J2EEBasic with Frameworks

- Same as last project, but using some common frameworks
- Same two webapps as last presentation
  - Hello webapp that just returns a hard-coded String of HTML with the current date inserted
  - Address webapp that lets you view/edit an address object, which is validated and persisted to an in-memory HSQL database
- This presentation focuses on using the frameworks
  - How to create the project
  - How to configure the frameworks

MVC

- MVC is used incorrectly in most every web app
- The model is not just plain beans – it is supposed to also encapsulate business logic
- This lack of encapsulation causes business logic to be randomly strewed about the controller, and is one of the primary reasons why web app projects devolve over time

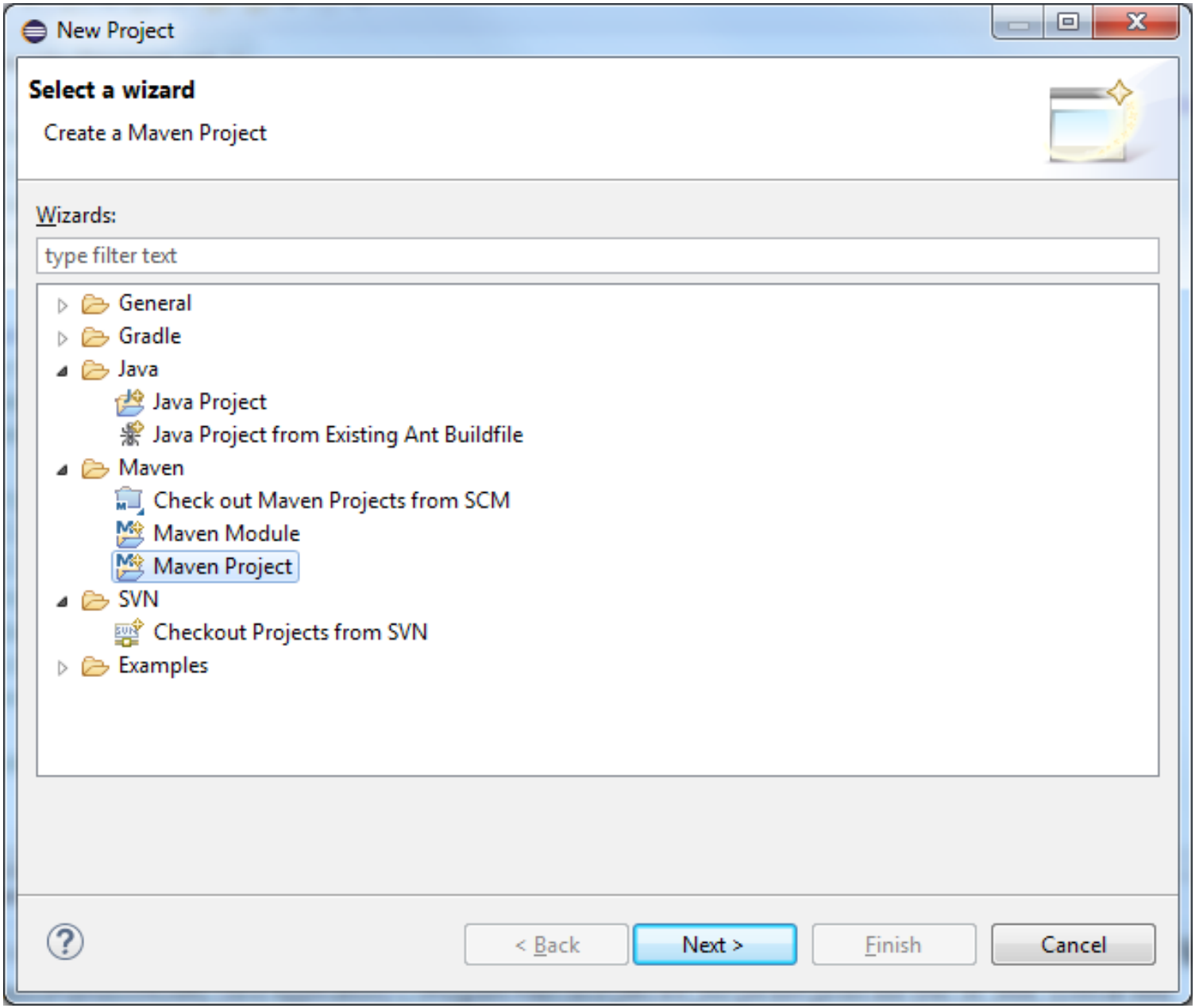
HSQL

- HyperSQL
- Mature SQL database, been around for years
- Has an in-memory database option, great for unit testing and simple example applications
- Has a "resources" option, a read-only database pre-filled with data that gets loaded into memory, good for unit testing
- Does not require any configuration, but a simple properties file can be used
- Easy to run and configure programmatically
- Single jar file

