Jfokus 2016
Testing Microservices

Katherine Stanley

Introduction

- Software Engineer
- WebSphere Application Server Liberty
- IBM

https://github.com/katheris

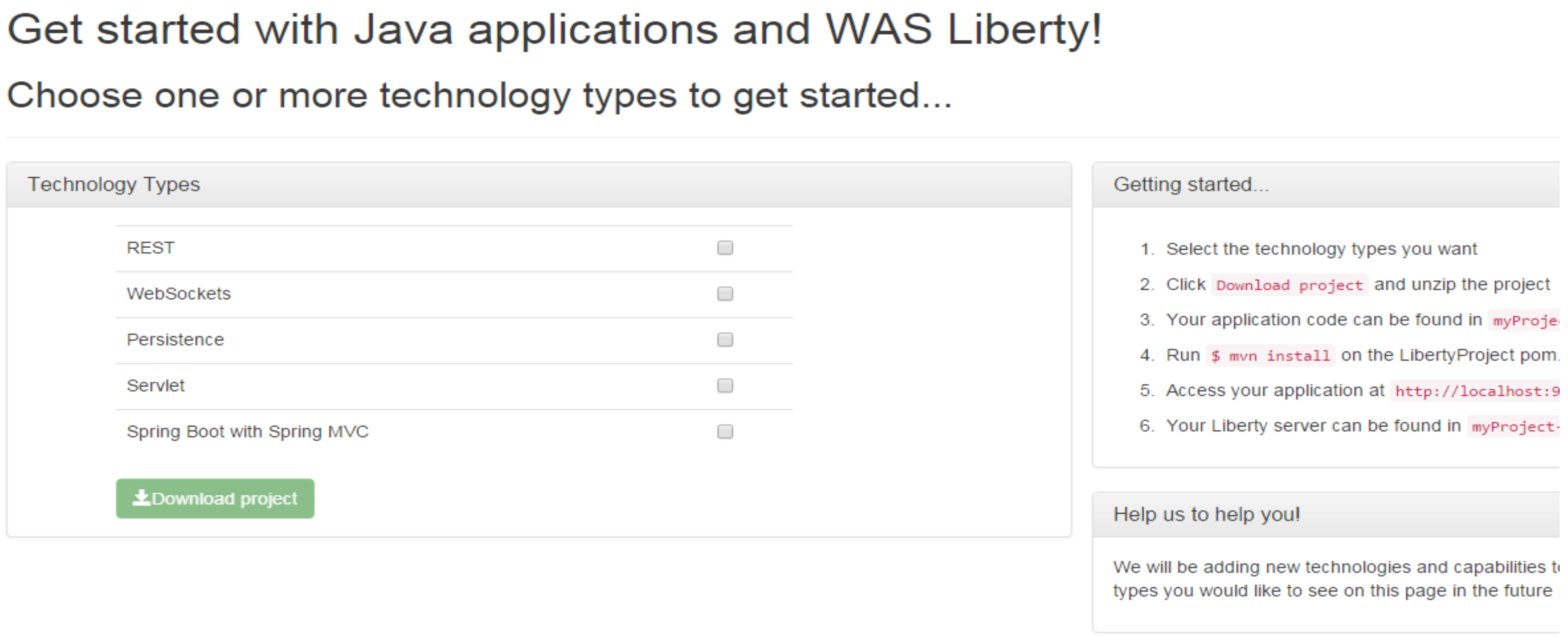
@KateStanley91

Agenda

- Background
- What are microservices?
- Example application architecture
- Testing strategies
- Evolving a monolithic application
- Conclusion

Background

- Liberty Starter – tool to help people get started



Get started with Java applications and WAS Liberty!

Choose one or more technology types to get started...

Technology Types

REST	☐
WebSockets	☐
Persistence	☐
Servlet	☐
Spring Boot with Spring MVC	☐

Download project

Getting started...

1. Select the technology types you want
2. Click `Download project` and unzip the project
3. Your application code can be found in `myProje`
4. Run `$ mvn install` on the LibertyProject pom.
5. Access your application at `http://localhost:9`
6. Your Liberty server can be found in `myProject-`

Help us to help you!

We will be adding new technologies and capabilities t
types you would like to see on this page in the future

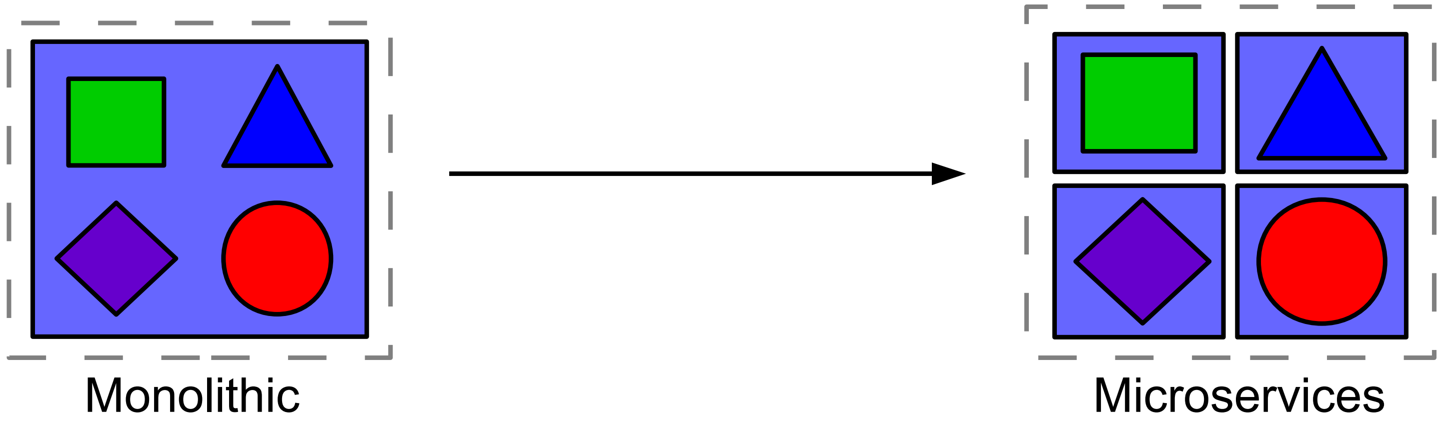
Out of scope for this presentation

- Performance testing
- Testing for scalability
- Security testing
- Monitoring

What are microservices?

What are microservices

- An application that is split down into different services:
- Each service has a different role
- The services communicate over language-agnostic protocol
  - e.g. REST
- The services are independently deployable



