FOCUS: MICROSERVICES

Migrating Enterprise Legacy Source Code to Microservices

On Multitenancy, Statefulness, and Data Consistency

Andrei Furda and Colin Fidge, Queensland University of Technology

Olaf Zimmermann, University of Applied Sciences of Eastern Switzerland, Rapperswil

Wayne Kelly and Alistair Barros, Queensland University of Technology

* *Microservice migration is a promising technique to incrementally modernize monolithic legacy enterprise applications and enable them to exploit the benefits of cloud-computing environments. This article elaborates on three challenges of microservice migration: multitenancy, statefulness, and data consistency.* ***//***



**MODERNIZATION OF LEGACY** en-terprise systems is a challenge faced by many organizations that want to exploit the new cloud-computing technologies to meet high-scalability and high-availability needs. Assum-ing that the legacy system’s source code is available and maintainable, microservices are a promising solu-tion in which centralized services are reimplemented as multiple in-dependent services1,2 (see Figure 1). Microservices support incremental modernization, leading to highly scalable systems3 with high avail-ability through redundancy of ser-vice instances and reduced costs. Microservices also facilitate the low-risk, small-scale incremental mod-ernization that is often preferred to large-scale approaches.4

In this article we explain three closely related challenges of mi-croservice migration: multitenancy, statefulness, and data consistency. (For a synopsis of these three chal-lenges, see the sidebar.)

**Multitenancy**

A multitenant software-as-a-service application fulfils the needs of mul-tiple groups of users, organizations, or departments. The application-level multitenancy model allows applica-tion instances to be shared by mul-tiple tenants. These instances can be configured to meet the tenants’ re-quirements.5,6 While the application instances are shared, the tenant data must be separated and must be accessi-ble only by the tenant that owns it7 (see Figure 2). In addition to the separation of tenant data, a multitenant environ-ment should ensure that computing resources are equally distributed be-tween the tenants. Performance sepa-ration ensures that increased demand for resources by one tenant does not negatively affect other tenants.8,9

074 0 - 74 5 9 / 18 / $ 3 3 . 0 0 © 2 0 18 I E E E MAY/JUNE 2018 | **IEEE SOFTWARE** **63**

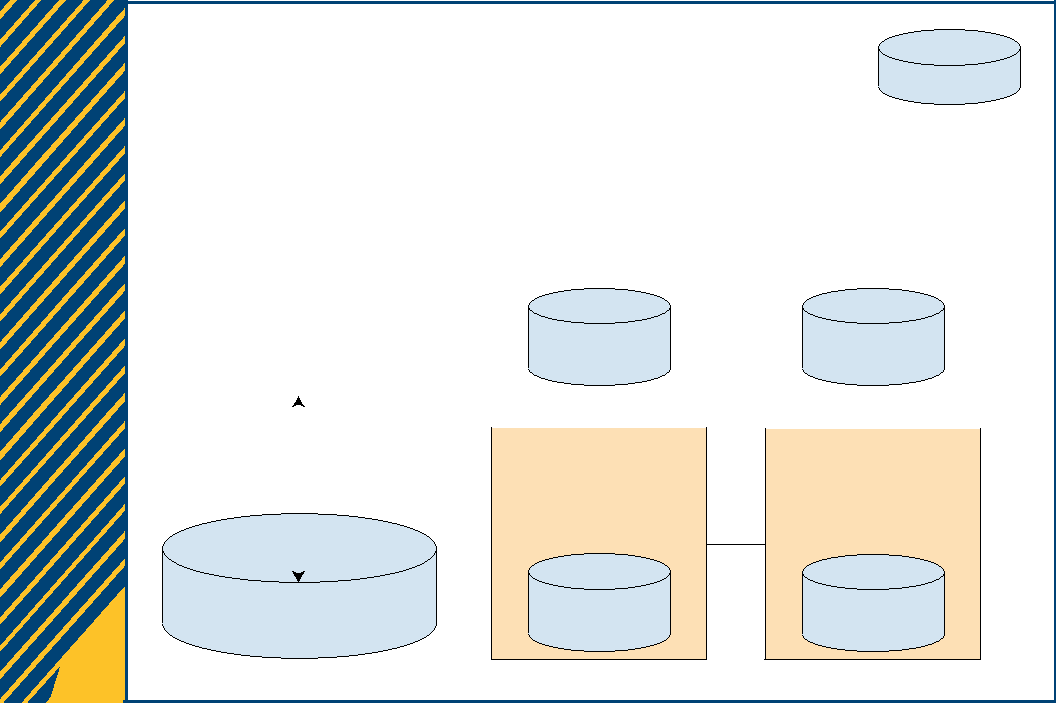
FOCUS: MICROSERVICES



**CHALLENGES OF MICROSERVICE MIGRATION**

*Multitenancy* is a system’s ability to fulfil the requirements ofmultiple groups of service consumers, organizations, and even competitors in an industry. It enables organizations that demand complete autonomy in the administration of their users and associated data to share access to the same physical instances of the system and to the application instances, while keeping their data strictly separated and ruling out any super-user having access to and control over it. Multitenant applications must be highly configurable with tenant-specific settings. Legacy enterprise applications have often been designed for single orga-nizations and operate in single-tenant mode.

*Statefulness* in the context of microservices is the abilityto retain state information that was generated previously. The response of a stateful microservice may depend not only on



the most recent service request but also on the retained state from previous interactions. Ideal microservices are stateless, but monolithic legacy code is often stateful.

*Data consistency* is enforced by a system’s ability toaffect data stored in a shared repository only in allowed ways. Data consistency defects can be introduced by mis-take when migrating sequential legacy code that accesses a centralized data repository to microservices. Microservices are designed to be scaled out, allowing multiple microser-vice instances to operate in parallel. To improve performance and scalability, microservices rely on decentralized data repositories, which leads to data consistency challenges when synchronizing the data. Concurrent database access is rarely supported in sequential legacy code.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | Microservice | | | | |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | application | | | | |  |  | Database | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Legacy application | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | Microservice | | | |  |  |  | Microservice | | | |  |  |
|  | User interface layer | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Processing layer | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | Local | | | |  |  |  |  | Local | | |  |  |
|  | Data access layer | |  |  |  | database | | | |  |  |  |  | database | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  | |  | |  |  |  |  | |  |  |  |  |
|  |  |  |  |  |  | Microservice | | | | |  | | Microservice | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Database | Local | Local |  |
|  | database | database |  |
|  |  |  |
| (a) |  | (b) |  |  |

FIGURE 1. Comparing (a) a legacy (e.g, monolithic) architecture and (b) a microservice architecture.