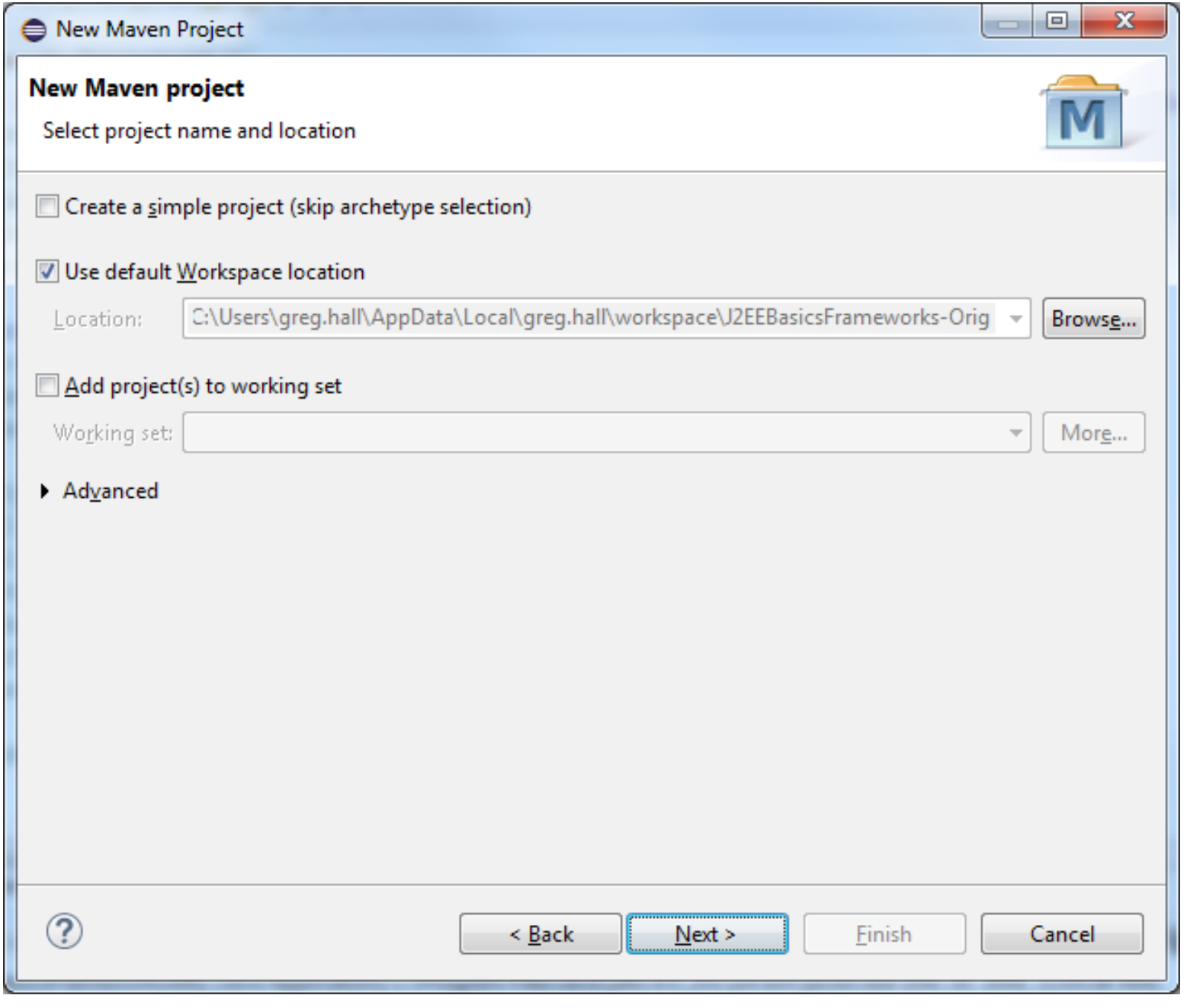
Maven

- Common build system to replace Ant
  - Gradle is the new shiny tool to replace Maven
  - Downloads dependencies - no libs in repo
  - Compiles code
  - Builds artifacts (jar file, war file, etc)
  - Imposes a specific directory structure
  - Understands variations like a web app versus a command line app
  - Can copy properties such as passwords from a local file in your home directory, to keep them out of the repo