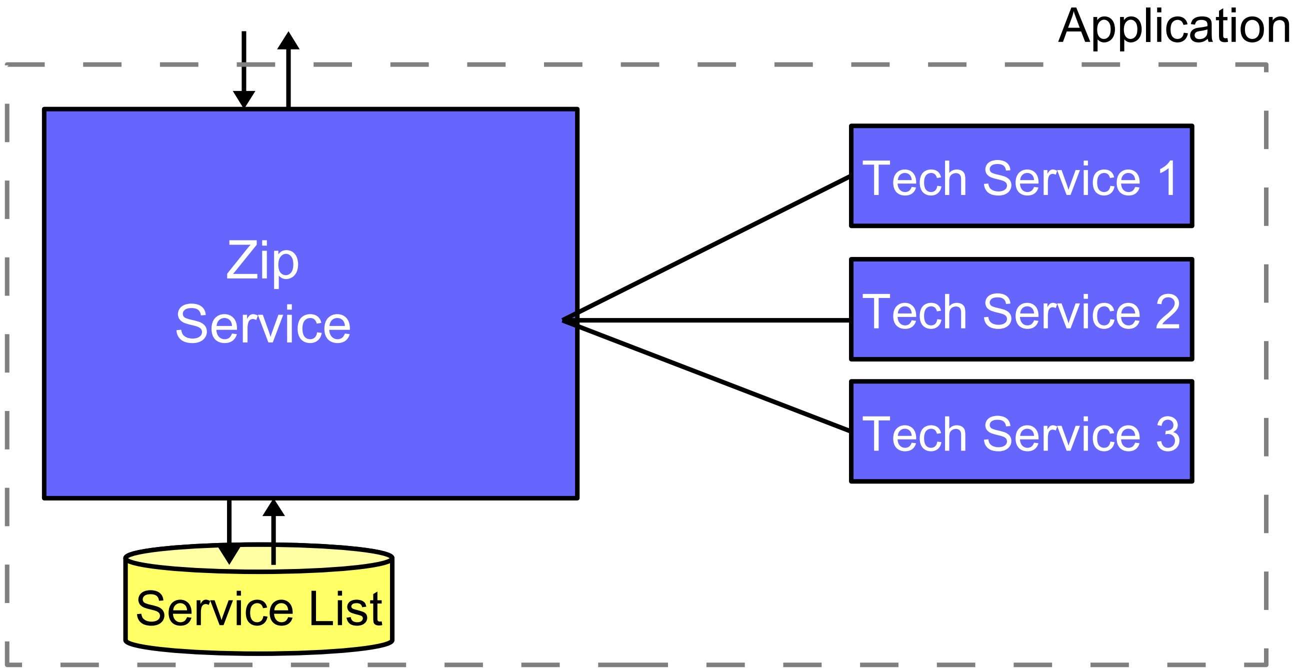
Why?

- To get rapid delivery
- To better use scaling resources
- To move pieces to the cloud – evolving a monolith
- Considerations:
  - Increased deployment overhead
  - Increased complexity
  - Testing strategy may need changing

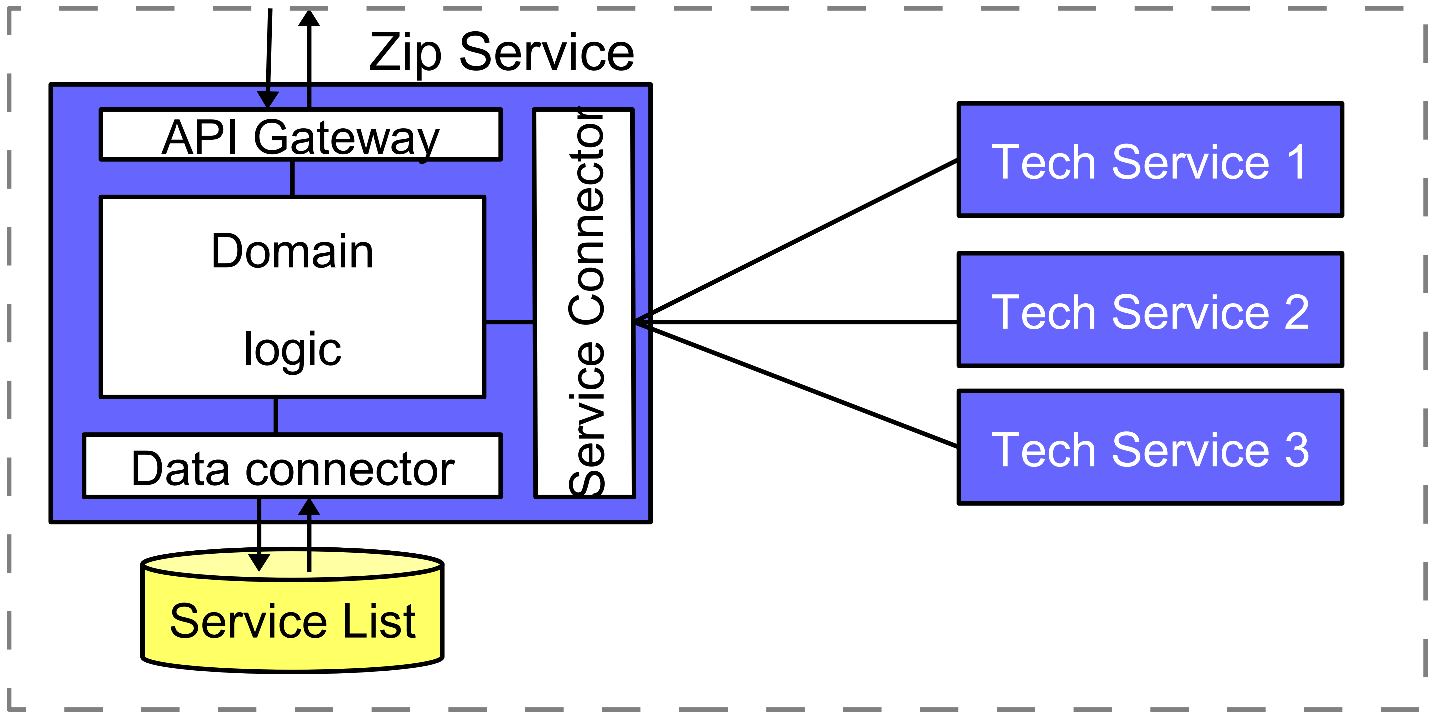
Our sample microservice

Sample microservice – anatomy

- Tool to provide starter code for java applications
- Split down into different technology services



Sample microservice – anatomy

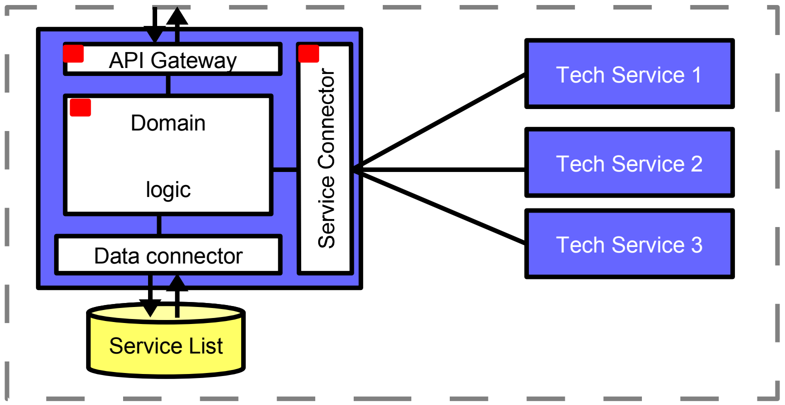


Testing strategies

Testing strategies

- Unit tests
- Component tests
- Contract tests
- Integration tests
- End-to-end tests

Testing strategies - unit



Unit testing

- Testing a small piece of behavior – usually class level
- Drives implementation when using test driven development
- Two types:
  - Black box
  - Test doubles (often called mocks)

