

Web Services and Web Development: Serverless

Tom Hinton 1809715

Table of Contents

Introduction	2
Serverless	3
Definition	3
Benefits of Serverless.....	3
Drawbacks of Serverless	4
Circuit-Breaker Pattern	6
Definition	6
Benefits of the Circuit-Breaker Pattern	7
Drawbacks of the Circuit-Breaker Pattern	7
Conclusion.....	7
References	8

Introduction

In this report, a brief overview of Serverless technologies is attempted. The advantage of avoiding server update and maintenance costs is discussed and the benefit of avoiding the difficulties of manual server configuration is noted. The high availability of Serverless is described. The scalability of Serverless applications is discussed and the advantage of resource efficiency is mentioned along with the disadvantage and potential dangers of cold starts. Possible solutions to the cold start problem are explored. The circuit breaker pattern is defined. Its overall structure and function is discussed, including its states and self-reopening functionality. The rationale for its implementation is covered along with its advantages. The considerations a developer must be aware of when implementing the circuit breaker pattern are also discussed.

Serverless

Definition

Serverless applications are run on servers which cloud service companies provide for the developer. Resources within the cloud-run servers are provisioned and released based on server load, allowing the application to scale up to meet demand and also scale down to save on costs. The developer is billed for an amount based on the resources that are used in their application, as opposed to dedicated servers which have set upkeep costs. [1] Another important term in Serverless technology is 'Function-as-a-service' (FaaS) which executes logic but does not have persistent data stores. [2]

Serverless runs by writing functions which are then called by predefined triggering conditions. These can then connect to cloud-based data storage in order to have the full functionality of a locally-run application. This encourages developers to take a more microservice-oriented approach as opposed to monolithic structures. [3] A common hurdle for Serverless development is that developers must become comfortable with their functions being run in stateless containers. A feature of Serverless is that instances are created to serve requests then removed in times of low traffic. For this reason, data cannot be assumed to persist natively between runs of a function since its container may have been closed and reopened.

Benefits of Serverless

The lack of server management in Serverless applications is a quality of life improvement for developers. Normally, the job of managing and maintaining a large set of servers for an application is arduous. For example, it was necessary to create a tool to track difficult to detect failures in server configurations as otherwise these errors are not apparent. [4] Similarly, a hill-climbing algorithm was created by a different team for the same purpose. [5] The removal of the necessity of configuring and provisioning servers can result in applications that are easier to work with.

Serverless applications charge the owner only when they are used, not while they are idle. This can cut down on costs. [6] In addition, the costs a developer would normally have to pay to set up rack space and maintain physical servers, as well as software licences and maintenance costs are avoided. The combined savings make Serverless a compelling choice for new companies and organisations that lack the capital to invest significant amounts into their projects.

Another benefit of Serverless is its elastic nature. A Serverless application will automatically create new instances of itself to field larger amounts of traffic (known as scaling up), allowing the application to handle almost arbitrarily large amounts of requests. [7] A Serverless application can also terminate instances in low traffic times such that none are running, thus ensuring greater efficiency. There is however, a drawback related to this scaling down which will be discussed later.

One powerful feature of Serverless is its high availability. Cloud service providers often make use of architecture that can facilitate multiple physically separated servers in the eventuality that racks become faulty or, in the worst case, to guard against natural disasters. For example, AWS's Lambda service runs on multiple data centres that are organised into Availability Zones, defined by the geographical area the centres are located in. A more thorough explanation can be found on the Amazon website [8]. Microsoft Azure uses a similar technique to ensure uptime as shown on their informational site [9]. This protects developers from unexpected loss of service due to faults with their physical servers.

Drawbacks of Serverless

While Serverless applications have many advantages that have allowed them to rise in popularity in web design, there are some issues with the technology that developers must be aware of before committing their project to a cloud provider.