Single-Tenancy and Multitenancy They were not developed for mod-

Challenges in Legacy Source Code ern multitenant cloud environments,

Legacy enterprise applications of- and they usually do not support

ten operate in single-tenant mode. multitenancy­. Single-tenant legacy

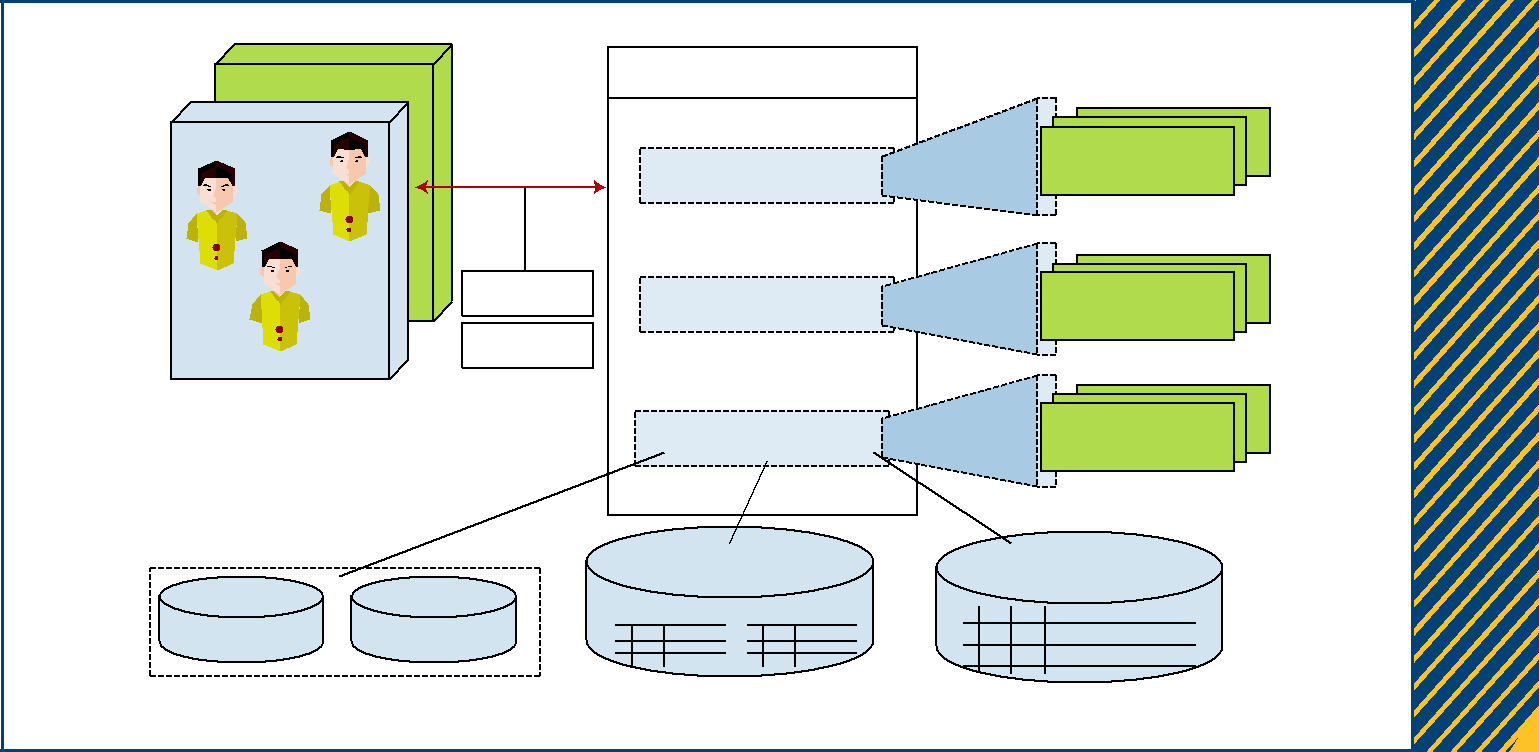
applications access the data reposi-tory of a single organization.

It is essential that the migration of single-tenant legacy code to mi-croservices also considers multi­ tenancy. Multitenant microservice instances can be shared by multiple tenants, configured to meet individ-ual requirements, while strictly sepa-rating the tenants’ data.

To ensure tenant data isolation, the source code needs to ensure that the data associated with tenant-aware input channels is tagged with the cor-rect tenant ID, and that this tenant tag is maintained unmodified until the data reaches a tenant-aware out-put channel. At the source code level, tenant-aware business objects need to be tagged with the correct tenant context as follows (see Figure 3).

Procedures that process tenant-aware input data (e.g., service re-quests, user interface inputs, and database read operations) need to retrieve the tenant ID from the run-time environment (env) and assign it

**64** **IEEE SOFTWARE** |W W W.C O M P U T E R .O R G / S O F T WA R E|@ I E E E S O F T WA R E



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tenant 2 |  | Multitenant application |  |  |
| Tenant 1 |  |  | Tenant-isolated |  |
|  | Request |  |  |
|  | User interface layer | component |  |
| User | User |  |  |  |
| Tenant ID | Processing layer | Tenant-isolated |  |
|  |  | component |  |
|  | Configuration |  |  |
| User |  |  |  |
|  |  |  |  |
|  |  | Data access layer | Tenant-isolated |  |
|  |  | component |  |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T1 | T2 | T1TableName T2TableName | Tenant ID |  |
|  | T1 |  |
|  |  |  | T2 |  |
|  | Database isolation | Table-based isolation | Row-based isolation |  |

FIGURE 2. A multitenant enterprise software-as-a-service architecture.6,7

to the input channel. For example, if the multitenant microservice is hosted by the Google App Engine, the tenant ID is retrieved from the Google Apps domain.10

Similarly, procedures that process tenant-aware data outputs (e.g., ser-vice responses, user interface outputs, and database write operations) retrieve the tenant ID and assign it to the out-put channel. For example, Windows Azure requires setting the tenant con-text before calling the storage API.10

In summary, the multitenant source code first retrieves the tenant ID (tenant context) before receiv-ing data from input channels, sets the tenant context for all required tenant-aware business objects, and sets the tenant context for all data that is sent to output channels.

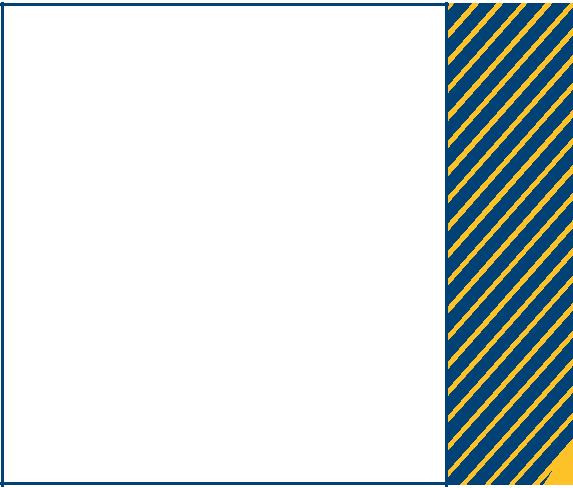
Pattern-Based Microservice Migration of Single-Tenant Legacy Source Code

Multitenancy challenges can be de-composed into subproblems that

can be solved using a combination of existing architectural patterns. In previous work, we have shown how to enable multitenancy by applying architectural patterns for enterprise applications and architectural refac-toring.6 We focused on enabling multitenancy in components; how-ever, the techniques and architec-tural patterns can also be applied to multitenant microservices.

The microservice’s data access component for a multitenant data-base can be implemented using the Two-Level Data Mapping Gateway pattern,6 a combination of the Data Mapper pattern and the Table Data Gateway pattern11 (see Figure 4). The first (Data Mapper) stage maps domain objects to the database gate-way and implements basic CRUD (create, read, update, delete) opera-tions, while the second (Gateway) stage loosely couples the access to a multitenant data repository and en-sures the separation of tenant data.

processInputChannel ( TenantAwareBO input ) { //1. retrieve the tenant context



tenantID = env.getTenantID();

//2. set the tenant context

input.setTenantID(tenantID);

. . .

}

processOutputChannel ( TenantAwareBO output ) {

. . .

