



GO

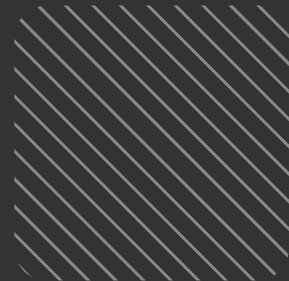
BOOTCAMP

Clase en vivo

//Go Web

IT BOARDING

BOOTCAMP



Objetivos de la clase:

- Comprender cómo implementar el método PUT en Go.
- Comprender cómo implementar el método PATCH en Go.
- Comprender cómo implementar el método DELETE en Go.

Índice



01 [Repaso](#)

02 [PUT en GO](#)

03 [PATCH en GO](#)

04 [DELETE en GO](#)

IT BOARDING

BOOTCAMP



1

Repaso

IT BOARDING

BOOTCAMP



PUT

IT BOARDING

BOOTCAMP





PUT [Utilidad y características]

Una solicitud **PUT** se utiliza para actualizar un recurso, reemplazándolo —en su totalidad— por otro recurso nuevo.

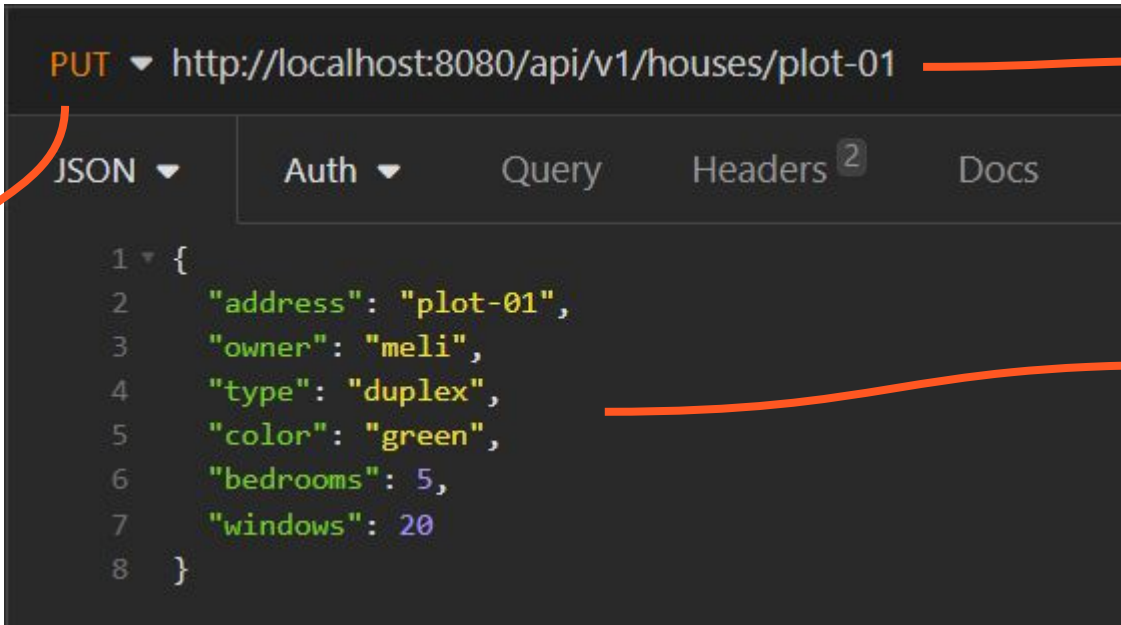
Una solicitud **PUT** siempre contiene un recurso completo. Esto es necesario, porque una cualidad de las solicitudes **PUT** es la *idempotencia*.



La *idempotencia* es la calidad de producir el mismo resultado, incluso, si la misma solicitud se realiza varias veces. No produce efectos secundarios.

Ejemplo simbólico haciendo un PUT

Así se visualiza una solicitud **PUT** para “poner” una casa prefabricada en lugar de otra ya existente:



The screenshot shows a REST client interface with a PUT request. The request is to the URL `http://localhost:8080/api/v1/houses/plot-01`. The request body is a JSON object. Annotations with orange arrows point to the **Method** (PUT), the **URL**, and the **Body** of the request.

