z/TPF Detailed Summary
z/TPF support for MongoDB

Claire Durant
z/TPF Development

What is MongoDB?

- Document-based NoSQL database system with client support for many common platforms & languages.

- A MongoDB server has multiple **databases**. Each database contains multiple **collections**. A collection should contain **documents** that share the same purpose.

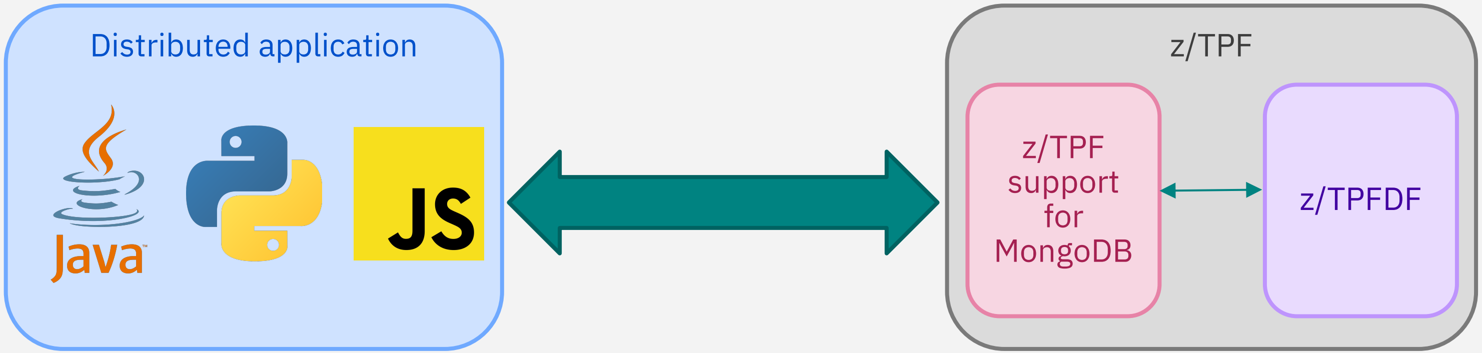
  - For example, you may have a collection called "CreditCards" that contains every credit card in your system.

- Data is stored as hierarchical BSON (Binary JSON) documents. For example:

```
{
  “name” : “Claire Durant”,
  “job” : {
    “title” : “software engineer”,
    “employer”: “IBM”
  }
}
```

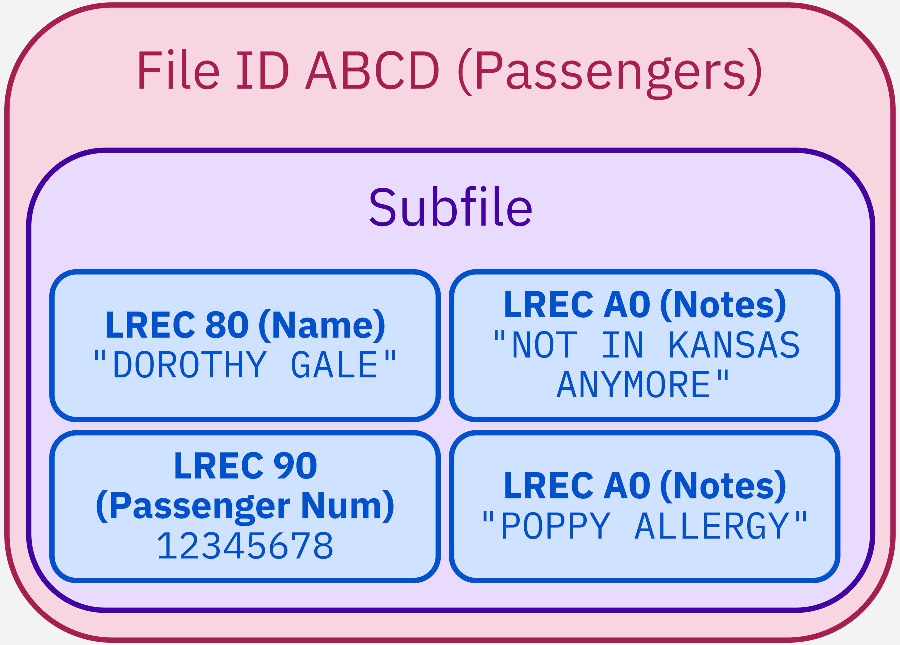
z/TPF support for MongoDB

- **NOT** a port of MongoDB on z/TPF!
- Interface that allows existing z/TPFDF databases to "act like" MongoDB collections.
  - Currently supports z/TPFDF databases only.