//1. retrieve the tenant context

tenantID = env.getTenantID();

//2. set the tenant context

output.setTenantID(tenantID);

sendResponse(output);

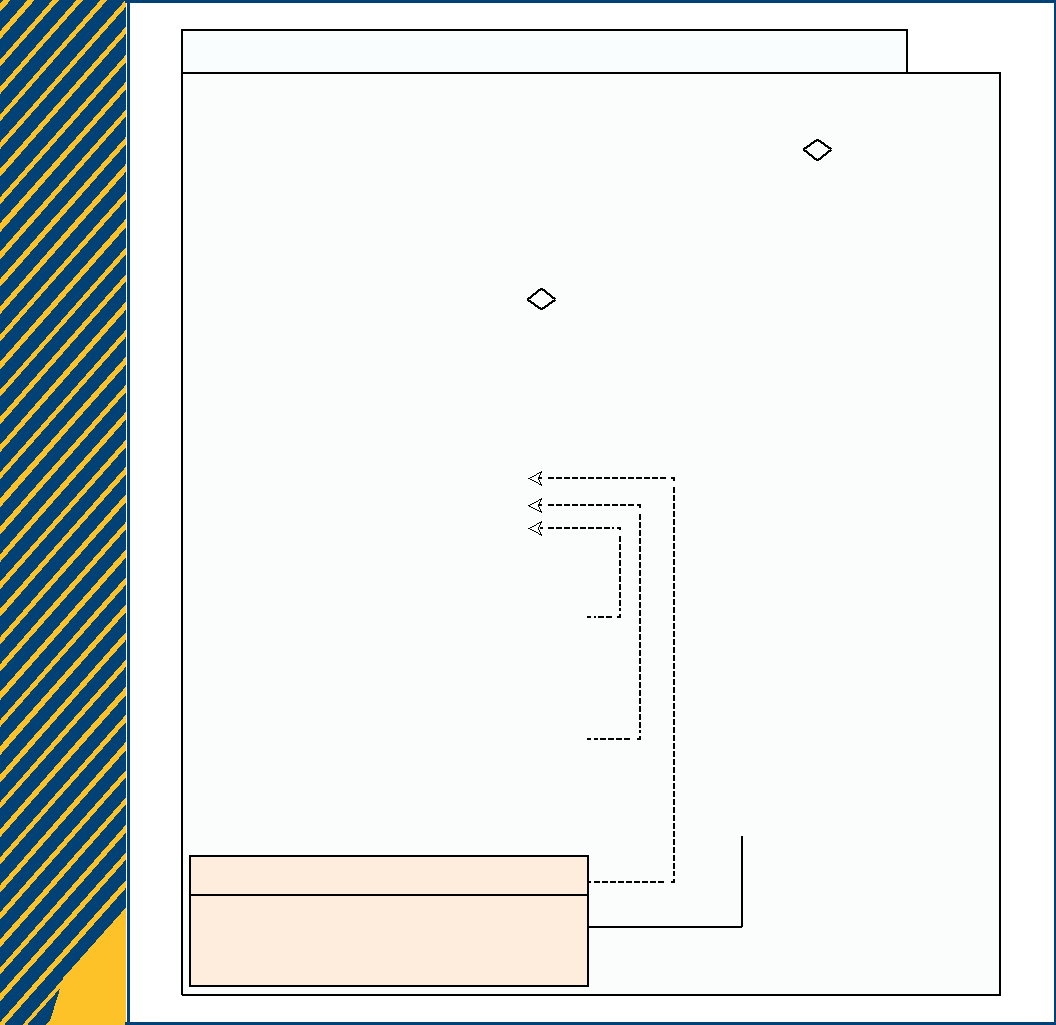
}

FIGURE 3. Multitenancy through tenant-aware I/O channels.

The operational-logic component of a multitenant microservice can be implemented using the Strategy pat-tern. This pattern lets you modify or extend tenant-specific logic imple-mentations without affecting other parts of the implementation6 (see Figure 5).

MAY/JUNE 2018 | **IEEE SOFTWARE** **65**

FOCUS: MICROSERVICES



Multitenant Data Access

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| IdentityMap |  |  |  | User |  |  |  |  |  |  | Tenant |  |
| authorizedTenantID |  |  |  | userName |  |  |  |  |  | tenantID | |  |
| authorizeTenant() |  |  |  | password : String | | | |  |  |  |  |  |
|  |  |  | tenantID |  |  |  |  |  |  |  |  |
| getDomainObject() |  |  |  |  |  |  |  |  |  |  |  |
| saveDomainObject(DomainObject) |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | DomainObject | | | |  |  |  |  |  |
| updateDomainObject(DomainObject) |  |  |  |  |  | |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | ID |  |  |  |  |  |  |  |  |
| fetchAllDomainObjects() |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | tenantID |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| <<interface>> |  |  |  |  |  |  |  |  |  |  |  |  |
| DomainObject\_TableDataGateway |  |  |  |  |  |  | \* |  |  |  |  |  |
| fetchAll(tenantID) |  |  |  |  |  |  |  |  |  |  |  |  |
| getDomainObject() |  |  |  |  |  |  |  |  |  |  |  |  |
| saveDomainObject(DomainObject) |  |  |  |  |  |  |  |  |  |  |  |  |
| deleteDomainObject(DomainObject) |  |  |  |  |  |  |  |  |  |  |  |  |
| updateDomainObject(DomainObject) |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| DomainObjectGateway\_DBTenantInstance |  |  |  |  |  |  |  |  |  |  |  |  |
| DBConnection |  |  |  |  |  |  | 1 |  |  |  |  |  |
| TableName : String |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | DomainObject\_DataMapper | | | | | | |  |
| DomainObjectGateway\_DBSharedSchema |  |  |  |  | readDomainObject(id) | | | | | | |  |
| DBConnection |  |  |  |  | updateDomainObject(id) | | | | | | |  |
|  |  |  |  |  |
| TableName : String |  |  |  |  | deleteDomainObject(id) | | | | | | |  |
| TenantID |  |  |  |  | createDomainObject(id) | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

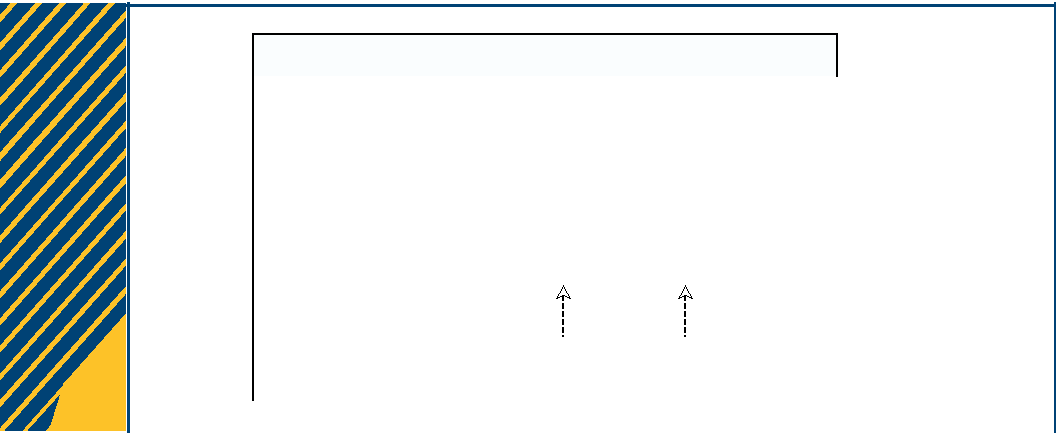
DomainObjectGateway\_DBSharedInstance

DBConnection

TableName : String

TenantID

FIGURE 4. A multitenant MVC (model–view–controller) model.



Multitenant Controller (with Strategy Design Pattern)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  | DomainObjController |  |  |  | <<interface>> |  |  |  |
|  |  |  |  |  | IDomainObjControllerStrategy |  |  |  |
|  | DomainObjControllerStrategy |  |  |  |  |  |
|  | createAction() |  | createActionStrategy(DomainObjController) | | |  |  |  |
|  |  | readActionStrategy(DomainObjController) | | |  |  |  |
|  | readAction() |  |  |  |  |
|  |  | updateActionStrategy(DomainObjController) | | |  |  |  |
|  | updateAction() |  |  |  |  |
|  |  | deleteActionStrategy(DomainObjController) | | |  |  |  |
|  | deleteAction() |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  | <<realize>> |  |  |  |
|  | <<realize>> | |  |  |  |  |
|  | Tenant1DomainObjControllerStrategy | |  |  | Tenant2DomainObjControllerStrategy |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  | | | | |  |  |  |  |
| FIGURE 5. A multitenant MVC controller. | | | | |  |  |  |  |
| The user interface component of a | | | | | decoupling of user interface data | | |  |
| multitenant microservice can be im- | | | | | from the layout. This allows the | | |  |
| plemented using the Two-Step-View | | | | | flexible implementation of tenant- | | |  |
| pattern.6 This pattern facilitates the | | | | | specific user interface layouts using | | |  |

the same user interface data (see Figure 6).