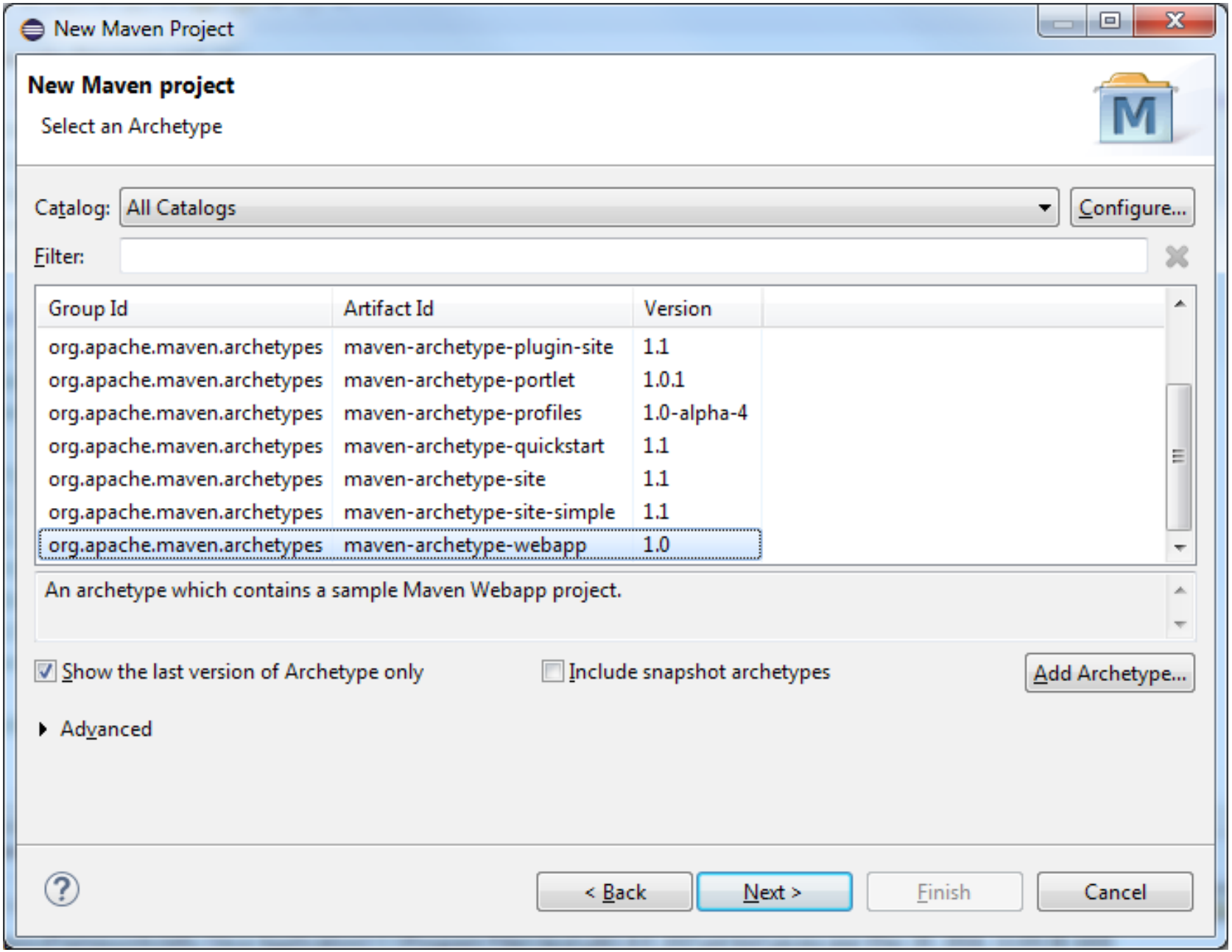
Maven


New Project
Select a wizard
Create a Maven Project
Wizards:
type filter text
General
Gradle
Java
Java Project
Java Project from Existing Ant Buildfile
Maven
Check out Maven Projects from SCM
Maven Module
Maven Project
SVN
Checkout Projects from SVN
Examples
< Back
Next >
Finish
Cancel