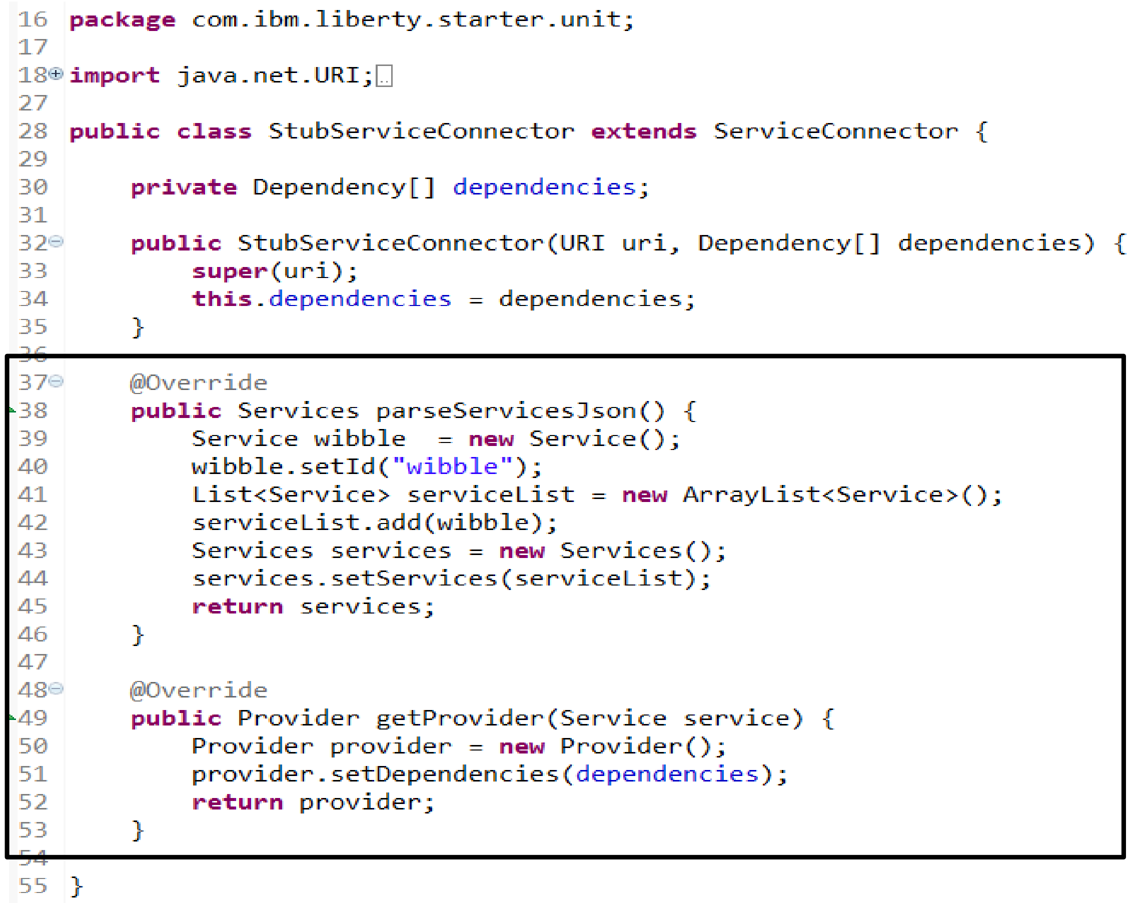
Unit testing – black box

- Use the actual objects and treat the unit as a black box
- For testing domain logic that is highly state-based
- e.g.
  - Test base project construction

Unit testing – test doubles

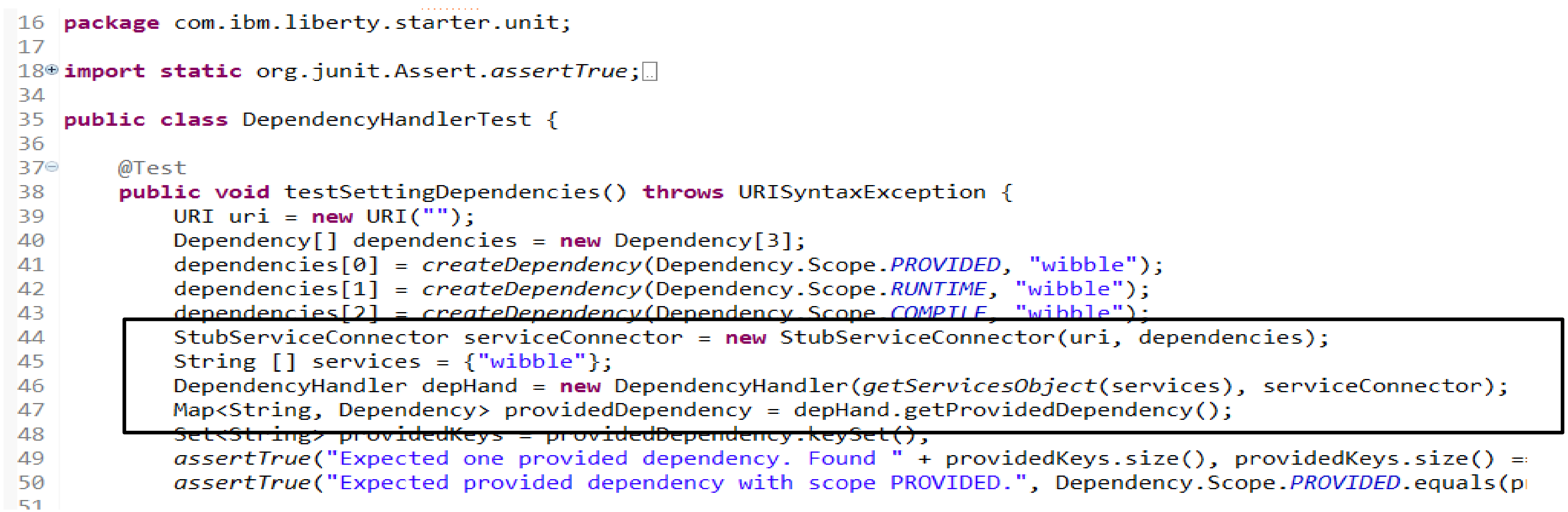
- Use test doubles to isolate the unit
- Useful for:
  - Routing layer
  - Gateway and repository testing

Unit testing – test doubles



```java
package com.ibm.liberty.starter.unit;

import java.net.URI;

public class StubServiceConnector extends ServiceConnector {

    private Dependency[] dependencies;

    public StubServiceConnector(URI uri, Dependency[] dependencies) {
        super(uri);
        this.dependencies = dependencies;
    }

    @Override
    public Services parseServicesJson() {
        Service wibble  = new Service();
        wibble.setId("wibble");
        List<Service> serviceList = new ArrayList<Service>();
        serviceList.add(wibble);
        Services services = new Services();
        services.setServices(serviceList);
        return services;
    }

    @Override
    public Provider getProvider(Service service) {
        Provider provider = new Provider();
        provider.setDependencies(dependencies);
        return provider;
    }

}
```

Unit testing – test doubles

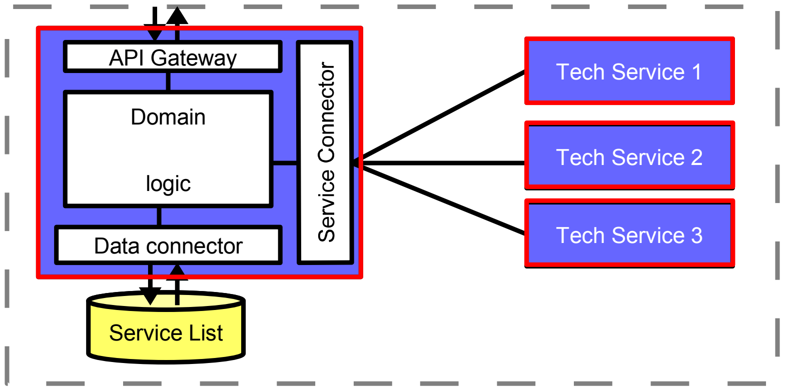


```java
package com.ibm.liberty.starter.unit;

import static org.junit.Assert.assertTrue;

public class DependencyHandlerTest {

    @Test
    public void testSettingDependencies() throws URISyntaxException {
        URI uri = new URI("");
        Dependency[] dependencies = new Dependency[3];
        dependencies[0] = createDependency(Dependency.Scope.PROVIDED, "wibble");
        dependencies[1] = createDependency(Dependency.Scope.RUNTIME, "wibble");
        dependencies[2] = createDependency(Dependency.Scope.COMPILE, "wibble");
        StubServiceConnector serviceConnector = new StubServiceConnector(uri, dependencies);
        String [] services = {"wibble"};
        DependencyHandler depHand = new DependencyHandler(getServicesObject(services), serviceConnector);
        Map<String, Dependency> providedDependency = depHand.getProvidedDependency();
        Set<String> providedKeys = providedDependency.keySet();
        assertTrue("Expected one provided dependency. Found " + providedKeys.size(), providedKeys.size() =
        assertTrue("Expected provided dependency with scope PROVIDED.", Dependency.Scope.PROVIDED.equals(p
```