Method → PUT

URL → `http://localhost:8080/api/v1/houses/plot-01`

Body →

```
1 {  
2   "address": "plot-01",  
3   "owner": "meli",  
4   "type": "duplex",  
5   "color": "green",  
6   "bedrooms": 5,  
7   "windows": 20  
8 }
```



PATCH

IT BOARDING

BOOTCAMP



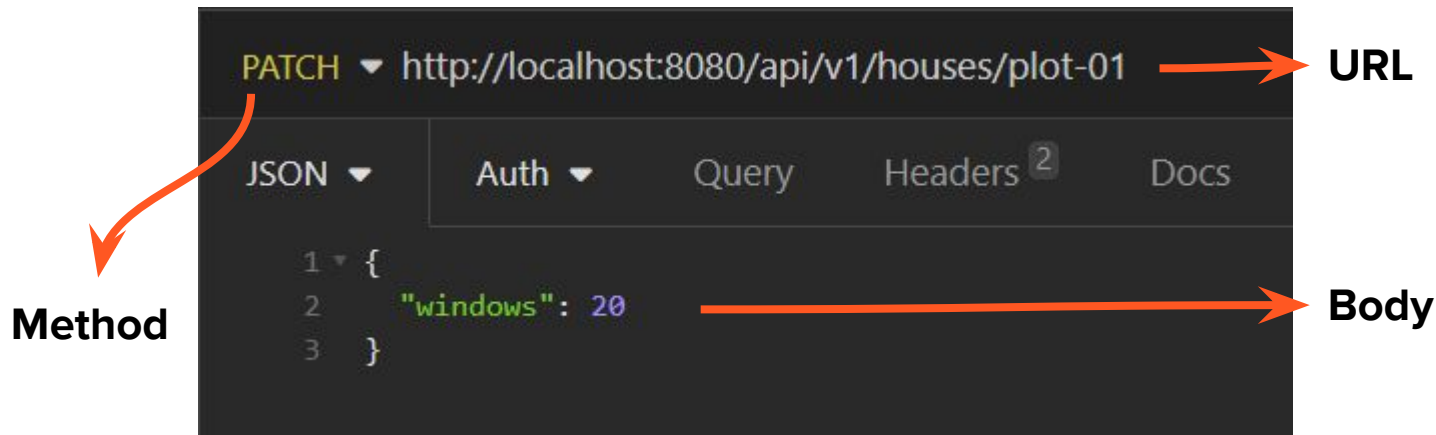


PATCH [Utilidad y características]

- Una solicitud **PATCH** se utiliza para enviar una representación **parcial** del recurso, con las **instrucciones** para aplicar las **modificaciones**.
- Es **idempotente**, pero puede no serlo.
- “Parchea” el recurso, cambiando sus propiedades.
- Se utiliza para realizar actualizaciones menores a los recursos.
- Una solicitud **PATCH** técnicamente puede contener un recurso completo, pero no es recomendado o su enfoque principal de uso.

Ejemplo simbólico haciendo un PATCH

Así se visualiza una solicitud **PATCH** para agregar una ventana en una casa ya existente:



DELETE

IT BOARDING

BOOTCAMP



// ¿Qué es y para qué sirve?

Una solicitud **DELETE** es un método HTTP que se utiliza para solicitar al servidor que elimine un recurso existente en la ubicación que indica en la URL.

Contiene la ubicación del recurso a eliminar (el **ID del recurso a eliminar**).

Es un método HTTP idempotente.

Ejemplo DELETE

Así se visualiza una solicitud **DELETE** para eliminar un recurso existente en el servidor mediante un **path parameter**:



En las solicitudes de tipo **DELETE**, no necesitamos enviar datos en el cuerpo (**body**) de las mismas. Sólo enviamos un **path parameter** que le indica al servidor el recurso a eliminar.

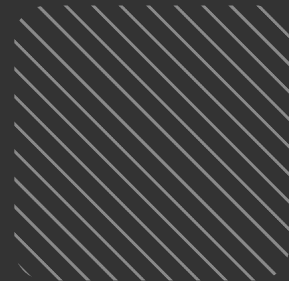


2

PUT en GO

IT BOARDING

BOOTCAMP



Petición/Respuesta

Al recibir una solicitud **PUT** con un **path parameter** se nos indica el *Id* del producto almacenado que se está intentando reemplazar.

Petición:

```
PUT http://localhost:8080/products/1
```

```
JSON Auth Query Headers 1
```

```
1 {
2   "name": "refrigerator",
3   "type": "home appliances",
4   "quantity": 2,
5   "price": 41500.75
6 }
```

Respuesta:

```
Preview Header 3 Cookie
```

```
1 {
2   "id": 1,
3   "name": "refrigerator",
4   "type": "home appliances",
5   "quantity": 2,
6   "price": 41500.75
7 }
```



Paso 1 - Agregar método en Interface

Aquí podemos ver el modelo o **schema** que utiliza nuestra **interface**

{}

```
// Product is a struct that contains the information of a product
type Product struct {
    Id      int
    Name    string
    Type    string
    Quantity int
    Price   float64
}
```



