SERVER-SIDE

MICROSERVICES II

DATABASE ACCESS

DATA SERIALIZATION

SECURITY, TELEMETRY & SCALABILITY

VERSION

v1.0 23 September 2021

ACKNOWLEDGMENTS

THANKS TO:

- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

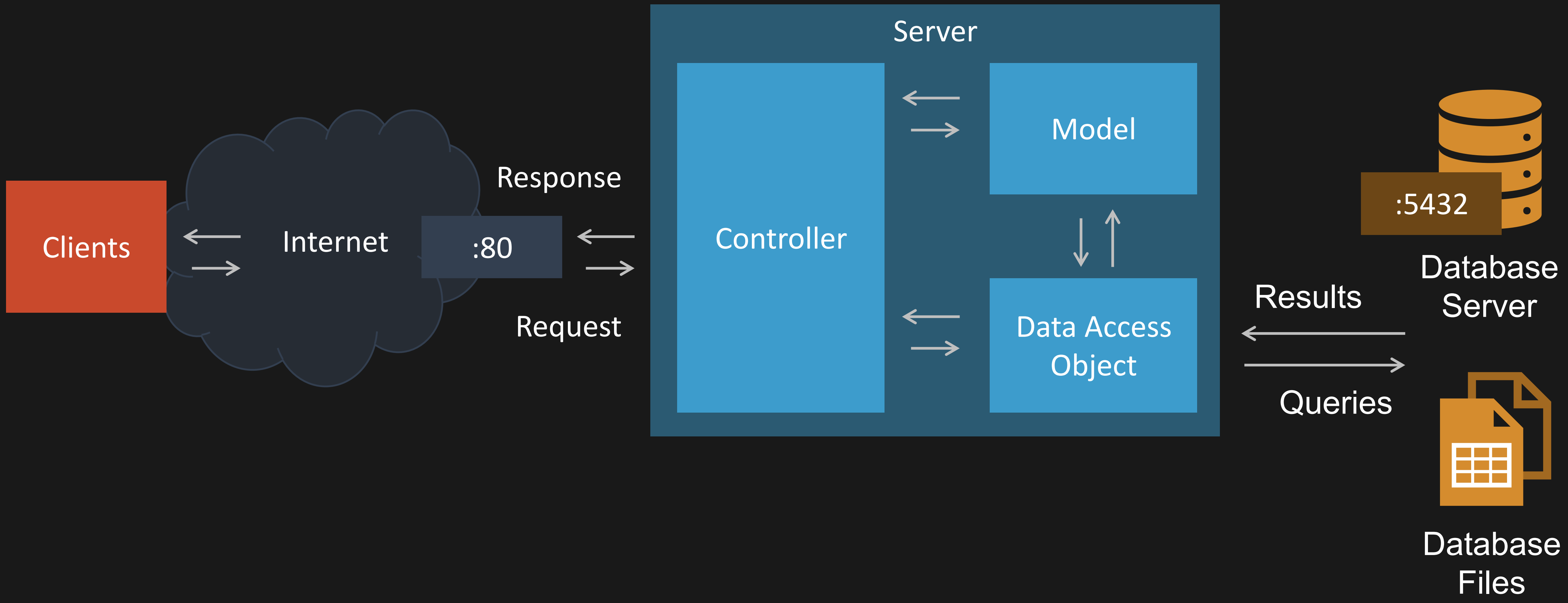
PRINTABLE VERSION OF THE TALK

Download PDF

DATABASE ACCESS

CONNECTING MICROSERVICES TO THE DATABASES

DATABASES IN THE MICROSERVICES ARCHITECTURE



WHAT IS A DATABASE?

A **database** is an organized collection of structured information or data. A database is usually controlled by a **database management system (DBMS)**. Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a **database system** or simply a database.

ID	SURNAME	GIVENNAME	GPA	YEARADMITTED
200715420	Andrews	Kelly	7.8	2007
200768902	Bartlett	Jasmine	7.1	2007
200420801	Golden	Dante	8.1	2004
200746650	Higgins	Alejandra	7.1	2007
200929837	Jones	Evan	8.6	2009
200721627	Shelton	Diana	9.0	2007

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient.