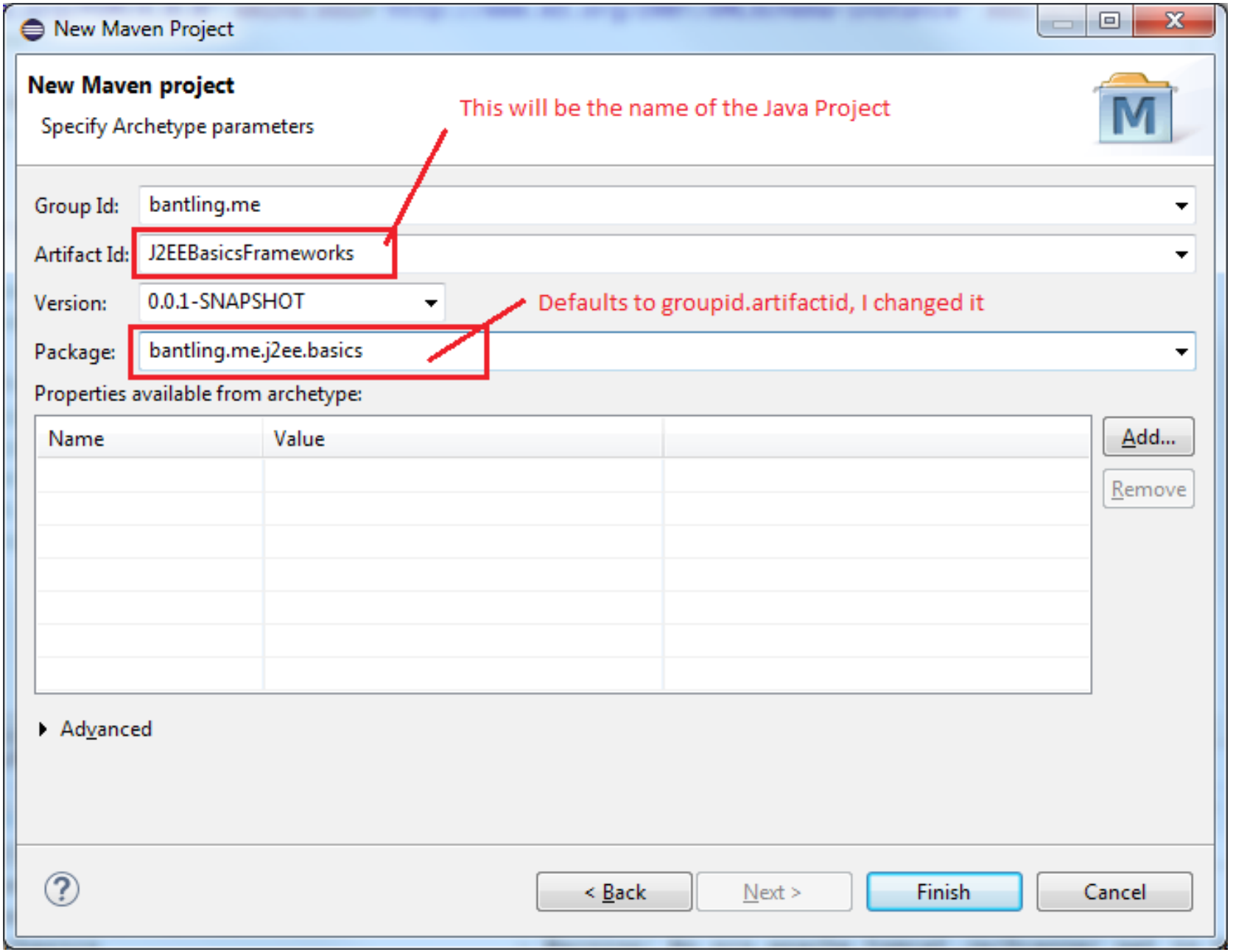
Maven


New Maven Project
New Maven project
Select project name and location
Create a simple project (skip archetype selection)
Use default Workspace location
Location:
C:\Users\greg.hall\AppData\Local\greg.hall\workspace\J2EEBasicsFrameworks-Orig
Browse...
Add project(s) to working set
Working set:
More...
Advanced
< Back
Next >
Finish
Cancel

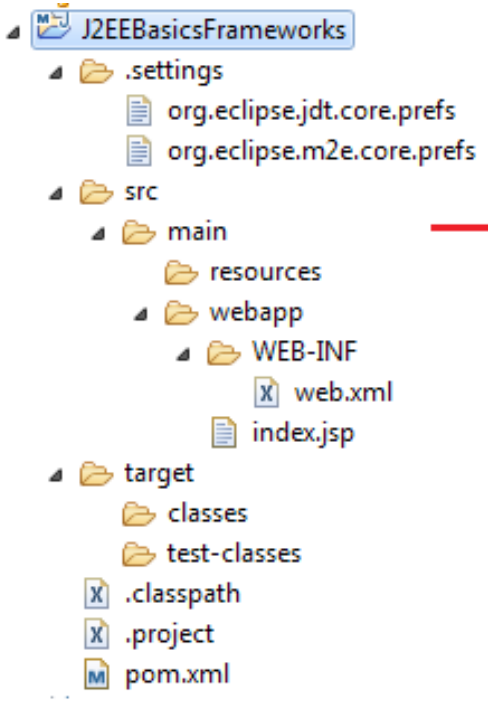
Maven


New Maven Project
New Maven project
Select an Archetype
Catalog: All Catalogs
Configure...
Filter:
Group Id
Artifact Id
Version
org.apache.maven.archetypes maven-archetype-plugin-site 1.1
org.apache.maven.archetypes maven-archetype-portlet 1.0.1
org.apache.maven.archetypes maven-archetype-profiles 1.0-alpha-4
org.apache.maven.archetypes maven-archetype-quickstart 1.1
org.apache.maven.archetypes maven-archetype-site 1.1
org.apache.maven.archetypes maven-archetype-site-simple 1.1
org.apache.maven.archetypes maven-archetype-webapp 1.0
An archetype which contains a sample Maven Webapp project.
Show the last version of Archetype only
Include snapshot archetypes
Add Archetype...
Advanced
< Back
Next >
Finish
Cancel