**Statefulness**

A stateful system produces outputs that depend on the state generated in previous interactions and conversa-tions. For example, an e-commerce application is stateful if it remembers previous shopping activities and past visits to the online shop.

Why Stateless Microservices Are Ideal

In many cases, stateless microser-vices are ideal because they exploit the benefits of cloud computing, such as on-demand elasticity, load balanc-ing, high availability, and high reli-ability through redundancy.12 Stateful microservices, on the other hand, re-quire a more complex logic for man-aging the state, are less scalable, and do not facilitate high availability and high reliability.13 For example, state information has to be made persistent and synchronized in hot standby con-figurations and recreated in failover scenarios.

Even without failure, statefulness leads to session affinity, which can decrease throughput and increase latency due to the need to wait for specific stateful instances. On the other hand, stateless microservices can decrease the average response time by distributing the load among the available microservices, with-out the need of complex state man-agement and state synchronization techniques.

In a typical microservice architec-ture (see Figure 7), service consumer requests are directed to a load bal-ancer that routes them to available microservice instances. System avail-ability is defined as

|  |  |  |  |
| --- | --- | --- | --- |
| *Availability* = | *MTBF* | , |  |
|  |  |
| *MTBF* + *MTTR* |  |  |

**66** **IEEE SOFTWARE** |W W W.C O M P U T E R .O R G / S O F T WA R E|@ I E E E S O F T WA R E

*Total requests*

where MTBF is the mean time be-tween failures and MTTR is the mean time to repair.14 Assuming that all microservice instances have a similar MTBF, the system avail-ability can be increased by reducing the MTTR. This can be achieved by allowing the load balancer to stop routing incoming consumer requests to failed microservice instances and to reroute them to operating in-stances instead. The load balancer is able to immediately route requests to already deployed redundant microser-vices only if these are stateless.

Stateless microservices also achieve a higher system reliability through re-dundancy. The reliability of a system is defined as the ratio between the number of successful responses to the total number of requests:14

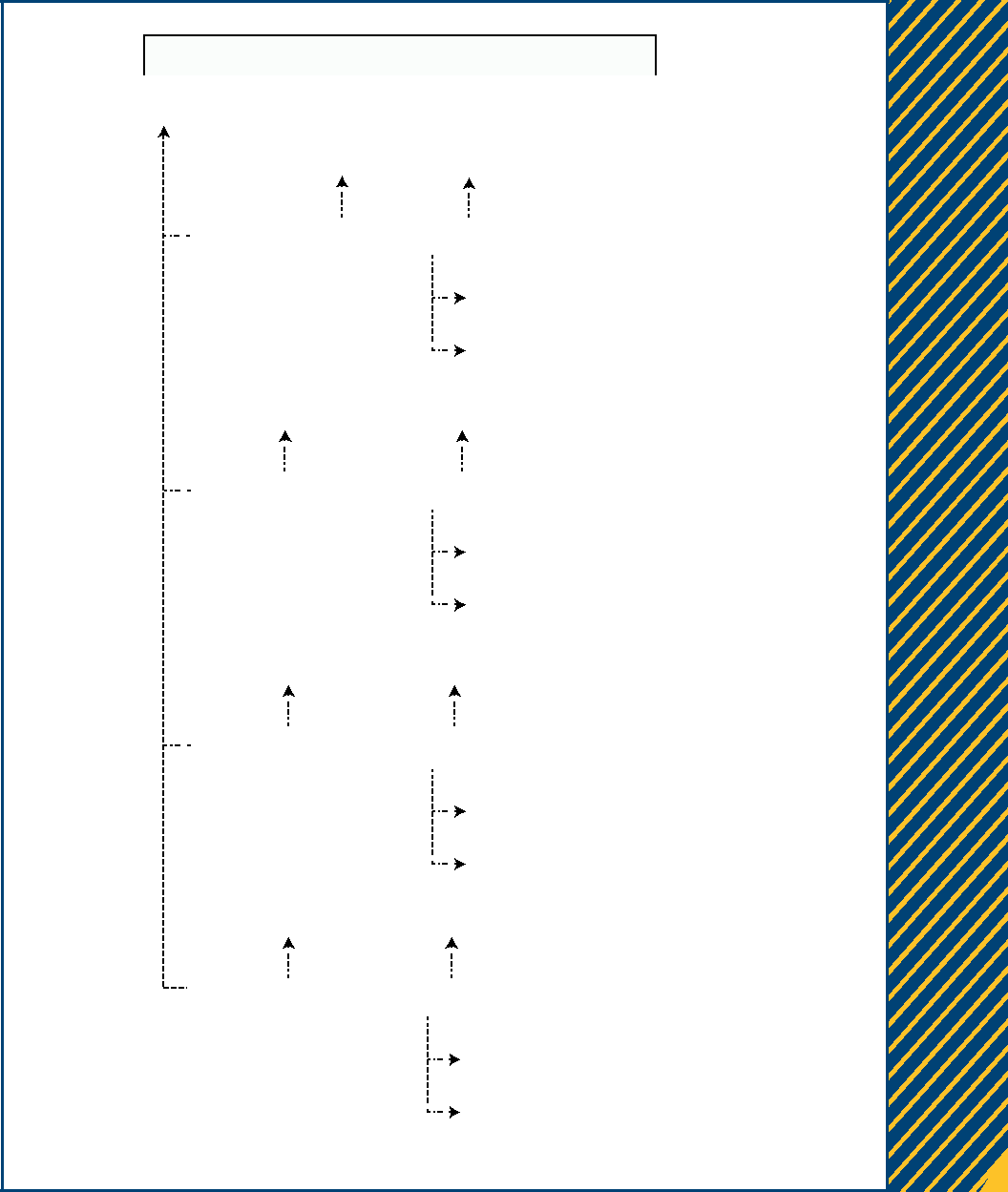
*Reliability* =

*Successful responses* ∗100% .

If the services are stateless, the load balancer can increase the probabil-ity of receiving a successful response from one of the available microser-vices by sending the same request to multiple available microservices, without the need to synchronize or replicate state in a session database.

Statefulness in Legacy Code

Multitenant View (with Two-Step-View Design Pattern)