Why MongoDB?

– z/TPFDF data maps well to MongoDB’s hierarchical document model



```
{
  "PassengerNameRecord" : [
    { "PassengerName" : "DOROTHY GALE" }
  ],
  "PassengerNumberRecord" : [
    { "PassengerNumber" : 12345678 }
  ],
  “NotesRecord" : [
    { “Note" : “NOT IN KANSAS ANYMORE“ }
    { "Note" : "POPPY ALLERGY" }
  ],
  "_id" : ObjectId("0000000000000000018043344")
}
```

How does it work?

- Use the `ZUDFM DESCRIPTOR` command to create the initial database descriptions
  - DFDL schema file describing the format of data in each LREC.
  - z/TPFDF collection descriptor describing some attributes of the file's database definition (DBDEF).
- Transfer files offline and customize
  - Provide meaningful names to z/TPFDF files, index paths, LRECs, and fields in LRECs
  - Specify appropriate data types for each field
  - Format LRECs as multiple distinct fields (if not already defined in the DSECT)
  - Use any supported DFDL features to accurately describe your data format
- Load DFDL schema files and collection descriptors through common deployment

Getting Started

Creating deployment descriptors

- `ZUDFM DESCRIPTOR FILE-`*`B426`*
  - B426 is the z/TPFDF file ID we want to access with MongoDB
- Generates `B426.tpfdf.dfdl.xsd` (DFDL schema file) and `B426.adbi.xml` (collection descriptor)
- Collection descriptor (`adbi.xml`) lets us:
  - Name the collection, indexes, and LRECs
  - Set up automatic indexing rules
  - Filter LRECs out of the collection
- DFDL schema file (`tpfdf.dfdl.xsd`) lets us:
  - Define the format of fields within each LREC
  - Define the format of indexes' algorithm strings

Customizing collection descriptors

- When you create the collection descriptor, it is filled with placeholder values.
  - Collection name is the DBDEF macro name. LRECs are named "lrec80", "lrec90", etc.
- You can change these names to be more descriptive. For example…

```
<tns:Indexes>
  <tns:Index name="PnrByName" number="0" length="33" readOnly="false" description="PNRs by Name"/>
  <tns:Index name="PnrByNumber" number="1" length="8" readOnly="false" description="PNRs by Number"/>
</tns:Indexes>

<tns:Lrecs>
  <tns:Lrec name="PassengerNumberRecord" id="70" />
  <tns:Lrec name="PassengerNameRecord" id="80" />
  <tns:Lrec name="AddressRecord" id="90" />
  <tns:Lrec name="FlightHistoryRecord" id="A0" />
  <tns:Lrec name="PassengerEmailRecord" id="B0" />
</tns:Lrecs>
```

A note on DFDL

- z/TPF support for MongoDB also requires a DFDL schema file for each collection.
- Modify the DFDL schema to specify the names and data types of each field in your LRECs and indexes' algorithm strings.
- These DFDL schemas support all of the same DFDL features as any other DFDL schema you may use.
- Unfortunately, we don't have time to dig deep into DFDL... but here's an example of modified field names.

```
<xs:complexType name="PassengerNumberRecord">
  <xs:sequence>
    <xs:element name="PassengerNumber" type="xs:long" dfdl:length="8" default="0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PassengerNameRecord" tddt:lrecId="80">
  <xs:sequence>
    <xs:element name="PassengerName" type="xs:string" dfdl:length="25"
                nillable="true" dfdl:useNilForDefault="yes" />
  </xs:sequence>
</xs:complexType>
```

Starting the MongoDB server

- Use common deployment to load both the collection descriptor and DFDL schema file to z/TPF.
- Both files are automatically deployed; there is no need to run `ZMDES DEPLOY` commands for them.
- z/TPF's MongoDB server is like any other server managed through `ZINET`.
  - `zinet add s-mongo model-daemon pgm-cads xparm-`*`options`*
- XPARM options include (but are not limited to):
  - IP address / port to listen on
  - Send and receive buffer sizes
  - SSL configuration
  - Timeout options
- Start the server:
  - `zinet start s-mongo`