The nature of Serverless apps involves a significant amount of the implementation and infrastructure being partially handled by the provider. This inevitably means the functions that are written by the developer are in danger of needing to use vendor-exclusive APIs and features. This becomes

problematic if the company needs to change providers as large swaths of architecture may need to be refactored to function with the new provider. [10]

Serverless applications may be resilient due to their Availability zones but there are security concerns. All sensitive data in the application is being handled by the cloud provider. For some projects this may constitute an unacceptable breach in security. On a similar note due to the large amount of automation in Serverless, developers have less control over the way their application is run.

Serverless applications are commonly perceived to be cheaper on the whole than maintaining a local server [6]. However, the payment model is erratic. An application can experience periods of high traffic without warning, which will need to be paid for. This spike in costs cannot be foreseen and is therefore difficult to budget for, unlike a traditional server which has monthly upkeep costs with an occasional need for upgrades or replacements.

While not an innate weakness of the platform, Serverless is still a young service and the way it operates is not typical of other software development platforms. Relearning how to create architecture using functions may take time and the newness of the field implies less community support may be available than for traditional servers.

A frequently discussed problem in Serverless applications is the “cold start” problem [3]. The feature of scaling down to zero is an excellent means of saving on resource costs but comes with an inherent drawback. The first request made to a currently idle Serverless application will cause it to scale up to one instance, which will require instantiation of a container to run the required functions and return the output to the user. This process has a noticeable delay that can take several seconds. This delay was documented in a blog post investigating the relationships between cold start times and various possible factors. [11] While this may be a small wait time for a user it may have a greater impact on processes running on other machines and this delay can cascade into other cold starts, the culmination of which could ultimately be to cause timeouts.

Most of the responsibility for mitigating cold start times lies with the cloud service provider as it is tied directly to the speeds of their infrastructure. [12] However, avoiding statically-typed languages has been known to decrease cold start response times and a workaround can be to build a timed function to stimulate the application and always keep one container active. This is more expensive but may be desirable if prompt response times are critical.

Circuit-Breaker Pattern

Definition

A circuit breaker pattern in software development is similar in concept to the physical circuit breakers used in electronic and electrical circuitry. In software, a function is first wrapped in a 'breaker'. [13] The function passes requests to the breaker which then sends them to the remote object. It also tracks how many requests fail and measures that data against a trigger condition. If the number of failures exceeds the trigger, the breaker activates and prevents any further calls to that object. It immediately responds to its function with an error, without needing a remote call, if the function attempts to reach the object. The name is derived from the analogous electrical circuit breaker which disables local power if it detects abnormal current. [14]

There are three states of a circuit breaker. In the 'closed' state the breaker allows requests to pass through it and reach the remote entity. In the 'open' state the breaker has been tripped and is returning errors when requests are made by the function to the remote entity. There is a third state which is unique to the design pattern: when a physical circuit breaker is tripped it must be reset manually. However in software engineering the breaker pattern can be built with the ability to send out its own test calls to its remote object to see if it is responsive. If it receives a response it will reopen itself and allow requests through. This state of probing for aliveness is called 'half-open'.

Benefits of the Circuit-Breaker Pattern

Sometimes in computing it is desirable to have a process exit quickly or gracefully. An automatic database call within a software system, for example, may end up failing or timing out multiple times. This can cause a severe buildup of delays, cascading system failures and substantial queueing of jobs. The circuit-breaker pattern's function is to help remedy this by allowing a remote connection to be considered "broken" without needing to attempt to access it. [13] Breakers can also be useful monitoring tools. A breaker being tripped usually indicates some large-scale system failure. Notifications can be attached to the breaker tripping, which can be sent to users as well as other software in the system.

Drawbacks of the Circuit-Breaker Pattern

A circuit breaker is a powerful diagnostic and resilience tool in a system but it needs to be used correctly. The tripping condition must be chosen with care as a breaker being activated erroneously will impede system flow unnecessarily. Furthermore, the power of the pattern lies in its response to a failure. Care must be taken in choosing what a system does to handle a breaker tripping, for example a recommended film function may provide a default list as a backup.