**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
| DomainObjectData | | |  |  |  |  |  |  |  |  |
| read | |  | ReadDomainObjectScreen | | | | |  |  |  |
|  |  |  | create |  |  |  | read |  | |  |
|  | ReadDomainObjectStage1 | | |  | ReadDomainObjectStage2 | | | |  |  |
|  |  |  |  |  |  |  |  | |  |  |
|  |  |  |  |  |  |  | ListDomainObject HTML Tenant1 | | |  |
|  |  |  |  |  |  |  |  | | |  |
|  |  |  |  | create | |  | ListDomainObject HTML Tenant2 | | |  |
|  |  |  | |  |  |  |  |  |  |  |
|  |  | DeleteDomainObjectScreen | | | |  |  |  |  |  |
|  |  |  | |  |  |  |  | |  |  |
| read | | create | |  |  | read | | | |  |
|  | DeleteDomainObjectStage1 | | |  | DeleteDomainObjectStage2 | | | |  |  |
|  |  |  |  |  |  |  |  | |  |  |
|  |  |  |  |  |  |  | DeleteDomainObject HTML Tenant1 | | |  |
|  |  |  |  |  |  |  |  | | |  |
|  |  |  |  | create | |  | DeleteDomainObject HTML Tenant2 | | |  |
|  |  |  | |  |  |  |  |  |  |  |
|  |  | CreateDomainObjectScreen | | | |  |  |  |  |  |
|  |  |  | |  |  |  |  | |  |  |
| read | | create | |  |  | read | | | |  |
|  | CreateDomainObjectStage1 | | |  | CreateDomainObjectStage2 | | | |  |  |
|  |  |  |  |  |  |  |  | |  |  |
|  |  |  |  |  |  |  | AddDomainObject HTML Tenant1 | | |  |
|  |  |  |  |  |  |  |  | | |  |
|  |  |  |  | create | |  | AddDomainObject HTML Tenant2 | | |  |
|  |  |  |  |  |  |  |  |  |
|  |  | UpdateDomainObjectScreen | | | | |  |  |  |  |
|  |  |  | |  |  |  |  | |  |  |
| read | | create | |  |  | read | | | |  |
|  | UpdateDomainObjectStage1 | | |  | UpdateDomainObjectStage2 | | | |  |  |
|  |  |  |  |  |  |  | | |  |  |
|  |  |  |  |  |  | UpdateDomainObject HTML Tenant1 | | | |  |
|  |  |  |  |  |  |  | | | |  |
|  |  |  |  |  |  | UpdateDomainObject HTML Tenant2 | | | |  |
|  |  |  |  | create | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

FIGURE 6. A multitenant MVC view.

In the context of extracting mi-croservices, legacy source code is stateful if it is capable of retaining values (i.e., its state) between invoca-tions and is able to generate outputs that depend not only on the current input parameters but also on the pre-viously retained state (see Figure 8).

Statefulness in the context of microser-vices. When analyzing the stateful-ness of legacy code for the purpose of microservice extraction, the

inclusion or exclusion of state vari-able definitions (i.e., in-memory ses-sion state) determines whether or not the analyzed code is stateful with re-spect to a specific state variable.

Figure 8 depicts this in an ex-ample. State variable *s*1 is defined within the analyzed code, a second state variable *s*2 is defined outside the analyzed code (this could be either a session or domain state),

and a third domain state value *s*3 is stored in a database. Microservice option A includes only state variable *s*1; therefore, microservice A is state-ful only with respect to *s*1 (writing to *s*2 or *s*3 can be considered an out-put). Microservice option B is state-ful with respect to *s*1 and *s*2, and microservice option C (which in-cludes the database) is stateful with respect to *s*1, *s*2, and *s*3.

MAY/JUNE 2018 | **IEEE SOFTWARE** **67**

FOCUS: MICROSERVICES



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tenant |  | Service request | Microservice |  |
|  |  |  |
|  |  |  | instance 1 |  |
|  |  |  | MTBF |  |
|  | User | Service response |  |  |
|  | MTTR |  |  |
| User | Loadbalancer |  |  |
|  |  |  |  |
| User |  | Service request |  |  |
|  |  |  |  |
|  |  |  | Microservice |  |
|  |  | Service response | instance *N* |  |
|  |  | MTBF |  |
| User |  |  |  |  |

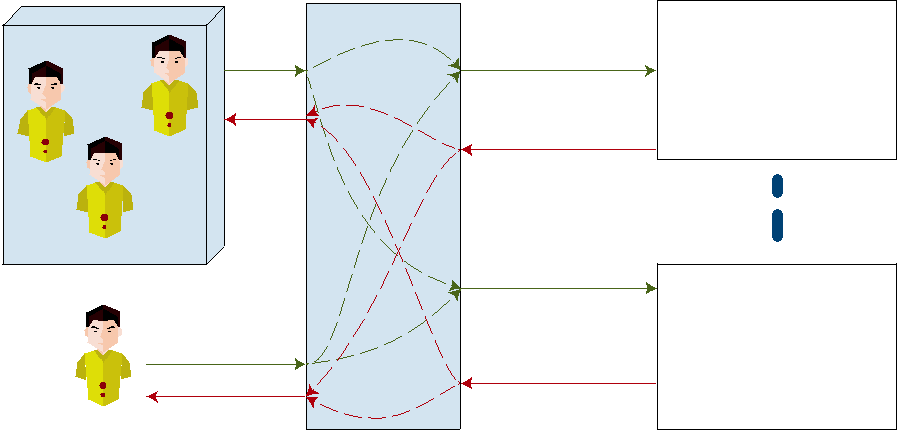
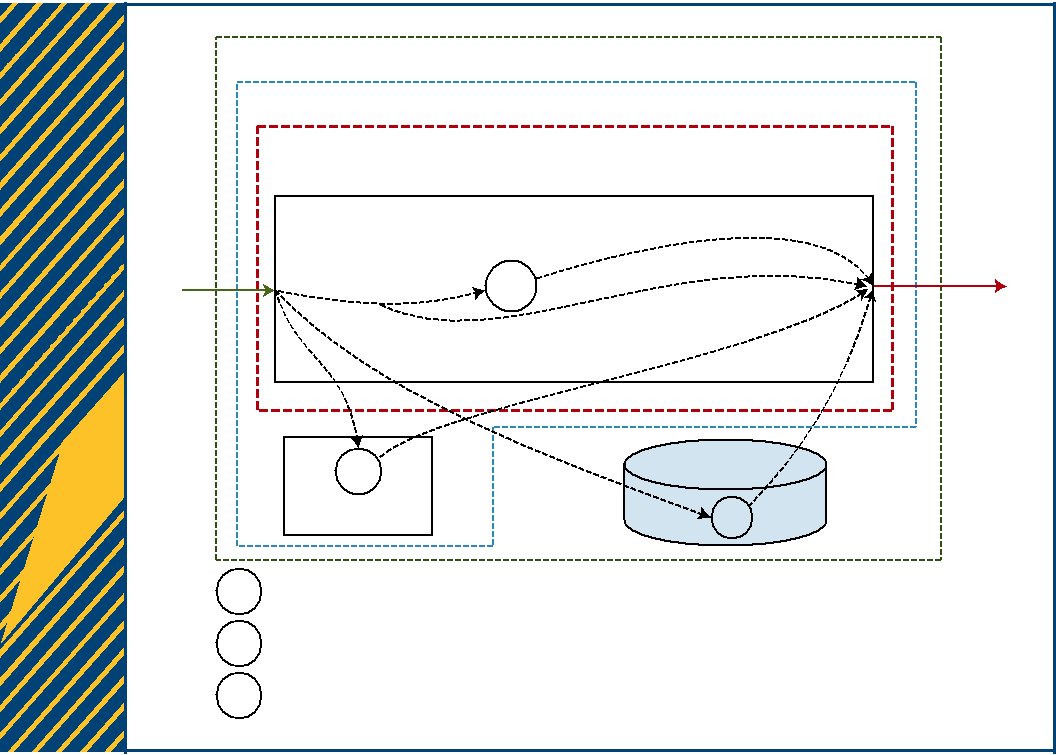


FIGURE 7. Availability and reliability through redundant (stateless) microservices.



Microservice option C

Microservice option B

Microservice option A

|  |  |  |
| --- | --- | --- |
|  | Legacy code |  |
| IN | OUT |  |
| *s*1 |  |

|  |  |
| --- | --- |
| *s*2 |  |
| External | *s*3 |

*s*1 Session state

*s*2 Session or domain state

*s*3 Domain state

FIGURE 8. Statefulness in the context of microservice extraction.

by invoking and returning values of other stateful procedures (see Figure 9, function p3).

The most common form of legacy code statefulness is found in objects (class instances). An object can be stateful by instantiating a class that implements stateful procedures or one with (state) properties (see Figure 10).

State properties can be static or nonstatic. Static properties lead to class-level statefulness that af-fects all instances of a class when modified (see Figure 10, class Stateful­ Static). In this case, the state can be changed only for all class in-stances at once, not for individ-ual ones. Nonstatic properties, on the other hand, affect individual class instances (see Figure 10, class StatefulNonStatic).