Client-Side Usage

Using MongoDB drivers with z/TPF

- z/TPF support for MongoDB is based on MongoDB version 2.6.
- MongoDB v2.6 is supported by the latest versions of the MongoDB drivers for these languages:
  - C#
  - Go
  - Java
  - Node.js
  - Python
  - Ruby
  - Scala

z/TPF MongoDB document format

- Documents are formatted as a series of arrays of LRECs, ordered by LREC ID.
- Here's what a document returned by z/TPF support for MongoDB might look like:

```
{
  "PassengerNumberRecord" : [
    {
      "PassengerNumber" : NumberLong(131509348)
    }
  ],
  "PassengerNameRecord" : [
    {
      "PassengerName" : "DOROTHY GALE             "
    }
  ],
  "_seq" : 1,
  "_id" : ObjectId("000000000000000018542ba6")
}
```

Querying documents

- When you search for documents with z/TPF support for MongoDB, you must search by an index or a file address.
- This command demonstrates locating a document by the `PnrByName` index:
  - `db.PNR.find({"_index.PnrByName": { { "name": "DOROTHY GALE" } } })`
- If you have the subfile's prime file address, you can turn the file address into an ObjectID and query that way:
  - `db.PNR.find({"_id": ObjectID("0000000012345678ABCDEF00")})`
  - The first 4 bytes (8 digits) should be 0's. The remaining 8 bytes (16 hex digits) are the prime file address.

Inserting and finding documents

– This Python example demonstrates inserting a document and then retrieving it.

```
from pymongo import MongoClient
client = MongoClient('my_tpf_ip_address')
pnr_coll = client.get_database('tpfdf').get_collection('PNR')

# The Python MongoDB driver allows you to build a MongoDB document from a Python dictionary!
pnr_to_insert = {
  "_index": {
    "PnrByNumber": { "number": 29 },
    "PnrByName": { "name": "DOROTHY GALE" }
  }
  "PassengerNumberRecord": [ { "PassengerNumber": 29 } ],
  "PassengerNameRecord": [ { "PassengerName": "DOROTHY GALE" } ]
}

# Insert the PNR. Because we included the _index field, this also indexes the PNR in TPFDF.
pnr_coll.insert(pnr_to_insert)
# Find the PNR.
found_pnr = pnr_coll.find_one({"_index.PnrByName": {"name": "DOROTHY GALE"}})
```

Projections

- On query operations, you can use projections to include or exclude specific elements from the document that is returned.
- You can explicitly include or exclude the _id field, the _seq field, and any LREC types.

```
from pymongo import MongoClient
client = MongoClient('my_tpf_ip_address')
pnr_coll = client.get_database('tpfdf').get_collection('PNR')

# Build a query request to find the PNR.
pnr_query = {"_index.PnrByName": {"name": "DOROTHY GALE"}}
# The returned document should only contain the _id and PassengerNumberRecord fields.
projection = {"_id": 1, "_seq": 0, "PassengerNumberRecord": 1}
found_pnr = pnr_coll.find_one(pnr_query, projection)
```

- The result:

```
{
  "PassengerNumberRecord" : [{"PassengerNumber" : NumberLong(131509348)}],
  "_id" : ObjectId("000000000000000018542ba6")
}
```

Replacing documents

- The simplest way to update a document is to completely replace it. This example shows how you can find a document, change one field, and issue a replace.
- Replace operations also support MongoDB's `upsert` option, which will insert the specified document if it doesn't already exist in the database.

```
from pymongo import MongoClient
client = MongoClient('my_tpf_ip_address')
pnr_coll = client.get_database('tpfdf').get_collection('PNR')
# Build a query request to find the PNR.
pnr_query = {"_index.PnrByName": {"name": "DOROTHY GALE"}}
found_pnr = pnr_coll.find_one(pnr_query)

# Change the name in the first PassengerNameRecord, and the index
found_pnr["PassengerNameRecord"][0]["PassengerName"] = "COWARDLY LION"
found_pnr["_index"]["PnrByName"]["name"] = "COWARDLY LION"

# Update the document, as a full replace
pnr_coll.update(pnr_query, found_pnr)
```