The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use **structured query language (SQL)** for writing and querying data.

```sql
SELECT S.ID, S.Surname, S.GivenName, S.GPA, S.YearAdmitte
FROM Roumani.SIS as S
WHERE S.Major = 'Computer Science'
  AND S.GPA >= 7;
```

Source: What Is a Database? - Oracle

JDBC: JAVA DATABASE CONNECTIVITY

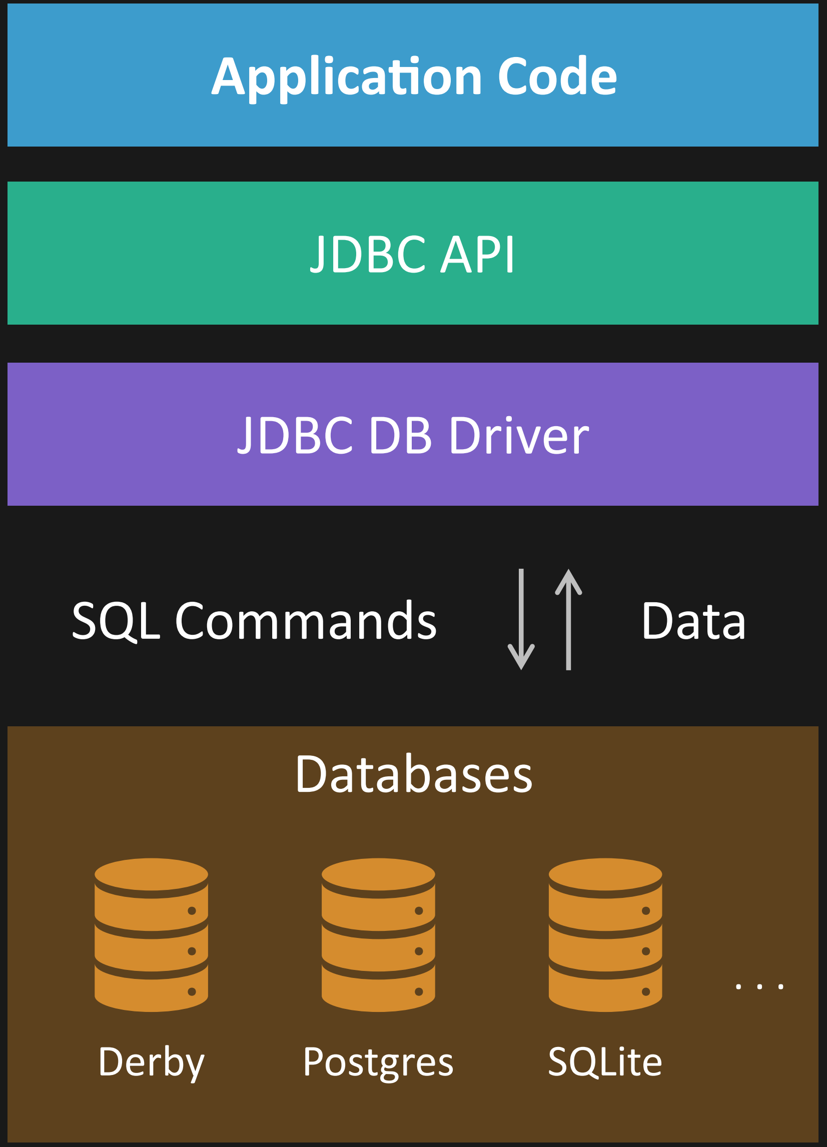
JDBC (Java Database Connectivity) is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database. It is oriented toward relational databases. It has been part of the Java Standard library since JDK 1.1 in 1997 and was one of the first components developed for the Java persistence layer.

Source: What is JDBC? Introduction to Java Database Connectivity

JDBC offers a programming-level interface that handles the mechanics of Java applications communicating with a database or RDBMS.

The JDBC interface consists of two layers:

1. The JDBC API supports communication between the Java application and the JDBC manager.
2. The JDBC driver supports communication between the JDBC manager and the database driver.