Stateful components. Stateful legacy components may include stateful procedures or stateful objects. For migrating such stateful components to microservices,12 it is important to distinguish between the microser-vice’s public API and its configura-tion API. The public API is accessible by service consumers, and therefore the ideal microservice is stateless with respect to the public API.

The microservice configuration API, on the other hand, allows the automatic deployment and automatic reconfiguration of microservice in-stances. For example, the configura-tion API may expose procedures for setting specific multitenancy-related properties, such as a database con-

Stateful procedures and objects. Legacy source code can be stateful at the procedure level, object level, or com-ponent level. A stateful procedure re-turns values that depend on variables whose lifetime exceeds the procedure execution; i.e., variables that are not reset between the invocations of the

procedure. In PHP, for instance, a stateful procedure can be imple-mented by declaring a static variable in a procedure (see Figure 9, function p1) or, as in most languages, by de-claring a global variable outside the procedure (see Figure 9, function p2). A procedure can itself become stateful

nection setting. Therefore, the con-figuration API does not necessarily need to be stateless.

Microservice Migration of Stateful Legacy Source Code

We observe that migrating legacy code to microservices is either a top-down,

**68** **IEEE SOFTWARE** |W W W.C O M P U T E R .O R G / S O F T WA R E|@ I E E E S O F T WA R E

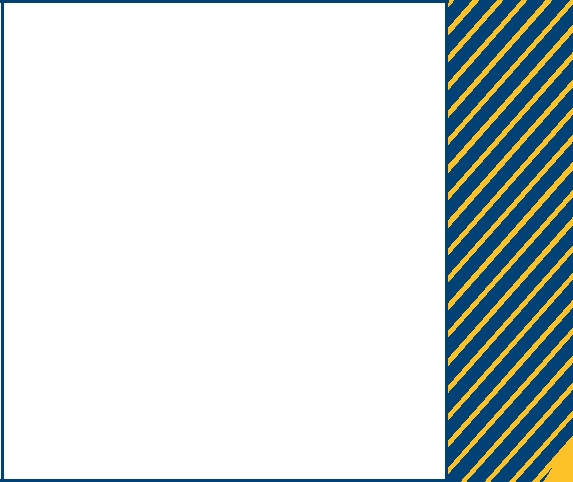
bottom-up, or meet-in-the-middle approach involving code-level refac-toring decisions15 of the legacy code and architectural decisions16,17 of the desired microservice solution. The following service-oriented ar-chitecture (SOA) patterns address statefulness and are applicable in microservice architectures.18

The Stateful Messaging pattern delegates internal state data to mi-croservice messages—i.e., the service request and service response. State-ful legacy procedures can be made stateless by replacing the local state variables with additional parameters

**Data Consistency**

Data consistency is a property of a distributed shared data storage sys-tem. It specifies the allowed behavior with respect to data access opera-tions. Sequential (single-threaded) legacy source code does not encoun-ter data consistency issues when accessing data in a repository, data-base, or shared memory. However, when migrating such sequential code to microservices, multiple instances of a microservice interact in parallel with a data repository, creating data consistency challenges.

function p1() {



static static state = 0;

static\_state++;

return static\_state;

}

glbl\_state ;

function p2() {

global glbl\_state ;

glbl state++;

return glbl\_state ;



}

function p3() {

return p1();

}

FIGURE 9. Examples of stateful procedures.

and returning values that allow set-ting and retrieving the state. These additional state parameters and return values are then linked to the microservice request and response, respectively. Stateful objects and components can also be refactored in the same way.

The Partial State Deferral pat-tern is an option if the microservice can remain partially stateful, and the goal is to reduce its memory consumption.

The State Repository pattern de-fers the state of a microservice to a dedicated state repository. By shar-ing the state repository among the microservice instances, these can be refactored to be entirely state-less for the purpose of increased availability and reliability. This pattern corresponds to option B in Figure 8.

The Stateful Service pattern defers the state of a microservice to a set of stateful utility services whose only purpose is to manage state information. This pattern iso-lates the problem of statefulness and defers it to other stateful ser-vices that have the same negative impact on scalability, availability, and reliability.

Identifying Data Consistency Issues in Legacy Code

Legacy source code often contains a mix of read/write operations from or to a data repository such as a data-base, file system, or other persistent storage. We observe that data con-sistency issues occur, however, when multiple instances of the same code (i.e., microservices) interact with a shared data repository. Therefore, before migrating such code to mi-croservices, data access operations should be grouped into read-only or read/write operations.

Read-only operations only re-trieve data from the data repository; they do not update, delete, or cre-ate new data. Read-only data is, for example, a multitenancy configura-tion setting that is read at start-up, and the local copy is never updated during the runtime of the microser-vice. Read/write operations up-date, delete, or create new data in a repository.

Restructuring Legacy Code into Data-Consistent Microservices

Microservices include their own data repository, such as a dedicated database instance, dedicated data-base schema in a shared database, or

class StatefulStatic {



private static state;



public static function setState( s) { self:: state = s;

}

public function getState() {

return self:: state;

}

}

class StatefulNonStatic {

private state;



public function setState( s) {

$this->state= s;

}

public function getState() {

return this->state;

}

}

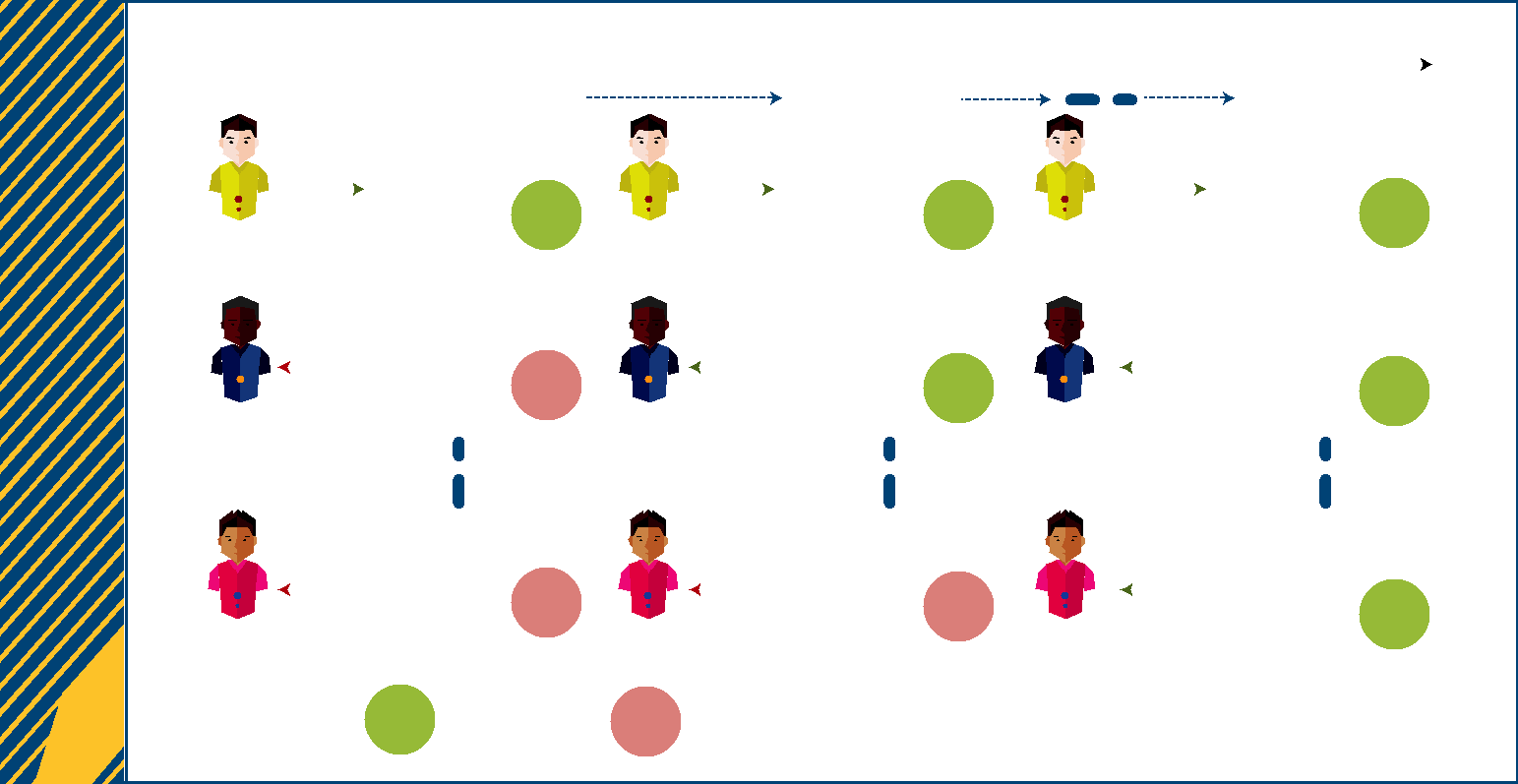
FIGURE 10. Stateful classes using a static state variable.