The $set operator

– The $set operator can come in handy to set an individual field in a document.

```
from pymongo import MongoClient
client = MongoClient('my_tpf_ip_address')
pnr_coll = client.get_database('tpfdf').get_collection('PNR')

# Build a query request that will be used to identify the PNR to update.
pnr_query = {"_index.PnrByName": {"name": "DOROTHY GALE"}}

# Build another dictionary representing the updates to perform
# This operation will update the Facts field in the first FactsRecord.
pnr_update = { "$set": {"FactsRecord.0.Facts": "Not in Kansas anymore..." } }

# Update the document
pnr_coll.update(pnr_query, pnr_update)
```

The `$push` and `$pull` operators

– You can also update documents by adding (pushing) or removing (pulling) entire LRECs.

– You can perform a combination of `$set`, `$push`, and `$pull` operations at once.

```
from pymongo import MongoClient
client = MongoClient('9.57.13.68')
pnr_coll = client.get_database('tpfdf').get_collection('PNR')

pnr_query = {"_index.PnrByName": {"name": "DOROTHY GALE"}}

# Pull an LREC and push an LREC in the same operation
pnr_update = {
  "$pull" : { "AddressRecord": { "Address" : "12 Oak Road, Anytown, Kansas" } },
  "$push" : { "FactsRecord"  : { "Facts": "Not in Kansas anymore..." } }
}

# Update the document
pnr_coll.update(pnr_query, pnr_update)
```

Authentication and Authorization

User security and authorization

- z/TPF support for MongoDB uses a **user security database** to limit access to particular collections to specific users.

- Each MongoDB client can use a different user ID with different permissions.

- Each user can have many roles.

- IBM provides three built-in roles:

  - **read**: Allows the user to read documents from all collections.

  - **readWrite**: Allows the user to read, insert, update, and delete documents in all collections.

  - **userAdmin**: Allows the user to create, drop, update, and display users and roles.

Custom roles

– The z/TPF user security database also allows you to create custom roles.

– Each custom role has a number of **privileges**. A privilege consists of a **resource** and a number of **actions**.

- A **resource** is a database/collection pair (the only currently supported database is `tpfdf`)
- The available **actions** are "find", "insert", "update", and "remove".

– For example, the following MongoDB command creates a role that has full permissions on the PNR collection, and is only allowed to find on the Security collection.

```
db.createRole({ role: "updatePnrsReadSecurity",
  privileges: [
   { resource: { db: "tpfdf", collection: "PNR”},
                 actions: [ "find", "insert", “update”, “remove”] },
   { resource: { db: "tpfdf", collection: "Security”},
                 actions: [ "find"] },
```

Managing users and roles

- Any MongoDB user with the `userAdmin` role can use the standard MongoDB commands to read and update roles and users in the user security database.
  - `createUser(), dropUser(), updateUser(), getUser()`
  - `createRole(), dropRole(), getRole()`
- Operators can use the `ZROLE` and `ZRUSR` commands to manage roles and users, respectively.

Authentication

- If you specify the `--auth` parameter as an `XPARM` parameter on the MongoDB `ZINET` server definition, each user that tries to connect to the MongoDB server will need to be authenticated.
- There are two ways to authenticate a user upon login:
  - If you specify a password when creating a user, MongoDB will authenticate the user against the user security database.
  - If you do not specify a password, MongoDB will call the UATH user exit to authenticate the user.

Advanced Topics

Automatic indexing

- In previous examples, we've been managing the subfile's indexes ourselves, by specifying the `_index` field.
- If we set up **automatic indexing** rules, we do not have to manage the indexes.
- When we add, remove, or change fields that have automatic indexing rules, z/TPF support for MongoDB will keep the z/TPFDF index records synchronized with the data in the subfile.
- Automatic indexing rules are set in the collection descriptor.
- You can only define automatic indexing rules for index fields that have a direct correlation to fields in specific LRECs.
- Each index's automatic indexing rules can only use data from one LREC type.
  - For example, you cannot set up an automatic indexing rule that uses data from both a PassengerNumberRecord and a PassengerNameRecord.