Maven

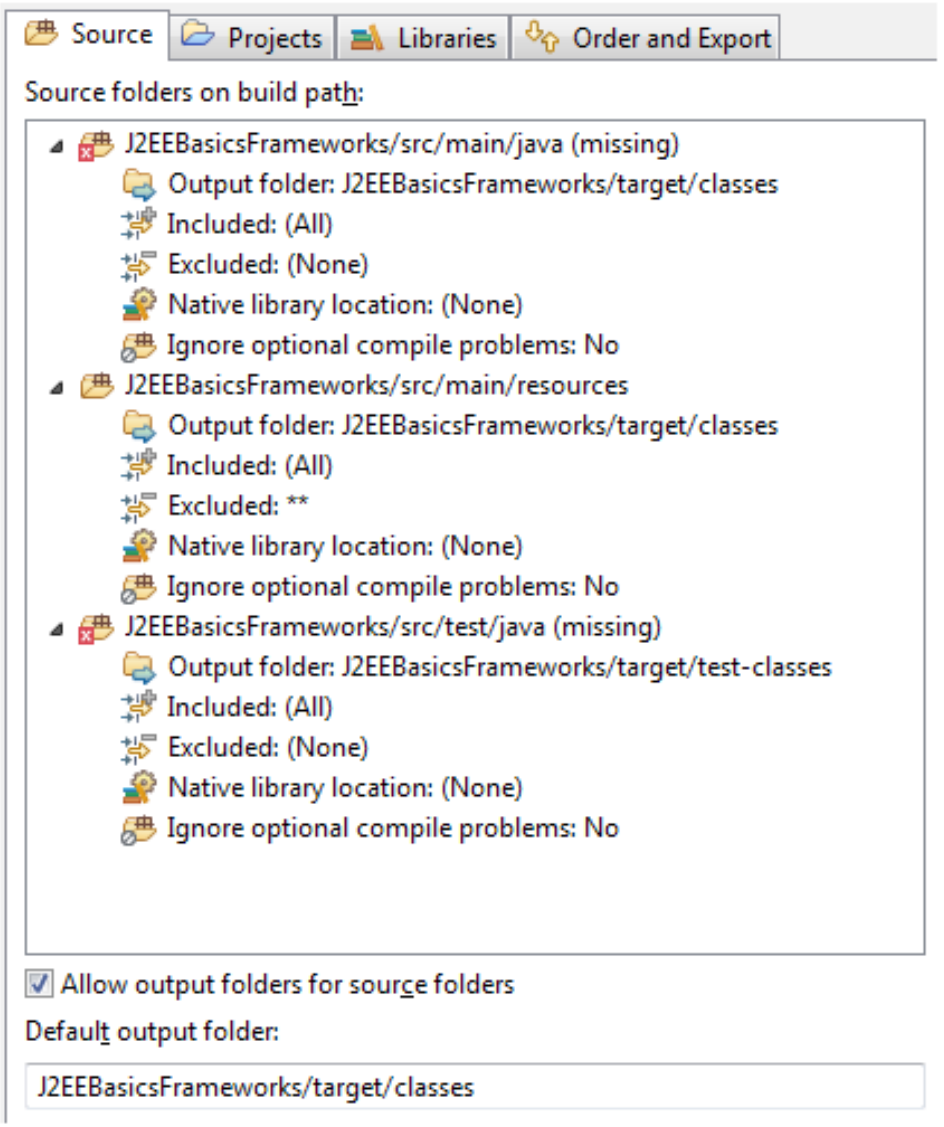

New Maven Project
New Maven project
Specify Archetype parameters
This will be the name of the Java Project
Group Id: bantling.me
Artifact Id: J2EEBasicsFrameworks
Version: 0.0.1-SNAPSHOT
Defaults to groupid.artifactid, I changed it
Package: bantling.me.j2ee.basics
Properties available from archetype:
Name
Value
Add...
Remove
Advanced
< Back
Next >
Finish
Cancel

Maven



This layout is missing obvious things like the Java package we specified, src/main/java, src/test/java, src/test/resources.

On the other hand, the build path already has entries for the missing src/main/java and src/test/java, just not for the src/test/resources.



Maven

The generated pom.xml file looks as follows:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>bantling.me</groupId>
  <artifactId>J2EEBasicsFrameworks</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>J2EEBasicsFrameworks Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>J2EEBasicsFrameworks</finalName>
  </build>
</project>
```

The resulting pom.xml file indicates:
- Maven version 4.0.0
- groupid and artifactid we specified
- create a war file
- version is 0.0.1-SNAPSHOT
- project name and url
- needs junit 3.8.1, only for testing
  - junit will not be in war file
- artifact will be called J2EEBasicsFrameworks.war

Maven

- pom.xml is "the" configuration file for Maven
- We don't need the name and url elements
- We do need:
  - Set Java compiler source and target (I chose 1.8)
  - Dependencies:
    - Jetty (web application container)
    - HSQL (in-memory SQL database)
    - JPA
    - SLF4J and Logback (logging)
    - Newer JUnit

Maven

Set Java compiler source and target to 1.8

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:>
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>bantling.me</groupId>
4    <artifactId>J2EEBasicsFrameworks-Orig</artifactId>
5    <version>0.0.1-SNAPSHOT</version>
6    <packaging>war</packaging>
7
8    <!-- Set Java compiler source/target versions -->
9
10   <properties>
11     <maven.compiler.source>1.8</maven.compiler.source>
12     <maven.compiler.target>1.8</maven.compiler.target>
13   </properties>
14
```