Unit testing

- Smaller services → more plumbing and coordination
- Unit tests can constrain implementation
- Trade off between time to maintain and usefulness
- Keep test suite small and focused

Testing strategies - component



Component testing

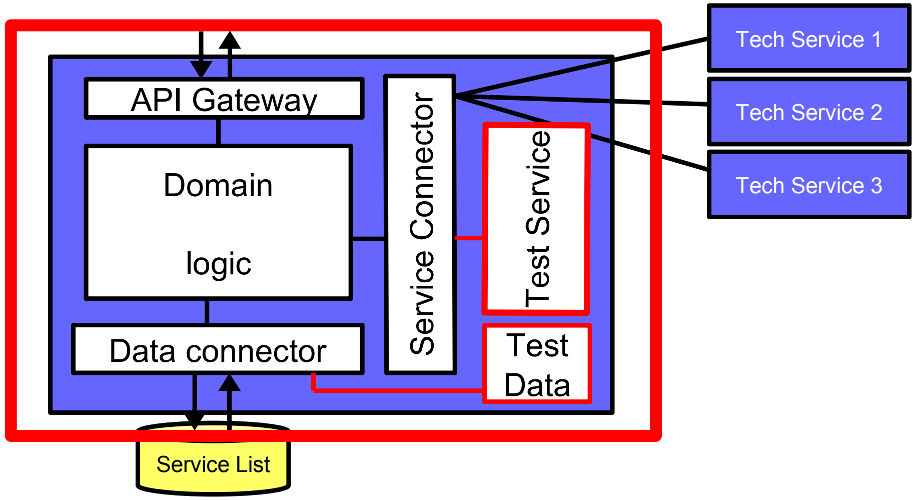
- Tests a portion of the system
- Component should be a replaceable piece
- In microservices – components are services
- Two options:
  - Alter the internals
  - Create a 'test service'

Component testing – altering the internals

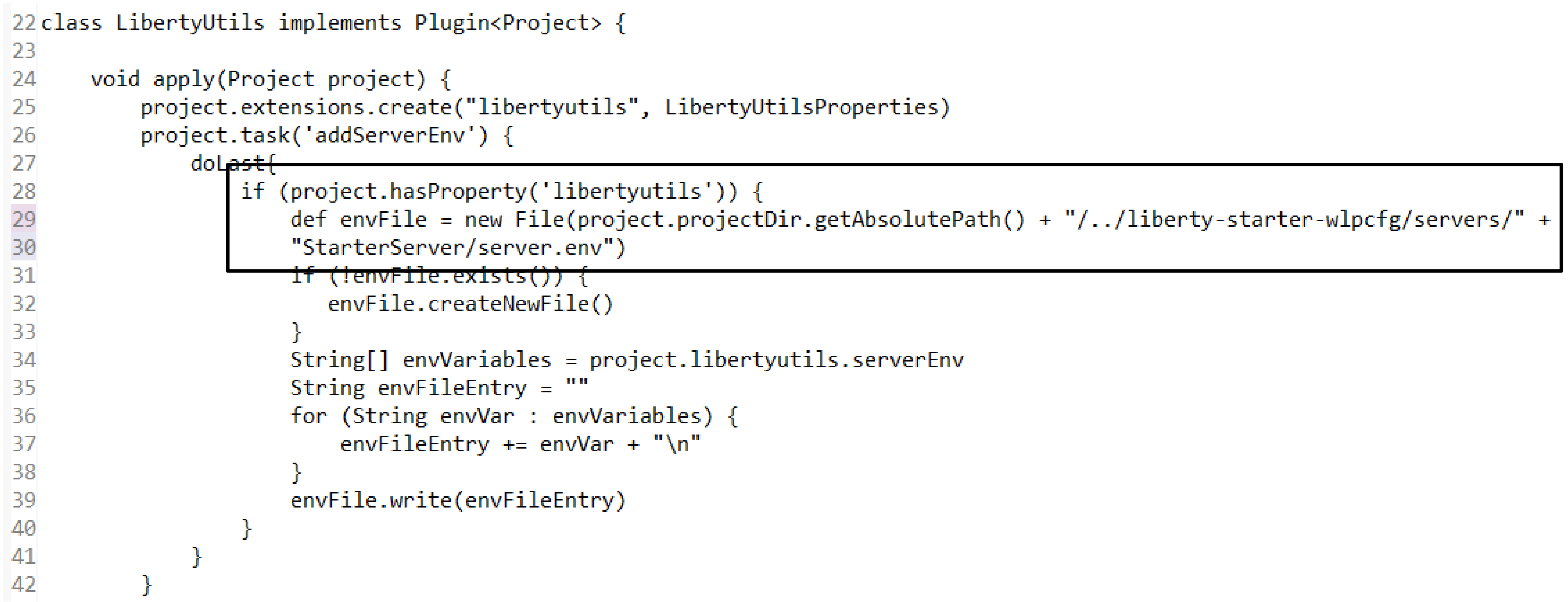
- Use internal interfaces to aid with testing
  - Does not touch the network
  - Fast execution, fewer moving parts
- Things to consider:
  - A less 'clean' system
  - Network specific problems could get missed

Component testing – test services

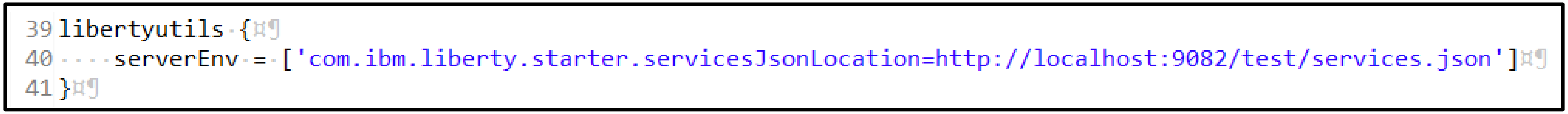
- Create a test service – on the same server, or elsewhere
  - Complexity is contained within the test microservice
  - Thoroughly tests network calls
- Things to consider:
  - Increased execution time



Component testing – test services

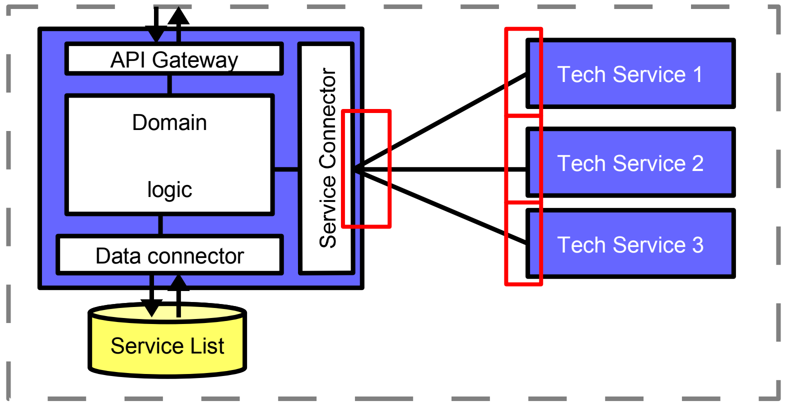


```groovy
class LibertyUtils implements Plugin<Project> {

    void apply(Project project) {
        project.extensions.create("libertyutils", LibertyUtilsProperties)
        project.task('addServerEnv') {
            doLast{
                if (project.hasProperty('libertyutils')) {
                    def envFile = new File(project.projectDir.getAbsolutePath() + "/../liberty-starter-wlpcfg/servers/" +
                    "StarterServer/server.env")
                    if (!envFile.exists()) {
                       envFile.createNewFile()
                    }
                    String[] envVariables = project.libertyutils.serverEnv
                    String envFileEntry = ""
                    for (String envVar : envVariables) {
                        envFileEntry += envVar + "\n"
                    }
                    envFile.write(envFileEntry)
                }
            }
        }
```