dedicated database tables. This leads to consistency challenges when the microservice data is synchronized with a centralized database.

The SOA pattern Service Data Replication replicates data in a ser-vice database.18 This pattern can be safely applied for read-only opera-tions. However, read/write opera-tions need additional data consistency checks.

MAY/JUNE 2018 | **IEEE SOFTWARE** **69**

FOCUS: MICROSERVICES



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | Timeline | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | Stage 1 | |  |  |  |  |  | Stage 2 |  |  |  |  |  | Stage *N* |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Write |  |  | Microservice |  |  |  | Write |  |  | Microservice |  |  | Write |  |  | Microservice |  |  |
|  |  |  |  | instance 1 |  |  |  |  |  | instance 1 |  |  |  |  | instance 1 |  |  |
| User A | | |  |  | State |  | User A | | |  |  | State |  |  |  |  |  | State |  |  |
|  |  |  |  |  |  |  | User A | | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Read |  |  | Microservice |  |  |  | Read |  |  | Microservice |  |  | Read |  |  | Microservice |  |  |
|  |  |  |  | instance 2 |  |  |  |  |  | instance 2 |  |  |  |  | instance 2 |  |  |
| User B | | |  |  | State |  | User B | | |  |  | State |  |  |  |  |  | State |  |  |
|  |  |  |  |  |  |  | User B | | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | Microservice |  |  |  |  |  |  | Microservice |  |  |  |  |  | Microservice |  |  |
|  |  | Read |  |  | instance *N* |  |  |  | Read |  |  | instance *N* |  |  | Read |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | instance *N* |  |  |
| User Z | | |  |  | State |  |  |  |  |  |  | State |  |  |  |  |  | State |  |  |
|  |  |  |  |  |  |  |  |  |  | User Z | | |  |  |  |  |
|  |  |  |  | User Z | | |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | State Consistent state | | State Inconsistent state | | | | | |  |  |  |  |  |  |  |  |

FIGURE 11. The Eventual Consistency pattern. Inconsistencies are corrected through multistage synchronization of the replicas.

The cloud-computing patterns Strict Consistency and Eventual Consistency address the consis-tency problem in cloud storage so-lutions.12 The Strict Consistency pattern allows a variable number of data replicas to be read from and written to. For example, in a system consisting of *n* data replicas, a write operation might access *w* replicas, while a read operation accesses *r* replicas. For each operation it is ensured that *n* , *w* 1 *r*, and strict consistency is guaranteed through the number of read or written rep-licas by making sure that each op-eration accesses at least one most current data version.12

The Eventual Consistency pattern is applied for unreliable or limited-bandwidth networks, or high data

volume, where simultaneously ac-cessing multiple replicas is not feasible12,19,20 (see Figure 11). In-stead, only one replica is accessed for read/write operations, result-ing in temporary data inconsis-tencies for the benefit of increased availability and performance. The inconsistencies are eventually cor-rected through synchronization of the data replicas. Data inconsis-tencies can persist if data entries in different replicas have been modified at the same time. In such a case, rules specify how such is-sues are resolved—for example, by simply dropping one of the modified versions.12 We are of the view that this rule works only in

1. non-data-critical system, while a critical enterprise system would

require a rollback or compensation activities.

**W**e have described threebasic challenges for mi-grating legacy code to

microservices. A best-practice solu-tion is to develop a microservice it-eratively, focusing on

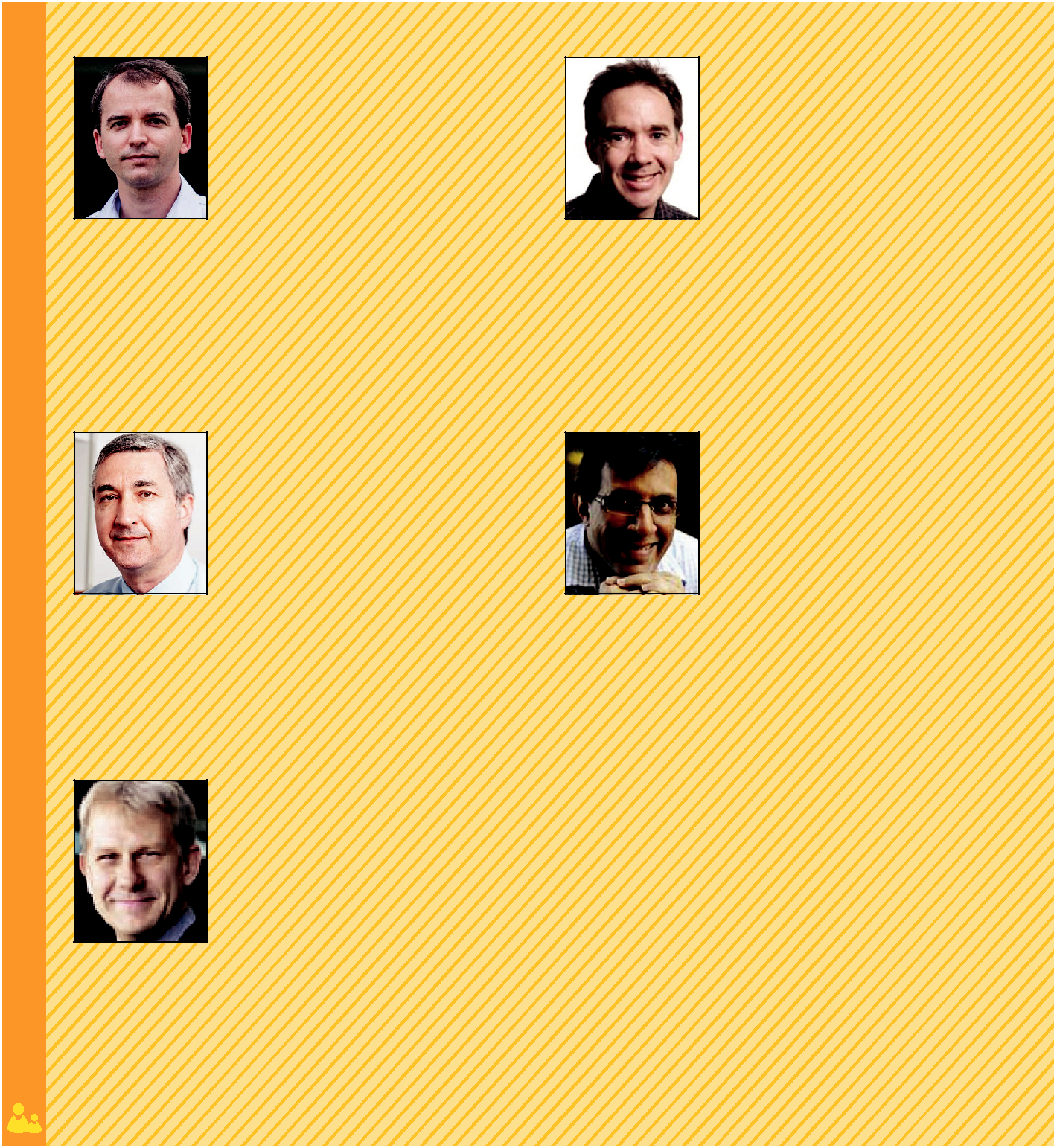
* eliminating statefulness from the extracted legacy code,
* implementing multitenancy functionalities, and
* solving potential new introduced data consistency challenges.