Custom indexes

- Typically, indexes to a file are determined by the path parameters on the DBDEF macro.
  - This allows z/TPFDF to manage the indexes without requiring applications to manipulate the index records.
- However, if your database's indexes are managed manually, you can define custom indexes in z/TPF support for MongoDB.
  - With custom indexes, you can write your own code to index, deindex, and locate subfiles that do not use the standard z/TPFDF indexing support.
- Define custom indexes by adding them to the collection descriptor.
- Write your code to index, deindex, and locate subfiles with your custom indexes in the UCAD user exit.

Filtered collections

- Setting up **filtered collections** for z/TPF support for MongoDB allows you to have multiple different views of the same z/TPFDF subfiles.
- Each filtered collection is defined with its own collection descriptor.
- You can set up several filtered collections for the same z/TPFDF file.
- Why?
  - Omit LRECs that contain sensitive information
  - Omit LREC types that aren't needed by the client
  - Format data differently depending on the client (using a different DFDL schema for each collection)
  - Only show LRECs that "belong" to the user

Logging

- z/TPF support for MongoDB can log requests and responses, and send the logs to another server.
- To start logging MongoDB requests and responses:

1. Set up a log receiver server. We recommend using Logstash. Refer to the support page "Using logstash with MongoDB Logging for z/TPF (PJ44239)".
2. Define a high speed connector endpoint group for the log receiver server, with a group name of `IMONGLOG`.
3. Use the `ZMONG LOG SET` command to specify which requests are logged for each collection.
4. Use the `ZMONG LOG START` command to start collecting logs.

Operations considerations

- Use ZSTAT SYSHEAP to monitor memory used by z/TPF support for MongoDB.
  - Owner names:
    - `IMONGO.SYSTEM.`*`pbi`*, where *pbi* is the subsystem's program base index.
    - `IMONGO.SERVER.`*`port`*, where *port* is the port that the MongoDB server is listens to.
    - `IMONGO.SOCKET.`*`socketDesc`*, where *socketDesc* is a socket descriptor.
    - `IMONGO.CURSOR.`*`socketDesc`*, where *socketDesc* is a socket descriptor.
- Use the TCP/IP network services database file (`/etc/services`) to limit the number of concurrent connections or number of messages to the MongoDB server.

Default keys

– In z/TPFDF, default keys are used to maintain the organization of the subfile when you add LRECs to the subfile.

– You must have default keys defined for the z/TPFDF file in order to perform updates using z/TPF support for MongoDB.

– There are two ways to define default keys:

  - Using the z/TPFDF DBDEF macro.
  - Using the collection descriptor.

– Example of defining default keys in the collection descriptor:

```
<tns:Collection reference="DR25BI" name="Number" collectionId="B425"
PKOrg="Ascending"                        dfdlfile="Number.tpfdf.dfdl.xsd">
```

User Exits

UCAD

- `UCAD_locate`: Locate a document using a custom index
- `UCAD_index`: Create a custom index for a document
- `UCAD_deindex`: Remove a custom index for a document
- `UCAD_command`: Process a custom command

UMON

- `UMON_pre_request`: Called before a request is processed.
  - Allows you to accept or reject the request.
  - One use could be checking system resources before allowing request to continue.
- `UMON_post_request`: Called after a request is processed.
  - Could be used to collect stats on how many requests succeeded or failed.

UATH

- `UATH_mongodb_cr`: Process MONGODB-CR authentication.
- `UATH_mongodb_plain`: Process authentication with password in plain text.
- `UATH_filtered_collection`: Authorize a user against a filtered collection.
  - Potential use: restrict sensitive LREC types to privileged users.

Quiz

Question 1

- True or False: z/TPF support for MongoDB is a port of the standard open-source MongoDB database management system.

Question 1

- True or False: z/TPF support for MongoDB is a port of the standard open-source MongoDB database management system.
  - FALSE!
  - z/TPF support for MongoDB is an interface layer that allows MongoDB clients to interact with existing z/TPFDF databases.

Question 2

- Which of the following types of data can you manipulate by using z/TPF support for MongoDB? Select all that apply.
  - A: Traditional z/TPF find/file records.
  - B: z/TPFDF subfiles and LRECs.
  - C: Standard MongoDB BSON documents stored on z/TPF.
  - D: The z/TPF file system, using GridFS.