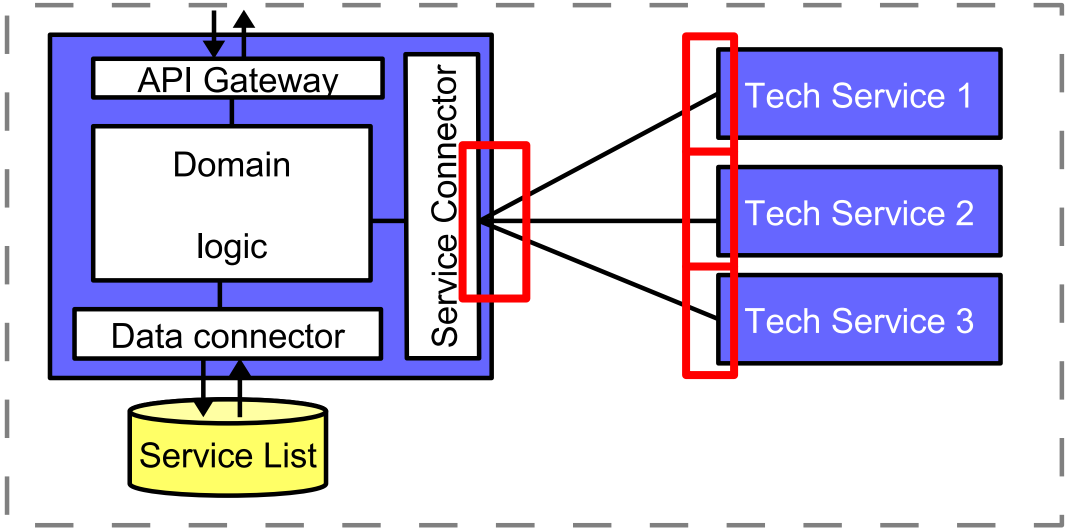
```groovy
libertyutils {
    serverEnv = ['com.ibm.liberty.starter.servicesJsonLocation=http://localhost:9082/test/services.json']
}
```

Testing strategies - contract



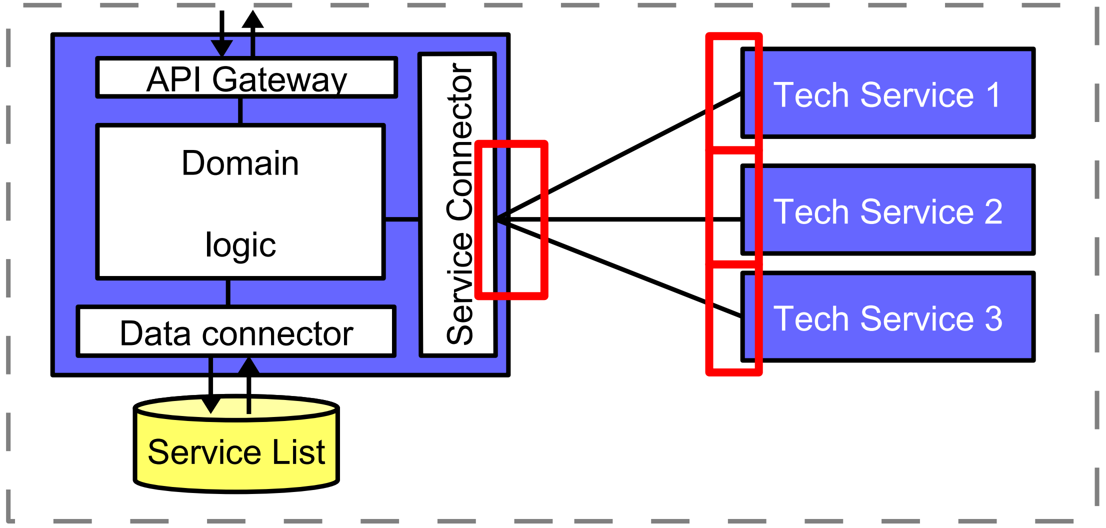
Contract testing

- Contract – an agreed set of input and output attributes
- Tests the inputs and outputs have required attributes
- Tests the boundary between your application and external services

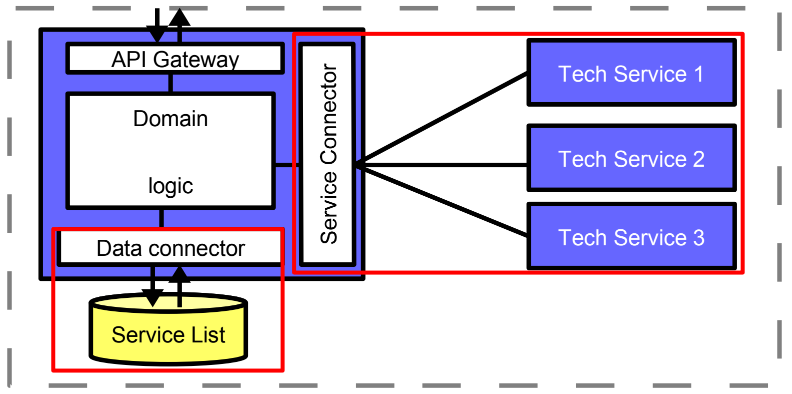


Contract testing – organization management

- Contract tests passed to the maintainers
- Sum of contract tests = service contract
- Service contract can be used to manage changes