Conclusion

Serverless is a cloud-based service that is designed to allow flexibility and scalability to developers for their projects in the form of functions and triggers. This has the advantage of lowering running costs as Serverless applications only charge the user when they are run. This is a marked improvement over traditional servers which must be maintained and updated, which has cost and time implications.

However there are also drawbacks to Serverless. The monetary cost is overall lower but also erratic and hard to plan around, which makes budgeting a challenge. The property of the provider taking control of various aspects of implementation can cause problems for the developer as it gives them

less access to features of their application and increases the amount of vendor-specific features used, limiting portability to a different provider.

The 'scale up, scale down' system in Serverless is what allows its payment model to function but carries with it the inherent weakness of the 'cold start' problem. This delay can cause system timeouts. This weakness can be mitigated using other functions but is largely left to the provider.

The circuit breaker pattern is a design pattern that performs the function of protecting calls to remote entities. Its purpose is similar to electrical circuit breakers, hence the name. It wraps a function and moderates the function's calls to an object. If the connection fails too many times, it automatically returns failures to the function without allowing any further attempts to be made. This has the benefits of protecting a system from becoming mired in a slow set of failures and suffering from queue overloads. However, the breaker is only as useful as the error handling attached to it.

References

- [1] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, Denver, 2016.
- [2] E. van Eyk, A. Iosup, S. Seif and M. Thömmes, "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures," in *First International Workshop on Serverless Computing (WoSC)*, Las Vegas, 2017.
- [3] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian et al, "Serverless Computing: Current Trends and Open Problems," *Research Advances in Cloud Computing*, pp. 1-20, 2017.
- [4] A. Whitaker, R. S. Cox and S. D. Gribble, "Configuration Debugging as Search: Finding the Needle

- in the Haystack," in *6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, 2004.
- [5] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia and L. Zhang, "A Smart Hill-Climbing Algorithm for Application Server Configuration," in *World Wide Web Conference*, New York, 2004.
- [6] M. Villamizar, O. Garcés, L. Ochoa, H. Castro et al, "Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservice Architectures," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, 2016.
- [7] A. Pérez, G. Moltó, M. Caballer and A. Calatrava, "Serverless Computing for Container-based Architectures," *Future Generation Computer Systems*, vol. 83, pp. 50-59, 2018.
- [8] Amazon Web Services, Inc., "Regions and Availability Zones - Amazon Elastic Compute Cloud," 10 April 2019. [Online]. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>.
- [9] Microsoft Corporation (MS), "What are Azure Availability Zones?," 10 April 2019. [Online]. Available: <https://docs.microsoft.com/en-us/azure/availability-zones/az-overview>.
- [10] S. K. Mohanty, G. PremSankar and M. Di Francesco, "An evaluation of open source serverless computing frameworks," in *IEEE 10th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nicosia, 2018.
- [11] A Cloud Guru, "Y. Cui," A Cloud Guru, 15 June 2017. [Online]. Available: <https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76> . [Accessed 10 April 2019].
- [12] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless Computing: An

Investigation of Factors Influencing Microservice Performance," in *IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, 2018.

[13] T. M. Nygard, *Release It!*, Dallas: Pragmatic Bookshelf, 2012.

[14] J. Carnell, *Spring Microservices in Action*, Greenwich: Manning Publications Co., 2017.

Web Services and Web Development:

RESTful Web Services

Tom Hinton 1809715

Contents

Introduction	13
Definition	13
Core Principles	13
Uniform resource identifiers.....	14
REST patterns and antipatterns	16
REST HTTP verbs	17
State and Statelessness.....	18
Conclusion.....	19
References	19

Introduction

The rapid growth of the World Wide Web necessitated development of a methodology that would allow extensibility while being consistent and human-readable. To meet these demands, the REST architectural style was developed [1]. Fielding and Taylor state that “REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations.” [2] Of the APIs in use on the Web, approximately 32% of them are RESTful [3].

