
    o String backorder
    o String refund
    --> Buyer buyer
    --> Seller seller
```


Defining Transactions

- Transactions use the same model language as assets and members.
- Here we will name a transaction and identify what must accompany a request for this transaction to be processed:
- This is a transaction Class
- It's named CreateOrder
- It has one field (Integer amount)
- and refers to 3 other instances
  - An Order
  - A Buyer
  - A Seller

```
transaction CreateOrder {
  o Integer amount
  --> Order order
  --> Buyer buyer
  --> Seller seller
}
```


Taking this information, define the following:

- Members:
  - Buyer, Seller, Provider, Shipper, FinanceCo
- Assets:
  - Order
- Transactions:
  - CreateOrder, Buy, OrderFromSupplier, RequestShipping, Deliver, BackOrder, Dispute, Resolve, Request Payment, Pay, Refund

Let’s check out the network

- Step 1, update the model files
- Step 2, create and archive and deploy it
- Step 3 load up composer and test it.

- Step 1, the Answers are in the Documents/answers folder
- Step 2, execute the following command from within Chapter04 folder
  - **`buildAndDeploy`**
- Step 3, go to: https://composer-playground.mybluemix.net/
  - import your model file from **`Chapter04/network/dist/zerotoblockchain-network.bna`**
  - test the model
  - You’ll notice on inspection that not much happens with the Order object, that's next

Writing the code to implement the transactions

- Each transaction needs implementing logic. For example, the Create Order transaction exists to allow a buyer to build an order and save it prior to sending it to a seller. The code is shown to the right.

```
/**
 * create an order to purchase
 * @param {org.acme.Z2BTestNetwork.CreateOrder} purchase - the order to be processed
 * @transaction
 */
function CreateOrder(purchase) {
    purchase.order.buyer = purchase.buyer;
    purchase.order.amount = purchase.amount;
    purchase.order.created = new Date().toISOString();
    purchase.order.status = "Order Created";
    return getAssetRegistry('org.acme.Z2BTestNetwork.Order')
        .then(function (assetRegistry) {
            return assetRegistry.update(purchase.order);
        });
}
```

- You can see that the class definition (below right) includes a link to the Order, the Amount of the order, and the Buyer. In this transaction code, the Seller information is not used - because the order has not yet been placed with the seller.

```
transaction CreateOrder {
  o Integer amount
  --> Order order
  --> Buyer buyer
  --> Seller seller
}
```


Writing code to test the transactions

- We are using the mocha service to test this application, the code looks like this:
- We will explore this code interactively in a moment
- When we're done, we can tell npm to test what we've created, which should deliver results like the following:

```
Finance Network
  #createOrder
    ✓ should be able to create an order (82ms)
  #issueBuyRequest
    ✓ should be able to issue a buy request (40ms)
  #issueOrderFromSupplier
    ✓ should be able to issue a supplier order (50ms)
  #issueRequestShipment
    ✓ should be able to issue a request to ship product (47ms)
  #issueDelivery
    ✓ should be able to record a product delivery (39ms)
  #issueRequestPayment
    ✓ should be able to issue a request to request payment for a product (58ms)
  #issuePayment
    ✓ should be able to record a product payment (48ms)
  #issueDispute
    ✓ should be able to record a product dispute (63ms)
  #issueResolution
    ✓ should be able to record a dispute resolution (48ms)
  #issueBackorder
    ✓ should be able to record a product backorder (53ms)


10 passing (1s)
```

```
describe('#createOrder', () => {

    it('should be able to create an order', () => {
        const factory = businessNetworkConnection.getBusinessNetwork().getFactory();

        // create the buyer
        const buyer = factory.newResource(NS, 'Buyer', buyerID);
        buyer.companyName = 'billybob computing';

        // create the seller
        const seller = factory.newResource(NS, 'Seller', sellerID);
        seller.companyName = 'Simon PC Hardware, Inc';

        // create the order
        let order = factory.newResource(NS, 'Order', orderNo);
        order = createOrderTemplate(order);
        order = addItems(order);
        order.orderNumber = orderNo;

        // create the buy transaction
        const createNew = factory.newTransaction(NS, 'CreateOrder');

        order.buyer = factory.newRelationship(NS, 'Buyer', buyer.$identifier);
        order.seller = factory.newRelationship(NS, 'Seller', seller.$identifier);
        createNew.order = factory.newRelationship(NS, 'Order', order.$identifier);
        createNew.buyer = factory.newRelationship(NS, 'Buyer', buyer.$identifier);
        createNew.seller = factory.newRelationship(NS, 'Seller', seller.$identifier);
        createNew.amount = order.amount;
        // the buyer should of the commodity should be buyer
        //order.buyer.$identifier.should.equal(buyer.$identifier);
        order.status.should.equal('Order Created');
        order.amount.should.equal(orderAmount);
        createNew.amount.should.equal(orderAmount);
        createNew.order.$identifier.should.equal(orderNo);
```

Invoke the composer-rest-server

- from Chapter04
  - execute the following command:
    - **`buildAndDeploy`**
  - this will load your completed network into docker
  - Execute the following command
    - **`./start_rest_server.sh`**
  - which will apply the following responses:

```
[? Enter your Fabric Connection Profile Name: hlfv1
[? Enter your Business Network Identifier : zerotoblockchain-network
[? Enter your Fabric username : admin
[? Enter your secret: adminpw
 ? Specify if you want namespaces in the generated REST API: always use namespaces
[? Specify if you want the generated REST API to be secured: No
[? Specify if you want to enable event publication over WebSockets: Yes
[? Specify if you want to enable TLS security for the REST API: No
```

- Go to **`localhost:3000/explorer`** and inspect and test your new RESTful APIs

The Plan: 30 minute Chapters with an hour or two of practice

Chapter 1:	What is Blockchain? Concept and Architecture overview
Chapter 2:	What's the story we're going to build
Chapter 2.1:	Architecture for the Story
Chapter 3:	Set up local HyperLedger Fabric V1 development environment
Chapter 4:	Build and test the network
Chapter 5:	Administration User Experience
Chapter 6:	Buyer Support and User Experience
Chapter 7:	Seller Support and User Experience
Chapter 8:	Shipper Support and User Experience
Chapter 9:	Provider Support and User Experience
Chapter 10:	Finance Company Support and User Experience
Chapter 11:	Combining for Demonstration
Chapter 12:	Events and Automating for Demonstration

Chapter 5: Administration User Experience

Learning Bluemix & Blockchain

Bob Dill, IBM Distinguished Engineer, CTO Global Technical Sales
David Smits, Senior Certified Architect, IBM Blockchain