Maven

- Maven defaults to Java 1.5
- Certain changes to the POM, like adding the java compiler, require manually updating Maven
  - Right click Project
  - Maven > Update Project...
- Other changes to the POM, like adding dependencies trigger an update automatically

Maven

Set variables for dependencies

- Often, there are multiple libraries for a given dependency that are all the same version

*J2EEBasicsFrameworks-Orig/pom.xml *J2EEBasicsFrameworks/pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" x
2
3    <properties>
4      <jetty.version>9.3.10.v20160621</jetty.version>
5      <hsql.version>2.3.1</hsql.version>
6      <slf4j.version>1.7.22</slf4j.version>
7      <logback.version>1.1.8</logback.version>
8      <junit.version>4.12</junit.version>
9    </properties>
10
11   <dependencies>
12
13     <!-- Jetty -->
14
15     <dependency>
16       <groupId>org.eclipse.jetty</groupId>
17       <artifactId>jetty-server</artifactId>
18       <version>${jetty.version}</version>
19     </dependency>
20
21     <dependency>
22       <groupId>org.eclipse.jetty</groupId>
23       <artifactId>apache-jsp</artifactId>
24       <version>${jetty.version}</version>
25     </dependency>
```

JPA

- Java Persistance Architecture
  - Abstracts translation of SQL rows to/from Java objects
  - Every SQL data-driven project uses it
  - Uses annotations
    - Specify table name (never matches the class name)
    - Specify column names when they don't match the field name
- Hibernate is a very popular implementation
- Often used with JTA (Java Transaction API)
  - You don't have to use JTA, this project doesn't
- Configured via META-INF/persistence.xml, which must be on the classpath somewhere

JPA queries

- Native queries
  - vendor specific query strings, like you'd use with JDBC
- JPQL (Java Persistence Query Language)
  - abstracted query strings
- HQL (Hibernate Query Language)
  - Hibernate specific variant of JPQL
- JPA Criteria API (introduced in JPA 2)
  - Method chaining (fluent API)
  - Essentially an OOP representation of a select query
- Most projects use these more or less at random
  - Many do not use JPA Criteria API at all
  - Approach used here is Native queries and JPA Criteria API only
    - Avoid JPA specific strings
    - Easier to figure out how to run the query manually in a GUI database tool

JPA in Maven POM

```
<!-- JPA (Hibernate) -->

<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.1-api</artifactId>
  <version>${jpa.api.version}</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.version}</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
</dependency>
```

JPA persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="hsqldb" transaction-type="RESOURCE_LOCAL">
      <description>HSQLDB Persistence Unit</description>
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <non-jta-data-source>jdbc/mem</non-jta-data-source>

      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
         <property name="hibernate.hbm2ddl.auto" value="none" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />
      </properties>
   </persistence-unit>
</persistence>
```

JPA

- Define entities
  - List them in the persistence.xml
  - Use annotations instead (approach used in this project)
- Can automatically create/update table definitions based on entities
  - In practice, projects manage such changes through custom SQL scripts
- EntityManager is entry point to making queries
  - Effectively wraps a database connection
  - Tracks entities, which can be attached or detached
    - Attached means the EntityManager is tracking it (EG, read from database via select), and will write out changes when transaction is committed
    - Detached means the EntityManager is not tracking it
      - Entities created via constructor
      - Entities specifically detached by calling EntityManager.detach
  - Two methods for writing changes back to database
    - persist: use it for entities that have never been explicitly detached
    - merge: use it to reattach entities that were explicitly detached and changed
  - Can also call flush to empty caches