Definition

Dedicated URLs over HTTP allow the creation of architectural frameworks that make data and functionality accessible. This architectural pattern can be known as REST. It features a request-response pattern where “the HTTP methods Post, Get, Put and Delete on a given resource are mapped to the respective CRUD operations Create, Read, Update, and Delete. Service responses contain the representation of the requested resource presented in CSV, JSON, XML, or similar formats.” [4]

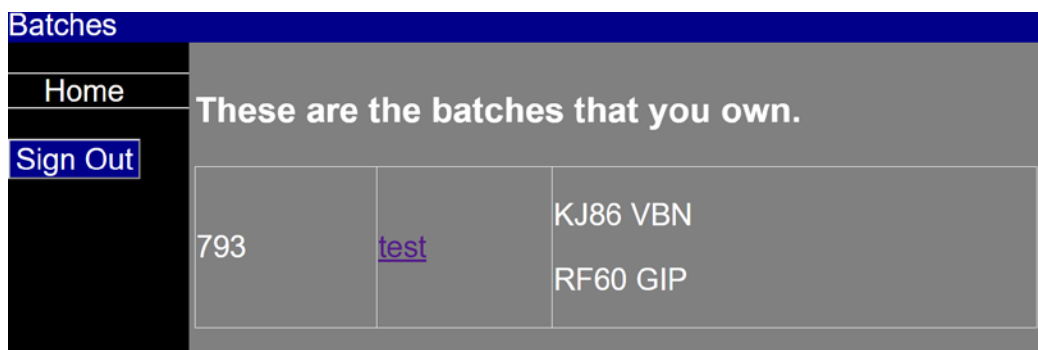
Core Principles

There are four main principles behind RESTful services [5]:

1. Resource addressability – APIs create and give access to resources. Each resource is addressable by a resource identifier.
2. Resource representations – Instead of being shown the resources themselves, clients work with representations that give the state of the resource they represent and also the changes the client can make to the resource.
3. Uniform interface – Access resources via HTTP requests must conform to a specific behaviour. The rules behind the requests will be covered later.

4. Hypermedia as the engine of state (HATEOAS) -also known as connectedness [6], is the property of being able to move between representations. Each representation should be linked to other representations via hyperlinks. A well-connected API has a path through the system entirely using links. A disconnected system requires manual assembly of URIs to access some of its resources. This important feature of REST makes it more human-readable.

Example of HATEOAS in the API of this project:



The buttons presented to the viewer are the possible state changes.

The links that a client is offered while on a representation are indications of what they are able to do with the resources at their disposal. This includes navigation and alteration. The name given to links of this type is “levers of state” [6], in that they show how to move to related states. Together these make up Representational State Transfer, or REST.

Uniform resource identifiers

The acronym ‘URI’ stands for Uniform Resource Identifiers; the three words together explain the functionality:

- *Uniform.* Allows resource identifiers of different types to be used in the same context regardless of the fact that their mechanisms may differ. Uniformity allows the names of many different resource identifiers to be interpreted in the same way. The presence of a uniform naming convention for resources makes it easier to add new names without

accidentally causing conflicts. In addition, an identifier that is appropriate in one context may also apply to others, generating a precedent for new URIs to follow.

- *Resource*. The resource that this refers to is in fact anything that the URI identifies. Examples of resources may include images, services, text documents and collections of other resources. Resources may also be abstract concepts such as mathematical operators and relationship types. Additionally, they can be entities that are not computational, such as people or companies.
- *Identifier*. This is the generic term for the means by which a resource can be differentiated from the other resources in the API. The way this is done varies by identifier. A resource can be found using its name, address, ID or other means. It is important to note that a URI may not access a resource that it identifies and an identifier may refer to multiple resources such as a set.