It is important to note that while these three challenges are interre-lated, they are created by different

**70** **IEEE SOFTWARE** |W W W.C O M P U T E R .O R G / S O F T WA R E|@ I E E E S O F T WA R E

|  |
| --- |
| ABOUT THE AUTHORS |

**ANDREI FURDA** is a lecturer in theQueensland University of Technology’s Science and Engineering Faculty, where he teaches enterprise systems and mobile-app development. His research interests include the modernization of legacy soft-ware systems for cloud computing. Furda received a PhD from Griffith University’s School of Engineering. Contact him at andreifurda@gmail.com.



**COLIN FIDGE** is a full professor in theQueensland University of Technology’s Science and Engineering Faculty, where he teaches computing fundamentals. His research interests include safety-critical, security-critical, and mission-critical soft-ware engineering. Fidge received a PhD in computer science from the Australian National University. Contact him at c.fidge@qut.edu.au.

**OLAF ZIMMERMANN** is a professorand institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil. His areas of interest include web-based application and integration architectures, service-oriented architec-ture and cloud design, and architectural knowledge management. Zimmermann received a doctorate in computer science from the University of Stuttgart. He’s on the *IEEE Software* editorial board.

**WAYNE KELLY** is a senior lecturer at theQueensland University of Technology’s School of Electrical Engineering and Com-puter Science, where he teaches enterprise software architecture, software engineer-ing, and compilers. His research interests include static analysis of program code, particularly for parallelization and detecting dataflow security violations. Kelly received a PhD in computer science from the Univer-sity of Maryland, College Park. Contact him at w.kelly@qut.edu.au.

**ALISTAIR BARROS** is a professor andthe Head of Services Science Discipline at the Queensland University of Technol-ogy’s School of Information Systems.

His research interests are software and enterprise architecture, business process management, technical software, and service analysis methods. Barros received a PhD in computer science from the University of Queensland. Contact him at alistair.barros@qut.edu.au.

types of requirements. Multitenancy is driven by the economic advantages resulting from sharing resources and infrastructure, while taking

into consideration the tenants’ data privacy needs. Statefulness is a characteristic that influences non-functional requirements with respect

to scalability, reliability, and avail-ability of the modernized system. Finally, data consistency is a func-tional requirement that defines the

MAY/JUNE 2018 | **IEEE SOFTWARE** **71**

FOCUS: MICROSERVICES



correct operation of the system and ranges from relaxed eventually con-sistent systems to strictly consistent ones. 

**Acknowledgments**

This research was supported in part by ARC-DP grant DP140103788.

**References**

1. O. Zimmermann, “Microservices Tenets,” *Computer Science—* *Research and Development*, vol. 32,nos. 3–4, 2017, pp. 301–310.
2. C. Pautasso et al., “Microservices in Practice, Part 1: Reality Check and Service Design,” *IEEE Software*, vol. 34, no. 1, 2017, pp. 91–98.
3. I. Gorton, “Hyper Scalability—the Changing Face of Software Architec-ture,” *Software Architecture for Big* *Data and the Cloud*, Elsevier, 2017,pp. 13–31.
4. S. Johann, “Dave Thomas on In-novating Legacy Systems,” *IEEE* *Software*, vol. 33, no. 2, 2016, pp.105–108.
5. J. Kabbedijk et al., “Defining Multi-tenancy: A Systematic Mapping Study on the Academic and the Industrial Perspective,” *J. Systems and Soft-ware*, Feb. 2015, pp. 139–148.
6. A. Furda et al., “Reengineering Data-Centric Information Systems for the Cloud—a Method and Architectural Patterns Promoting Multitenancy,” *Software Architecture for Big Data and the Cloud*, Elsevier, 2017, pp. 227–251.
7. F. Chong, G. Carraro, and R. Wolter, “Multi-tenant Data Architecture,” Microsoft, 2006; ramblingsofraju

.com/wp-content/uploads/2016/08 /Multi-Tenant-Data-Architecture.pdf.

1. V. Narasayya et al., “SQLVM: Per-formance Isolation in Multi-tenant Relational Database-as-a-Service,” *Proc. 6th Biennial Conf. Innovative Data Systems Research* (CIDR 13),2013.
2. R. Taft et al., “STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments,” *Proc. 7th ACM Symp. Cloud Com-puting*, 2016, pp. 388–400.
3. S. Walraven, E. Truyen, and W. Joosen, “Comparing PaaS Offer-ings in Light of SaaS Development,” *Computing*, vol. 96, no. 8, 2014, pp.669–724.
4. M. Fowler, *Patterns of Enterprise* *Application Architecture*, Addison-Wesley, 2003.
5. C. Fehling et al., *Cloud Computing* *Patterns: Fundamentals to Design, Build, and Manage Cloud Applica-tions*, Springer, 2014.
6. M.F. Gholami et al., “Cloud Migra-tion Process—a Survey, Evaluation Framework, and Open Challenges,” *J. Systems and Software*, Oct. 2016,pp. 31–69.
7. E. Bauer and R. Adams, *Reliability* *and Availability of Cloud Comput-ing*, Wiley–IEEE Press, 2012.
8. M. Fowler et al., *Refactoring: Im-proving the Design of Existing Code*,Addison-Wesley, 1999.
9. O. Zimmermann, “Architectural Refactoring: A Task-Centric View on Software Evolution,” *IEEE Software*, vol. 32, no. 2, 2015, pp. 26–29.
10. O. Zimmermann et al., “Archi-tectural Decision Guidance across Projects—Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge,” *Proc.* *12th Working IEEE/IFIP Conf. Soft-ware Architecture* (WICSA 15), 2015.
11. T. Erl, *SOA Design Patterns*, Prentice Hall, 2009.
12. M. Fowler, “Eventual Consistency,” *Microservice Trade-Offs*, 2015;martinfowler.com/articles /microservice-trade-offs.html #consistency.
13. W. Vogels, “Eventually Consistent,” *Comm. ACM*, vol. 52, no. 1, 2009,pp. 40–44.

SUBMIT

**TODAY**

**IEEE** TRANSACTIONS ON

BIG DATA

**SUBSCRIBE AND SUBMIT**

For more information on paper submission, featured articles, calls for papers, and subscription

links visit:

www.computer.org/tbd

*TBD* is financially cosponsoredby IEEE Computer Society, IEEE Communications Society, IEEE Computational Intelligence Society, IEEE Sensors Council, IEEE Consumer Electronics Society, IEEE Signal Processing Society, IEEE Systems, Man & Cybernetics Society, IEEE Systems Council, IEEE Vehicular Technology Society

*TBD* is technically cosponsored byIEEE Control Systems Society, IEEE Photonics Society, IEEE Engineering in Medicine & Biology Society, IEEE Power & Energy Society, and IEEE Biometrics Council

**72** **IEEE SOFTWARE** |W W W.C O M P U T E R .O R G / S O F T WA R E|@ I E E E S O F T WA R E