USING JDBC

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        while (rs.next()) {
          String province = rs.getString("province");
          String code     = rs.getString("code");
          String type     = rs.getString("type");
          double gst      = rs.getDouble("gst");
          double pst      = rs.getDouble("pst");
          log.printf("Taxes in %s (%s): %s, %.2f%%, %.2f%%\n", province, code, type, gst, pst);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        while (rs.next()) {
          String province = rs.getString("province");
          String code     = rs.getString("code");
          String type     = rs.getString("type");
          double gst      = rs.getDouble("gst");
          double pst      = rs.getDouble("pst");
          log.printf("Taxes in %s (%s): %s, %.2f%%, %.2f%%\n", province, code, type, gst, pst);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        while (rs.next()) {
          String province = rs.getString("province");
          String code     = rs.getString("code");
          String type     = rs.getString("type");
          double gst      = rs.getDouble("gst");
          double pst      = rs.getDouble("pst");
          log.printf("Taxes in %s (%s): %s, %.2f%%, %.2f%%\n", province, code, type, gst, pst);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        while (rs.next()) {
          String province = rs.getString("province");
          String code     = rs.getString("code");
          String type     = rs.getString("type");
          double gst      = rs.getDouble("gst");
          double pst      = rs.getDouble("pst");
          log.printf("Taxes in %s (%s): %s, %.2f%%, %.2f%%\n", province, code, type, gst, pst);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        while (rs.next()) {
          String province = rs.getString("province");
          String code     = rs.getString("code");
          String type     = rs.getString("type");
          double gst      = rs.getDouble("gst");
          double pst      = rs.getDouble("pst");
          log.printf("Taxes in %s (%s): %s, %.2f%%, %.2f%%\n", province, code, type, gst, pst);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.
- Store the result data into a *Java bean*, a *Plain old Java Object (POJO)*. More on Java Beans in a bit.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = 'ON'";

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        TaxBean bean = new TaxBean();
        while (rs.next()) {
          bean.setName(rs.getString("province"));
          bean.setCode(rs.getString("code"));
          bean.setType(rs.getString("type"));
          bean.setGst(rs.getDouble("gst"));
          bean.setPst(rs.getDouble("pst"));
          log.println(bean);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.
- Store the result data into a *Java bean*, a *Plain old Java Object (POJO)*. More on Java Beans in a bit.
- For queries with user input, use `PreparedStatements`. String concatenation is insecure and can lead to SQL injection attacks.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";
    String province = args[0];

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = " + province; // <!-- SQL injection attack !!!

      try (
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery(query);
      ) {
        TaxBean bean = new TaxBean();
        while (rs.next()) {
          bean.setName(rs.getString("province"));
          bean.setCode(rs.getString("code"));
          bean.setType(rs.getString("type"));
          bean.setGst(rs.getDouble("gst"));
          bean.setPst(rs.getDouble("pst"));
          log.println(bean);
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.
- Store the result data into a *Java bean*, a *Plain old Java Object (POJO)*. More on Java Beans in a bit.
- For queries with user input, use `PreparedStatements`. String concatenation is insecure and can lead to SQL injection attacks.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";
    String province = args[0];

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE code = ?";

      try (PreparedStatement statement = connection.prepareStatement(query)) {
        statement.setString(1, province);
        try (ResultSet rs = statement.executeQuery()) {
          TaxBean bean = new TaxBean();
          while (rs.next()) {
            bean.setName(rs.getString("province"));
            bean.setCode(rs.getString("code"));
            bean.setType(rs.getString("type"));
            bean.setGst(rs.getDouble("gst"));
            bean.setPst(rs.getDouble("pst"));
            log.println(bean);
          }
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.
- Store the result data into a *Java bean*, a *Plain old Java Object (POJO)*. More on Java Beans in a bit.
- For queries with user input, use `PreparedStatements`. String concatenation is insecure and can lead to SQL injection attacks.
- Handle multiple results by storing them in a data structure like a List.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";
    double pst  = Double.parseDouble(args[0]);

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE pst > ?";

      try (PreparedStatement statement = connection.prepareStatement(query)) {
        statement.setString(1, province);
        try (ResultSet rs = statement.executeQuery()) {
          ArrayList<TaxBean> list = new ArrayList<>();
          while (rs.next()) {
            TaxBean bean = new TaxBean();
            bean.setName(rs.getString("province"));
            bean.setCode(rs.getString("code"));
            bean.setType(rs.getString("type"));
            bean.setGst(rs.getDouble("gst"));
            bean.setPst(rs.getDouble("pst"));
            list.add(bean);
          }
          for (TaxBean bean : list) {
            log.println(bean);
          }
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

USING JDBC

- Specify the database's URL.