Testing strategies - integration

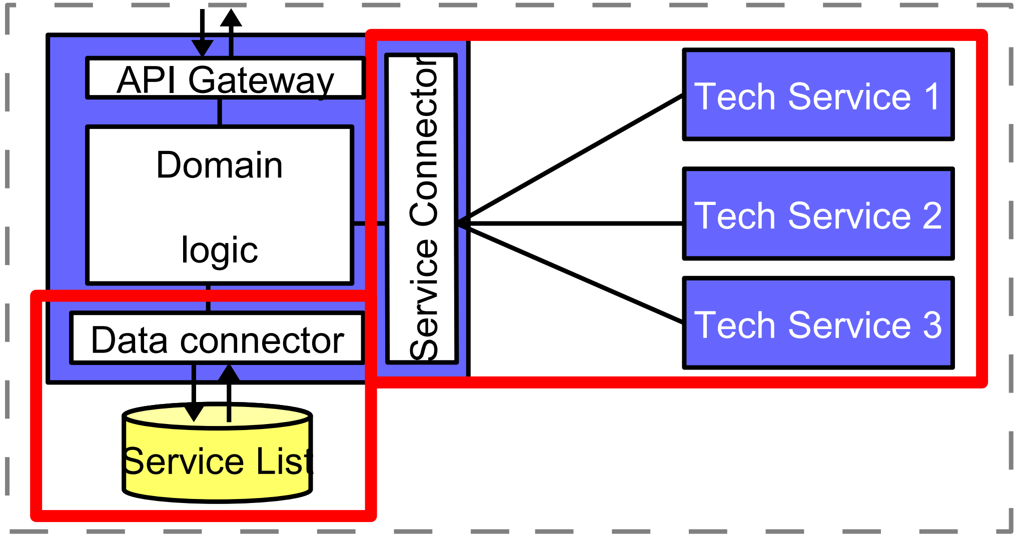


Integration testing

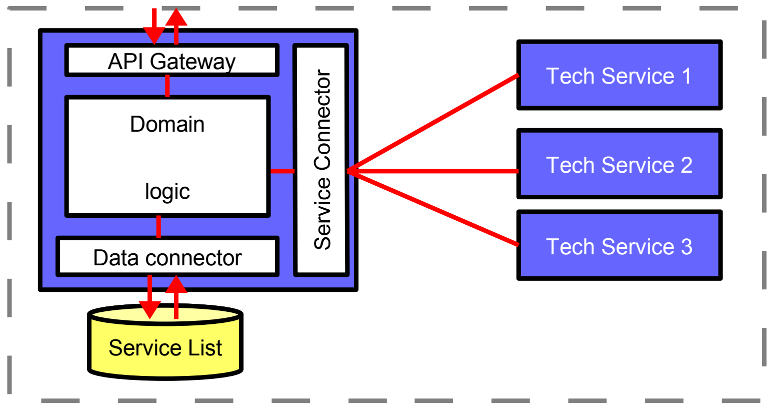
- Tests the interactions between components
- Test for basic success and error paths
- In microservice environment tests interaction with:
  - Other services
  - Data stores

Integration testing – things to consider

- Connection failures could cause false errors
- Use unit and contract testing for behavior
- Consider moving these tests to the build pipeline



Testing strategies - end-to-end

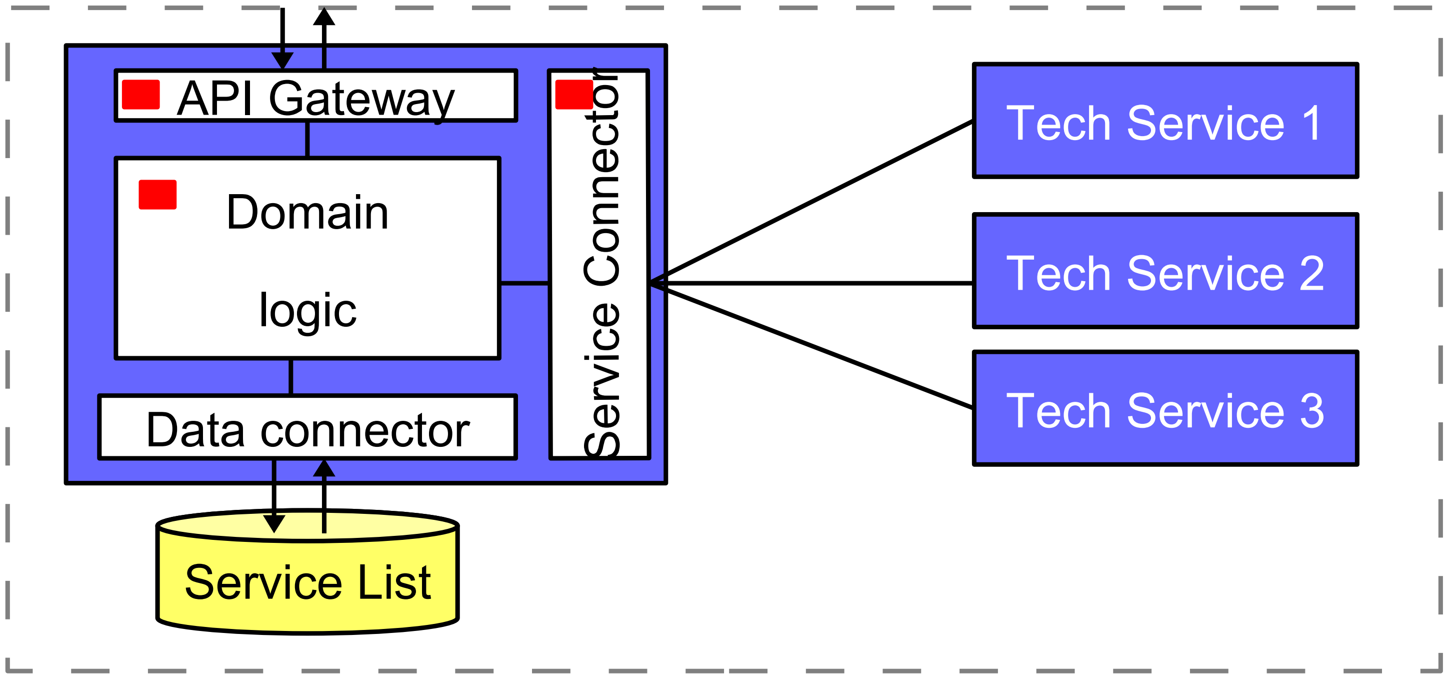


End-to-End

- Verifies system meets external requirements
- Sanity check
- Important in stateful systems
- Include UI testing