Question 2

- Which of the following types of data can you manipulate by using z/TPF support for MongoDB? Select all that apply.
  - A: Traditional z/TPF find/file records.
    - FALSE!
  - B: z/TPFDF subfiles and LRECs.
    - TRUE!
  - C: Standard MongoDB BSON documents stored on z/TPF.
    - FALSE!
  - D: The z/TPF file system, using GridFS.
    - FALSE!

Question 3

- What is the primary way to create collections for each z/TPFDF file that you want to access using z/TPF support for MongoDB?
  - A: The `ZMONG` command can be used to create collections for z/TPFDF files.
  - B: When you try to access a z/TPFDF file, the MongoDB server will create the corresponding collection for you.
  - C: MongoDB collections are generated by the DBDEF macro for each z/TPFDF file.
  - D: Customize common deployment descriptors that are generated by the `ZUDFM DESCRIPTOR` command.

Question 3

- What is the primary way to create collections for each z/TPFDF file that you want to access using z/TPF support for MongoDB?
- The correct answer is:
  - D: Customize common deployment descriptors that are generated by the `ZUDFM DESCRIPTOR` command.

Question 4

- Which of the following are required to read and update a collection with z/TPF support for MongoDB? (Select all that apply).
  - A: A z/TPFDF database.
  - B: A DFDL schema file.
  - C: A collection descriptor.
  - D: A filtered collection.
  - E: Default key definitions on either the DBDEF macro or collection descriptor.

Question 4

- Which of the following are required to read and update a collection with z/TPF support for MongoDB? (Select all that apply).
  - A: A z/TPFDF database.
    - TRUE!
  - B: A DFDL schema file.
    - TRUE!
  - C: A collection descriptor.
    - TRUE!
  - D: A filtered collection.
    - FALSE!
  - E: Default key definitions on either the DBDEF macro or collection descriptor.
    - TRUE!

Question 5

- True or False: You can manage MongoDB users, roles, and permissions by loading a configuration file to common deployment.

Question 5

- True or False: You can manage MongoDB users, roles, and permissions by loading a configuration file to common deployment.
  - FALSE!
  - MongoDB users, roles, and permissions are managed by MongoDB user and role management commands, as well as the ZRUSR and ZROLE commands.

Question 6

- Which of the following are potential use cases for filtered collections? (Select all that apply).
  - A: Restrict sensitive data to privileged users.
  - B: Reduce size of messages by omitting LRECs that clients do not need.
  - C: Allow different clients to connect on different ports.
  - D: Only allow users to access data that "belongs" to each user.

Question 6

- Which of the following are potential use cases for filtered collections? (Select all that apply).
  - A: Restrict sensitive data to privileged users.
    - TRUE!
  - B: Reduce size of messages by omitting LRECs that clients do not need.
    - TRUE!
  - C: Allow different clients to connect on different ports.
    - FALSE!
  - D: Only allow users to access data that "belongs" to each user.
    - TRUE!

Question 7

– How do you configure logging options for z/TPF support for MongoDB?

  - A: Use the `ZMONG LOG SET` command.
  - B: Create a customized `log4j2.xml` file that suits your logging needs.
  - C: You can't; z/TPF support for MongoDB either logs everything or nothing.

Question 7

- How do you configure logging options for z/TPF support for MongoDB?
- The correct answer is:
  - A: Use the `ZMONG LOG SET` command.

Question 8

– True or False: All subfiles that you use with z/TPF support for MongoDB must have indexes that are managed by z/TPFDF.

Question 8

- True or False: All subfiles that you use with z/TPF support for MongoDB must have indexes that are managed by z/TPFDF.
- The correct answer is:
  - FALSE!
  - You can use custom indexes to handle indexing and locating subfiles that do not have z/TPFDF-managed indexes.
  - z/TPF support for MongoDB does not require any indexing when working with fixed-file z/TPFDF databases.

Thank You!

Questions or Comments?

Trademarks

Notes

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

This presentation and the claims outlined in it were reviewed for compliance with US law. Adaptations of these claims for use in other geographies must be reviewed by the local country counsel for compliance with local laws.

Icons created by Smashicons & Pause08, from www.flaticon.com.