Some of the rules of naming URIs are as follows [7]:

- *Forward slashes* are used to show hierarchy. As an example, cars are a child resource of vehicles and would therefore be separated with a slash. A slash at the end of a URI is to be avoided as it offers no benefit and may make the URI more confusing.
- *Hyphens* can be used to denote spaces in URI names, such as “first-post”, but underscores are not allowed since they can be hidden by certain fonts, making them harder to select with a mouse. A singular noun is used for document names, a plural for collections and resource stores and a verb or verb phrase is used for controller names.

Example in the API of this project:

```
...vz BatchRepository.java x EntityManagerUtil.java x AuctionController.java x AuctionExpiration.java...
Source History
110 }
111
112 // (NOTE: ModelAndView)
113 @GetMapping("/auctions")
114 public ModelAndView auctionsPage(ModelMap model) {
115     String username = getUsername();
116     Integer userid = getUserId(username);
117     String isAdmin = isAdmin() ? "True" : "False";
118     ModelAndView mav = new ModelAndView("auctions", model);
119     LOG.info("Getting model and view");
120     List<Auction> auclist = AuctionUtil.getAllAuctions();
121     LOG.info("Grabbing auclist of size " + auclist.size());
122     mav.addObject("auclist", auclist);
123     mav.addObject("username", username);
124     mav.addObject("userid", userid);
125     mav.addObject("isAdmin", isAdmin);
126     LOG.info("Submitting list");
127
128     return mav;
129 }
130
131 // @GetMapping("/batches")
132 // @ResponseBody
```

REST patterns and anti-patterns

There are some recommended procedures, or design patterns, for RESTful applications. [8] Similarly there are anti-patterns, or common mistakes that need to be avoided when developing for REST.

Some examples of patterns and anti-patterns are discussed below.

- *Entity Linking* is a pattern that ensures that hyperlinks are set up between representations, in the response body or in the header. These allow for more natural state transfer and ensure high connectedness while keeping coupling of the states low as they are only connected via links. The anti-pattern of this is 'Forgetting Hypermedia', or not linking resources.
- *End point Redirection* is where endpoints can be configured to return status codes in the 300 range to send clients to an alternative service. This is a useful feature to have access to in case of issues with the primary service, or if the request made to the resource is invalid.
- *Content Negotiation* is a pattern that allows a resource to have different representations, possibly for different file formats. This makes the system more flexible since support for new outputs can be added more easily. When used, the pattern is loosely coupling. The anti-pattern of this is called the 'Ignoring MIME Types' anti-pattern.

REST HTTP Verbs

The methods available for each resource are accessed by using specific HTTP requests. For best practice in REST, each of the requests must behave in a certain way [9]. The form of the requests synergises with the form of the URIs, as it is a verb that is applied to plural nouns.

The GET method's function is to access a resource and return contents. The action of moving between pages on the Internet is a GET request. A GET request is known as 'safe': it does not change the current state. GET is also idempotent, which is the property that a single iteration has the same outcome as multiple iterations. This means calling GETs repeatedly will not change the function of the service. The server can make changes to the state as a result of a safe request but this would not be the responsibility of the client. An example of a state change by a safe request may be a visit counter.

The POST method creates a resource. It is recommended that POST parameters are held within the body and query parameters should not be used with POSTs. POST methods are not safe by definition as they write to the state and they are not idempotent as it may be desired to create multiple resources at once.

Example of a POST mapping in the API for this project:

```
@PostMapping("/auctions")
public ModelAndView makeAuction(ModelMap model,
    @ModelAttribute("auction") AuctionDto auction) {

    ModelAndView mav = new ModelAndView("auctions", model);

    Integer batchCount = auction.getBatchcount();
    Integer plateCount = auction.getPlatecount();
    Integer timeUp = auction.getTimeup();

    //Build up some empty boxes to fill
    Set<Batch> bset = new HashSet<>();
    Set<Plate> pset = null;

    // Make the batches
    for (int i=0;i<batchCount;i++) {
        // Reset the plate set from the last run
        pset = new HashSet<>();

        // Make the plates for a batch
        for (int j=0;j<plateCount;j++)
            pset.add(PlateUtil.createPlate());

        // Create the batch using the brand new plate set
        bset.add(BatchUtil.createBatch(pset));
    }
}
```