  `jdbc:sqlite:/path/to/database/file`
  `jdbc:derby://host:port/name;credentials`

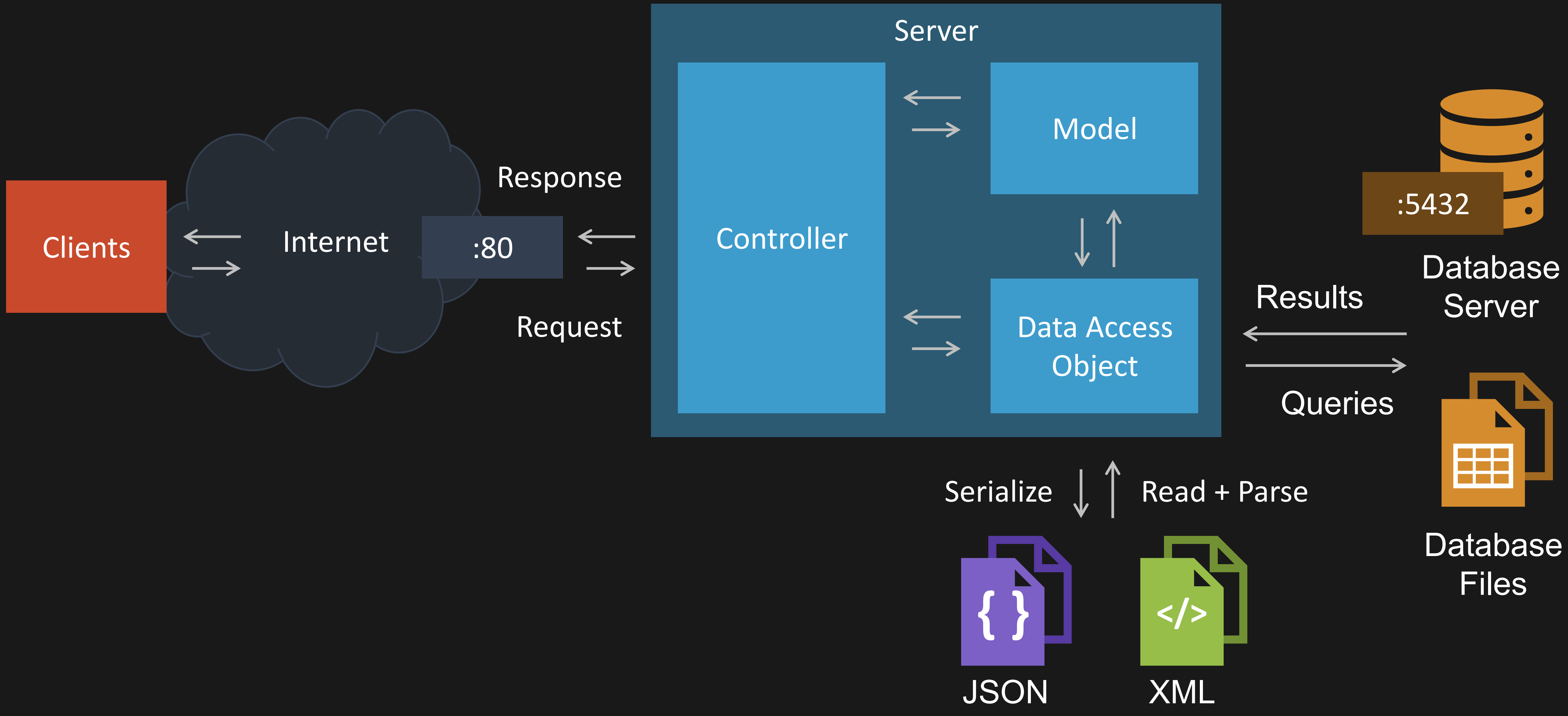
- Get a connection to the database.
- Create a SQL statement.
- Execute the query and see its result.
- Store the result data into a *Java bean*, a *Plain old Java Object (POJO)*. More on Java Beans in a bit.
- For queries with user input, use `PreparedStatements`. String concatenation is insecure and can lead to SQL injection attacks.
- Handle multiple results by storing them in a data structure like a List.
- Eclipse has tools (Data Source Explorer, SQL Scrapbook) to explore the database and test queries. For SQLite, I prefer CLI –see https://sqlite.org/cli.html. Similarly, Derby provides the CLI command `ij`.

```java
class SqliteJDBCExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    String home = System.getProperty("user.home");
    String url  = "jdbc:sqlite:" + home + "/4413/pkg/sqlite/Models_R_US.db";
    double pst  = Double.parseDouble(args[0]);

    try (Connection connection = DriverManager.getConnection(url)) {
      log.printf("Connected to database: %s\n", connection.getMetaData().getURL());
      String query = "SELECT * FROM Tax WHERE pst > ?";

      try (PreparedStatement statement = connection.prepareStatement(query)) {
        statement.setString(1, province);
        try (ResultSet rs = statement.executeQuery()) {
          ArrayList<TaxBean> list = new ArrayList<>();
          while (rs.next()) {
            TaxBean bean = new TaxBean();
            bean.setName(rs.getString("province"));
            bean.setCode(rs.getString("code"));
            bean.setType(rs.getString("type"));
            bean.setGst(rs.getDouble("gst"));
            bean.setPst(rs.getDouble("pst"));
            list.add(bean);
          }
          for (TaxBean bean : list) {
            log.println(bean);
          }
        }
      }
    } catch (SQLException e) {
      log.println(e);
    } finally {
      log.println("Disconnected from database.");
    }
  }
}
```