JPA

- Common point of confusion is the usage of persist versus merge
  - Commonly, the logic in projects is obfuscated by layers of method calls in the controller, hard to follow
  - Approach used here is to abstract these details into the model, the caller doesn't even need to know JPA is being used
  - Never uses merge, because objects are never explicitly detached
  - Easier to understand

JPA

- Common point of confusion is the usage of flush to empty the cache
- Necessary when rows are updated in a way where JPA does not know which entity(s) are affected
  - Most commonly by using an HQL/JPQL/Native query to update only specific columns of a row
  - Useful in cases where you don't want to read in the entire row before making changes. EG, maybe you have a CLOB column that can be many MB in size, and all you want to do is update one or two columns
  - Detach the entity for the row being updated, flush the cache, perform the update query

JPA Native queries

```
// hsqldb is the value of the persistence-unit name in
// persistence.xml
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("hsqldb");

EntityManager em = emf.createEntityManager();

Query q = em.createNativeQuery("update ...");

// returns the number of rows affected
q.executeUpdate();
```

JPA Criteria Query API

```
// hsqldb is the value of the persistence-unit name in
// persistence.xml
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("hsqldb");

EntityManager em = emf.createEntityManager();

CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<AddressEntity> query =
  builder.createQuery(AddressEntity.class);
Root<AddressEntity> root = query.from(AddressEntity.class);
ParameterExpression<Integer> byId =
  builder.parameter(Integer.class);
query.select(root).where(builder.equal(root.get("id"), byId));

TypedQuery<AddressEntity> typed = em.createQuery(query);
// "id" is a value passed to this code by the caller
typed.setParameter(byId, new Integer(id));
List<AddressEntity> results = typed.getResultList();
```

JPA Caching

- JPA can use writeback caching, where rows are not actually written to the database until an explicit flush or commit is executed
  - Hibernate does this by default
- This means you should ensure that the cache is flushed before rendering your view
  - If you wait until after the view has rendered to flush the cache, then there is no opportunity to display any error to the user. The user thinks the operation succeeded, but the logs show an error.

JPA Lifecycle

- EntityManagerFactory = connection pool
- EntityManager = connection
- An EntityManagerFactory should be acquired as part of initialization during application startup (this is true for any kind of app, not just webapps)
- The web app request/response cycle should do the following:
  - Get a new EntityManager
  - Start a transaction
  - Perform controller actions
  - Flush the cache
  - Render the view
  - Commit or rollback

JPA Lifecycle in the example

- App initialization
  - ServletContextTransactionFilter dynamically adds a filter to each servlet defined in the app that comes before any other filter. This filter is the ServletTransactionFilter class.
- Request/Response cycle
  - ServletTransactionFilter
    - acquires a connection
    - starts transaction
    - executes filter chain (including servlet and rendering of view)
    - commit if no exception caught, else rollback
  - AddressServlet
    - gets transaction, performs operation (select or update)
    - flushes cache
    - renders view

Proper usage of MVC

- In MVC, the model is not just plain old beans, but all business logic
- The model should be able to work in a web app just as well as a desktop app
- Server-side validation should really only be done by the model
  - Only the model should know details like the length limit on a field
  - Multiple controllers can use the same model, and get consistent behaviour (EG, all controllers get the same validations applied)
- Practially speaking, the server side view may need validation
  - EG, A single view field for a phone number may be split into multiple model fields for country code, area code, phone, and extension. The view may need to validate the field before parsing it.
- Client side validation is useful to reduce wasted round trips for basic problems like missing required fields
  - Need to find a balance of not doing too much on the client
  - Focus on low hanging fruit that aid the user in entering data faster

Model considerations

- API that completely abstracts database
  - Perhaps some operations will use SQL, while others may use a document database like MongoDB, a graph database like Neo4J, etc.
  - Validation API that provides useful errors
  - Transactions
- Better yet, abstract all data outside the JVM
  - Files on disk, configuration parameters, remote service calls, etc
  - Localization
  - Can all be considered part of the model