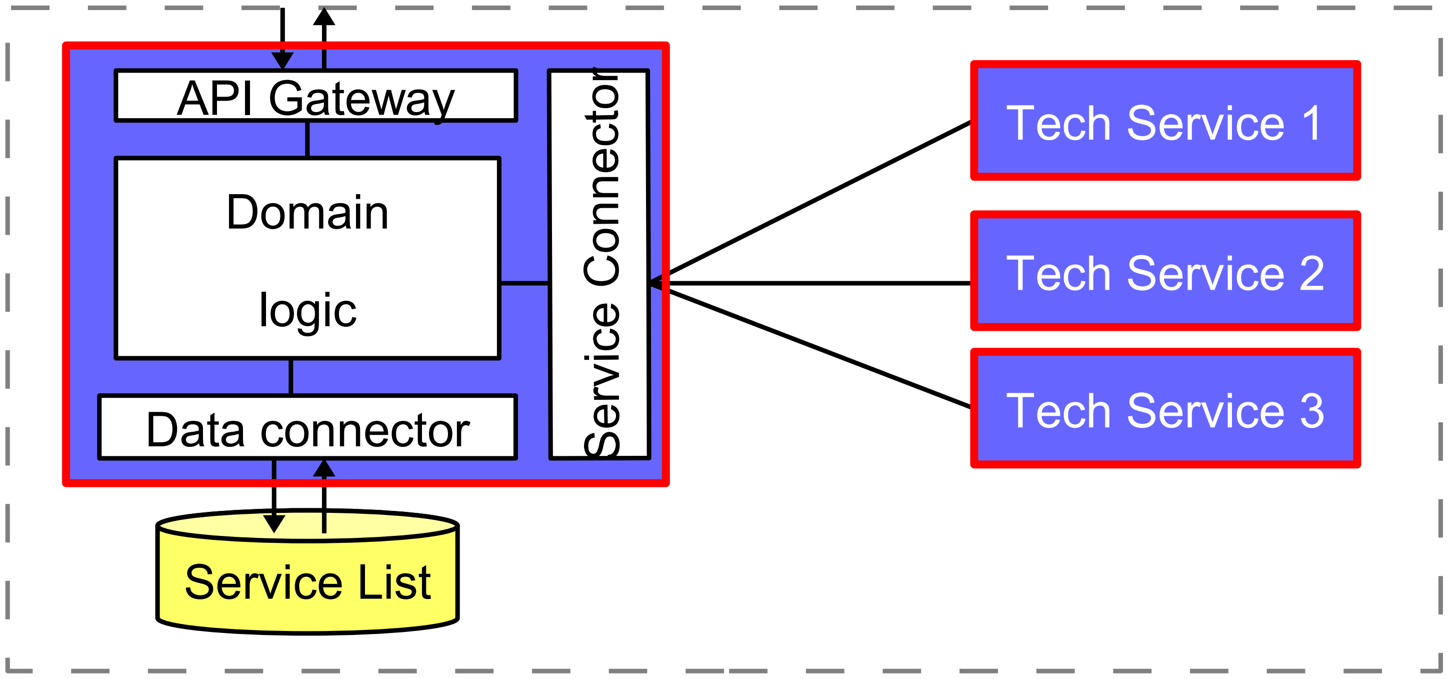
Testing Strategies

- Unit



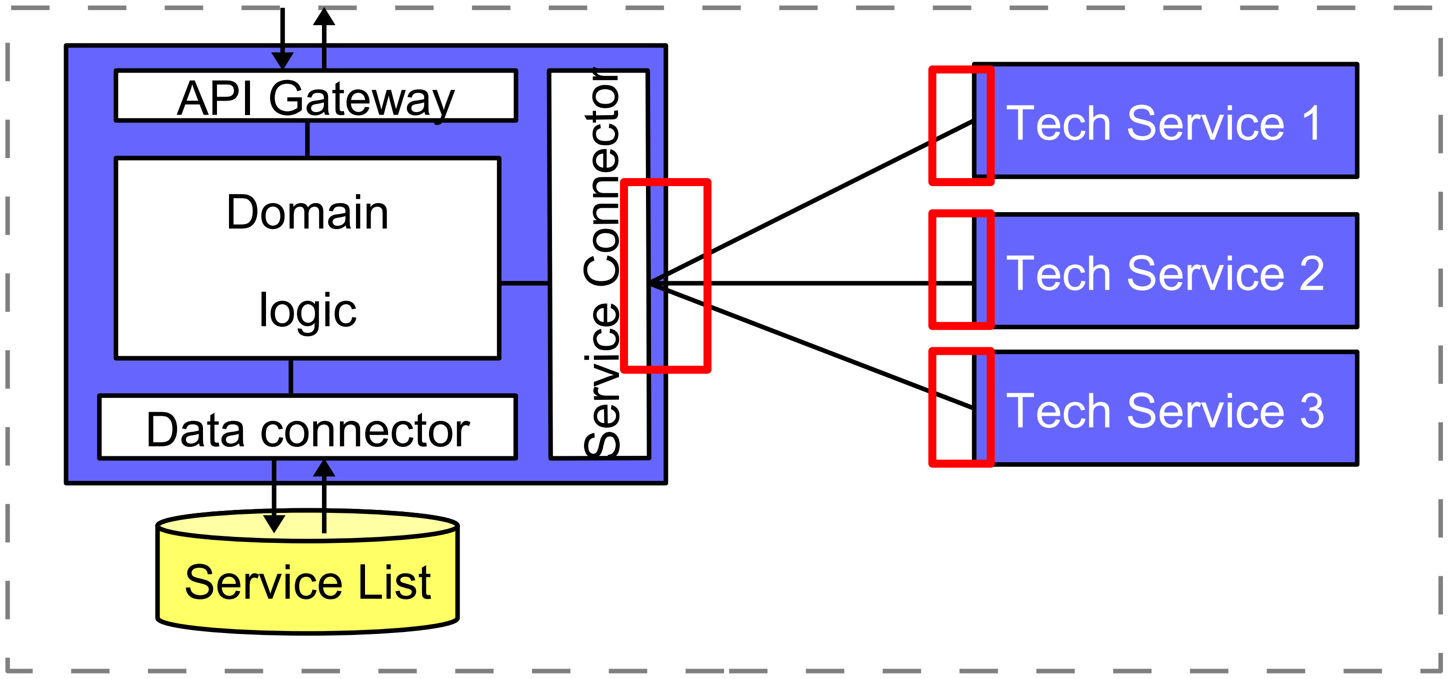
Testing Strategies

- Unit
- Component



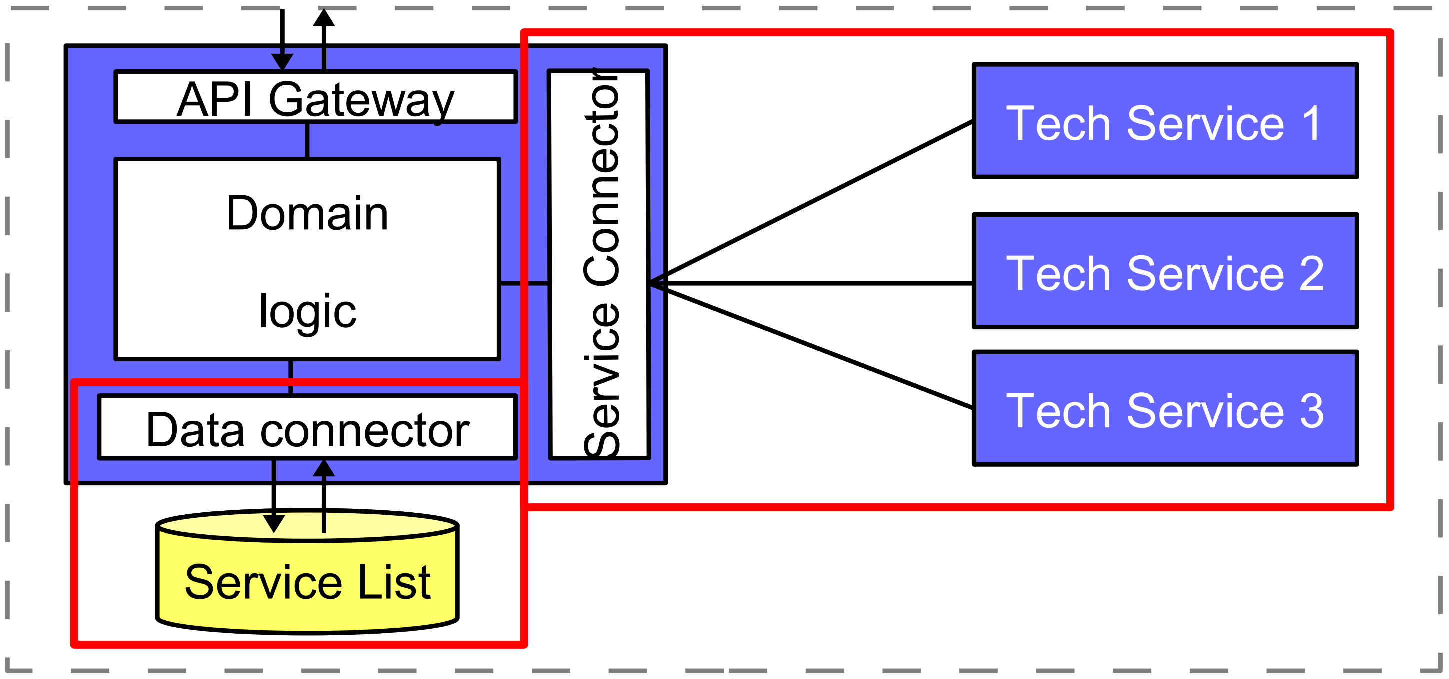
Testing Strategies

- Unit
- Component
- Contract



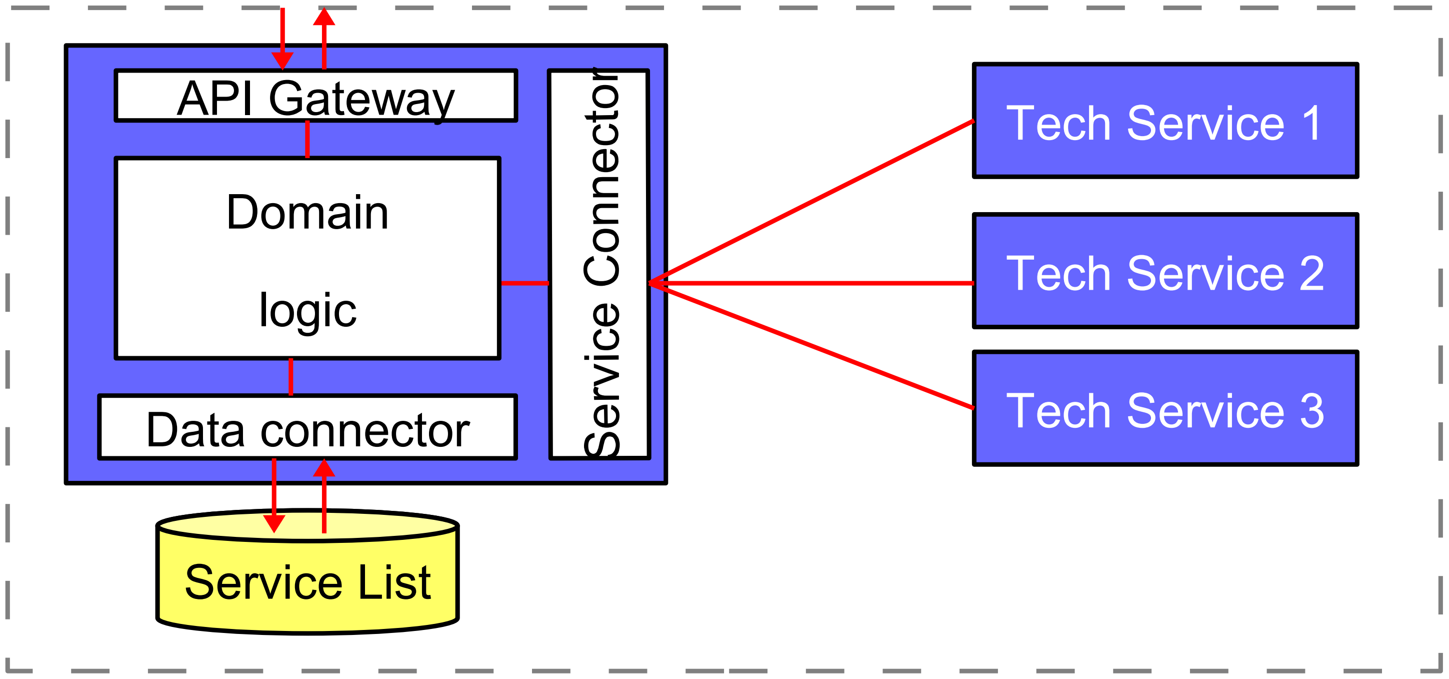
Testing Strategies

- Unit
- Component
- Contract
- Integration



Testing Strategies

- Unit
- Component
- Contract
- Integration
- End-to-End



Monolith → Microservice

Monolith → Microservice

- Add tests for current function – fix flaws later
- The monolith will change, so will your tests
- Useful tools:
  - Minimal set of externally calling classes
  - Mocking for unit test
  - Use a 'test' service for component tests

Conclusion

Conclusion

- Still a developing area
- For evolution take small steps
- Dummy service useful tool
- Adapt tests to suit the development and deployment process

Questions?

http://martinfowler.com/articles/microservice-testing/

wasdev.net → Docs → Microservices

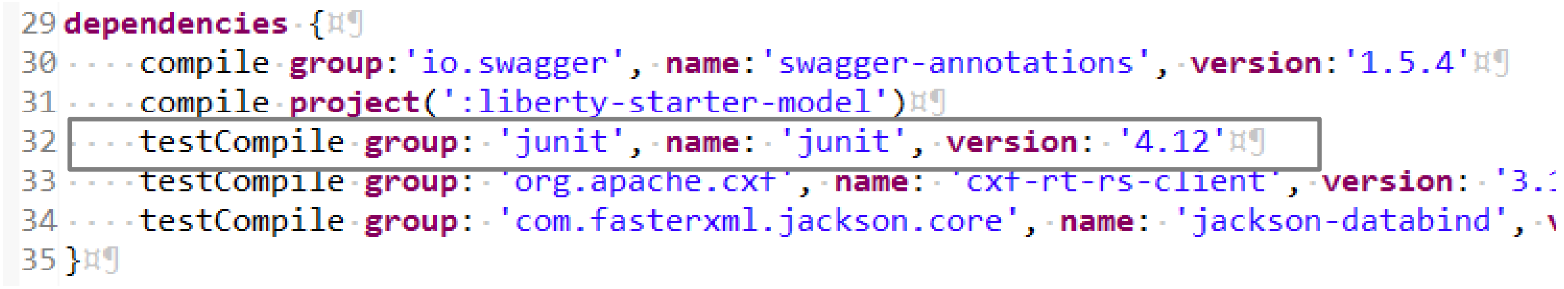