DATA SERIALIZATION

Serialization is the process of converting the state information of an object instance into a binary or textual form to persist into a storage medium or transported over a network.

DATA SERIALIZATION IN THE ARCHITECTURE



JAVA BEAN

A `JavaBean` is just a Plain old Java Object that means the following requirements:

1. All properties are private (use getters/setters).
2. A public no-argument constructor.
3. Implements `Serializable`.

```java
public class TaxBean implements Serializable {
  private String name;
  private String code;
  private String type;
  private double pst;
  private double gst;

  public TaxBean() { }

  public String getName() { return name; }
  public String getCode() { return code; }
  public String getType() { return type; }
  public double getPst()  { return pst; }
  public double getGst()  { return gst; }

  public void setName(String name) { this.name = name; }
  public void setCode(String code) { this.code = code; }
  public void setType(String type) { this.type = type; }
  public void setPst(double pst)   { this.pst  = pst; }
  public void setGst(double gst)   { this.gst  = gst; }
}
```

DATA FORMATS

XML

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<menu id="breakfast-menu">
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>Two of our famous Belgian Waffles with plenty...</description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>Light Belgian waffles covered with strawberries...</descriptio
    <calories>900</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>Two eggs, bacon or sausage, toast, and our...</description>
    <calories>950</calories>
  </food>
  ...
</menu>
```

JSON

JavaScript Object Notation (JSON) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays (or other serializable values).

```json
{
  "id": "breakfast-menu",
  "food": [{
    "name":        "Belgian Waffles",
    "price":       "$5.95",
    "description": "Two of our famous Belgian Waffles with plenty...",
    "calories":    650
  }, {
    "name":        "Strawberry Belgian Waffles",
    "price":       "$7.95",
    "description": "Light Belgian waffles covered with strawberries...",
    "calories":    900
  }, {
    "name":        "Homestyle Breakfast",
    "price":       "$6.95",
    "description": "Two eggs, bacon or sausage, toast, and our...",
    "calories":    950
  }, ... ]
}
```

XML SERIALIZATION

Java Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. JAXB also provides a way to generate XML schema from Java objects.

Sources: Lesson: Introduction to JAXB; and Java Architecture for XML Binding (JAXB) (by Ed Ort and Bhakti Mehta, March 2003).

```java
public class ToXMLExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxBean bean = ...;

    try (ByteArrayOutputStream output = new ByteArrayOutputStream()) {
      JAXBContext context = JAXBContext.newInstance(TaxBean.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
      m.marshal(bean, output);

      log.println(out);
    } catch (Exception e) {
      log.println(e);
    }
    ...
  }
}

@XMLRootElement(name = "tax")
public class TaxBean implements Serializable {
  ...
}
```

XML SERIALIZATION

Java Architecture for XML Binding (JAXB) provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for unmarshalling (reading) XML instance documents into Java content trees, and then marshalling (writing) Java content trees back into XML instance documents. JAXB also provides a way to generate XML schema from Java objects.

Sources: Lesson: Introduction to JAXB; and Java Architecture for XML Binding (JAXB) (by Ed Ort and Bhakti Mehta, March 2003).