The PUT method is used to update a resource as opposed to creating a new one. A PUT request is idempotent – changing a resource to a new form repeatedly is the same as changing it once – but it

is not safe. The difference between PUT and POST is the URI that handles the request – POST acts on a collection where PUT is given a target resource name. It attempts to update the resource there, or creates a new one if it cannot find one. Both can be used for creation, but the other major difference in behaviour is PUT's idempotency, which may be desired for limiting resource creation. [9]

The DELETE method is used to remove a resource. Attempting to delete a resource multiple times has no effect and therefore the DELETE method is idempotent and not safe. DELETE requests return the same output if you call them repeatedly due to other instances of best practice [10].

The HEAD method is similar to the GET method except it returns only the HTTP headers and not content. HEAD is safe and idempotent.

Other elements of REST best practice are:

- *Pagination* – When returning a resource collection, by default the output can result in many elements, all clustered on one page. This can be difficult to read. It is therefore preferable to limit the number of results returned. The client should be told how many results exist in total and what offset through the results they occupy so that it is clear where in the resource they are. Setting a default page size if it is not specified can be wise.
- *Error handling* – If a query fails, a corresponding failure code should be returned to the client. The error body should contain a message, code and a description of the fault if there is one.

State and Statelessness

There are two types of state in REST. [6] The resource state stores information about the resources in the system and generally stays on the server. The client receives representations of resource states. The application state holds information on the path that the client has taken. This stays on the client until it is needed as part of a creational HTTP request, at which point it becomes resource state.

If the server never stores application state, the RESTful service is stateless. In a stateless service, each request is considered in isolation and as acting on the current resource state only. Any application state data that the client wants to be relevant must be submitted as part of the request, including authentication details.

Conclusion

Soon after it became evident that the World Wide Web would become ubiquitous, an architectural style to uniformly present data in a human-readable form became a necessity. REST was developed as a response to this need in that it, in a relatively simple way, provided a means to upload and locate many different types of data. In addition, REST is highly extensible in that it is not difficult to add new design features or upgrade existing ones.

In this report, the core principles of REST were described with reference to the number plate project. Uniform Resource Identifiers (URIs) were discussed with each character of the acronym being examined in turn. Common patterns and anti-patterns were also commented upon, as were HTTP verbs. The two types of state in RESTful APIs, resource and application, were covered and statelessness was defined.

References

- [1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures (PhD)," University of California, Irvine, 2000.
- [2] R. T. Fielding and R. T. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115-150, 2002.
- [3] M. Maleshkova, C. Pedrinaci and J. Domingue, "Investigating Web APIs on the World Wide

- Web," in *Eighth IEEE European Conference on Web Services* , Ayia Napa, 2010.
- [4] A. L. Lemos, F. Daniel and B. Benatallah, "Web Service Composition: A Survey of Techniques and Tools," *ACM Computing Surveys*, vol. 48, no. 3, pp. 33:1-41, 2015.
 - [5] C. Rodriguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali et al, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in *16th International Conference on Web Engineering (ICWE2016)*, Lugano, 2016.
 - [6] L. Richardson and S. Ruby, *RESTful Web Services*, Sebastopol: O'Reilly Media, 2007.
 - [7] M. Massé, *REST API Design Rulebook*, Sebastopol: O'Reilly Media, Inc, 2012.
 - [8] F. Palma, J. Dubois, N. Moha and Y.-G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-based Approach," in *12th International Conference on Service-Oriented Computing*, Paris, 2014.
 - [9] B. Mehta, *RESTful Java Patterns and Best Practices*, Birmingham: Packt Publishing, 2014.
 - [10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, RFC Editor, 1999.
 - [11] C. Pautasso, O. Zimmermann and F. Leymann, "RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision," in *17th International Conference on World Wide Web*, Beijing, 2008.