ibm.biz/LibertyStarter

@KateStanley91

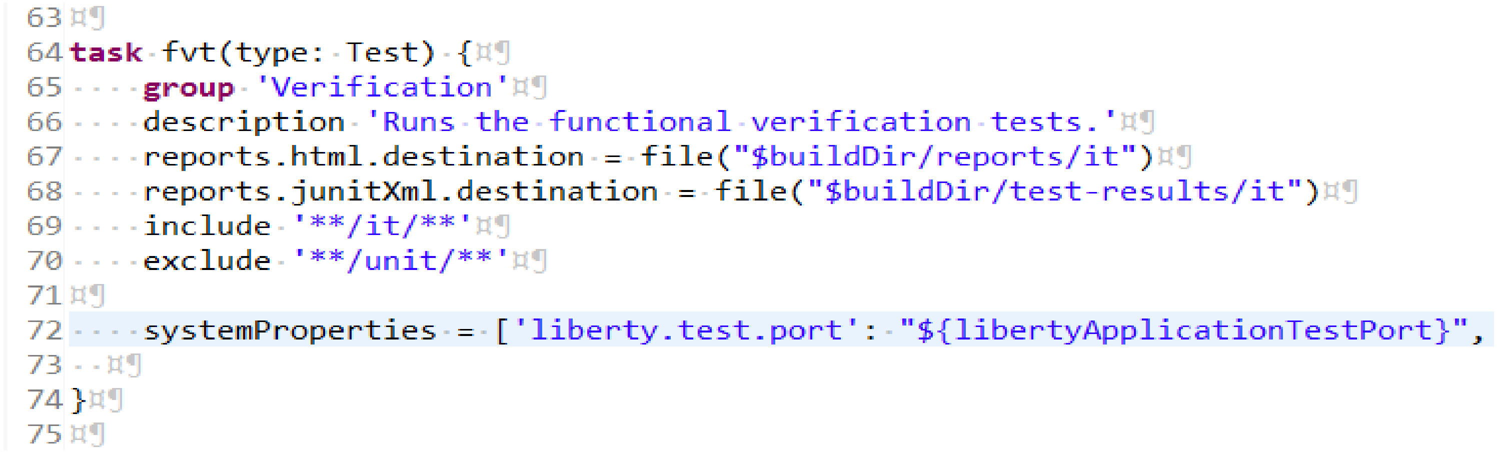
Testing in maven and gradle

- Maven
    - Structured
    - Transferable skill
- Gradle
    - More freedom
    - Risk of creating a very complicated build system

Testing in gradle



```
dependencies {
    compile group:'io.swagger', name:'swagger-annotations', version:'1.5.4'
    compile project(':liberty-starter-model')
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testCompile group: 'org.apache.cxf', name: 'cxf-rt-rs-client', version: '3.
    testCompile group: 'com.fasterxml.jackson.core', name: 'jackson-databind', 
}
```



```

task fvt(type: Test) {
    group 'Verification'
    description 'Runs the functional verification tests.'
    reports.html.destination = file("$buildDir/reports/it")
    reports.junitXml.destination = file("$buildDir/test-results/it")
    include '**/it/**'
    exclude '**/unit/**'

    systemProperties = ['liberty.test.port': "${libertyApplicationTestPort}",
  
}

```