```java
public class ToXMLCollectionExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxCollection collection = ...;

    try (ByteArrayOutputStream output = new ByteArrayOutputStream()) {
      JAXBContext context = JAXBContext.newInstance(TaxCollection.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
      m.marshal(collection, output);

      res.println(out);
    } catch (Exception e) {
      log.println(e);
    }
    ...
  }
}

@XMLRootElement(name = "tax")
public class TaxBean implements Serializable {
  ...
}

@XmlRootElement(name="taxes")
public class TaxCollection implements Serializable {
  private List<TaxBean> taxes;
  public TaxCollection() { }

  @XmlElement(name="tax")
  public List<TaxBean> getTaxes() { return taxes; }
  public void setTaxes(List<TaxBean> taxes) { this.taxes = taxes; }
}
```

XML DE-SERIALIZATION

Sources: Introduction to JAXB - Basic Examples.

```java
public class FromXMLExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxBean bean;
    String xml = ...;

    try (InputStream stream = new ByteArrayInputStream(xml.getBytes())) {
      JAXBContext context = JAXBContext.newInstance(TaxBean.class);
      Unmarshaller u = context.createUnmarshaller();
      bean = (TaxBean) u.unmarshal(input);
      ...
      log.println(bean);
    } catch (Exception e) {
      log.println(e);
    }
    ...
  }
}
```

```java
public class FromXMLCollectionExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxCollection collection;
    String xml = ...;

    try (InputStream stream = new ByteArrayInputStream(xml.getBytes())) {
      JAXBContext context = JAXBContext.newInstance(TaxCollection.class);
      Unmarshaller u = context.createUnmarshaller();
      collection = (TaxCollection) u.unmarshal(input);
      ...
      log.println(collection);
    } catch (Exception e) {
      log.println(e);
    }
    ...
  }
}
```

JSON SERIALIZATION

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source code of.

- Provides simple `toJson` and `fromJson` methods to convert Java objects to JSON and vice-versa
- Allows pre-existing unmodifiable objects to be converted to and from JSON
- Extensive support of Java Generics
- Allows custom representations for objects
- Supports arbitrarily complex objects (with deep inheritance hierarchies and extensive use of generic types)

Sources: Gson - Github; and Gson - Quick Guide (TutorialsPoint).

```java
public class ToJSONExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxBean bean = ...;

    Gson gson   = new Gson();
    String json = gson.toJson(bean);

    log.println(json);
    ...
  }
}
```

```java
public class ToJSONCollectionExample {
  private static PrintStream log = System.out;
  public static void main(String[] args) {
    ...
    TaxCollection collection = ...;

    Gson gson   = new Gson();
    String json = gson.toJson(collection);

    log.println(json);
    ...
  }
}
```

JSON DE-SERIALIZATION

- It is perfectly fine (and recommended) to use private fields.
- There is no need to use any annotations to indicate a field is to be included for serialization and deserialization. All fields in the current class (and from all super classes) are included by default.
- If a field is marked `transient`, (by default) it is ignored and not included in the JSON serialization or deserialization.
- This implementation handles `null`s correctly.
  - While serializing, a `null` field is omitted from the output.
  - While deserializing, a missing entry in JSON results in setting the corresponding field in the object to its default value: null for object types, zero for numeric types, and false for booleans.
- If a field is synthetic, it is ignored and not included in JSON serialization or deserialization.
- Fields corresponding to the outer classes in inner classes, anonymous classes, and local classes are ignored and not included in serialization or deserialization.

```java
class BagOfPrimitives {
  private int value1 = 1;
  private String value2 = "abc";
  private transient int value3 = 3;
  BagOfPrimitives() {
    // no-args constructor
  }
}

// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> json is {"value1":1,"value2":"abc"}

// Deserialization
BagOfPrimitives obj2 = gson.fromJson(json, BagOfPrimitives.class);
// ==> obj2 is just like obj
```

Sources: Gson User Guide - Github.

This slide is intentionally left blank.

Return to Course Page.