Paso 1 - Agregar método en Interface

En el package **storage**, agregamos el método **UpdateOrCreate** en la **interface Products**, el cual podremos utilizar en el método **PUT**, dónde se **actualiza** el recurso completo, o en caso de no existir, se **crea**.

```
{}
```

```
// Products is an interface that contains the methods of a storage of products
type Products interface {
    // Get returns all the products
    Get() (p []Product, err error)
    // GetByID returns a product by id
    GetByID(id int) (p *Product, err error)
    // Save saves a product
    Save(p *Product) (err error)
    // UpdateOrCreate updates or creates a product if it does not exist
    UpdateOrCreate(p *Product) (err error)
}
```



Paso 2 - Storage (Map Implementation)

Aquí tenemos nuestra implementación de storage in memory (map) para la interfaz **Products**.

```
// Product is a struct that implements the interface Products
type ProductAttributes struct {
    Name      string
    Type      string
    Quantity  int
    Price     float64
}

type ProductsMap struct {
    db      map[int]ProductAttributes
    lastId  int
}
```



Paso 2 - Storage (Map Implementation)

Se implementa la funcionalidad para actualizar el producto en memoria, en caso de no existir, se crea un nuevo recurso con el id manejado internamente por el storage.

{}

```
// UpdateOrCreate updates or creates a product if it does not exist
func (pm *ProductsMap) UpdateOrCreate(p *Product) (err error) {
    // serialize
    attr := ProductAttributes{Name: p.Name, Type: p.Type, Quantity: p.Quantity, Price: p.Price}
    // update
    _, ok := pm.db[p.Id]
    switch ok {
    case true:
        pm.db[p.Id] = attr
    default:
        pm.lastId++
        pm.db[s.lastId] = attr
    }
    return
}
```



Paso 3 - Handler

En el package **handler**, creamos una implementación para los **handlers** de **Products** y exponer el método UpdateOrCreate de **Products**

```
{}  
// Products is a struct that contains the methods of a handler of products  
type Products struct {  
    // st is the interface Products for storage operations  
    st storage.Products  
}
```



Paso 3 - Handler

Creamos un struct **RequestBody** struct para manejar el **http.RequestBody** de la petición. También parseamos el **ID**.

```
// UpdateOrCreate updates or creates a product if it does not exist, by id
type RequestBodyUpdateOrCreate struct {
    Name      string `json:"name"`
    Type      string `json:"type"`
    Quantity  int    `json:"quantity"`
    Price     float64 `json:"price"`
}

func (p *Products) UpdateOrCreate(w http.ResponseWriter, r *http.Request) {
    // request
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        code := http.StatusBadRequest
        body := map[string]any{"message": "invalid id", "data": nil}

        response.JSON(w, code, body)
        return
    }
}
```

{}



Paso 3 - Handler

Utilizando el package **request**, aplicamos un **decoding** desde el **Reader** **http.RequestBody** hacia la **struct RequestBodyUpdateOrCreate**

```
// UpdateOrCreate updates or creates a product if it does not exist, by id
type RequestBodyUpdateOrCreate struct {
    Name      string `json:"name"`
    Type      string `json:"type"`
    Quantity  int    `json:"quantity"`
    Price     float64 `json:"price"`
}

func (p *Products) UpdateOrCreate(w http.ResponseWriter, r *http.Request) {
    // request
    var reqBody RequestBodyUpdateOrCreate
    if err := request.JSON(r, &reqBody); err != nil {
        code := http.StatusBadRequest
        body := map[string]any{"message": "invalid request body", "data": nil}

        response.JSON(w, code, body)
        return
    }
}
```

{}



Paso 3 - Handler

Deserializamos el **struct RequestBodyUpdateOrCreate** hacia el **schema** que utiliza la **interface storage (Product)**

```
func (p *Products) UpdateOrCreate(w http.ResponseWriter, r *http.Request) {  
    // process  
    // -> deserialize  
    pr := storage.Product{  
        Id:      id,  
        Name:    reqBody.Name,  
        Type:    reqBody.Type,  
        Quantity: reqBody.Quantity,  
        Price:   reqBody.Price,  
    }  
    // -> update or create  
    if err := p.st.UpdateOrCreate(&pr); err != nil {  
        code := http.StatusInternalServerError  
        body := map[string]any{"message": "internal server error", "data": nil}  
  
        response.JSON(w, code, body)  
        return  
    }  
}
```

{}



Paso 3 - Handler

Finalmente devolvemos nuestra response final

```
{}  
  
func (p *Products) UpdateOrCreate(w http.ResponseWriter, r *http.Request) {  
    // response  
    code := http.StatusOK  
    body := map[string]any{"message": "product updated or created", "data": nil}  
  
    response.JSON(w, code, body)  
}
```



Paso 4 - Router

Dentro de la función **main**, creamos nuestro **router** y registramos el **endpoint** “**/products/{id}**” con método **PUT**, asociado al handler **UpdateOrCreate**

```
{ }
```

```
func main() {  
    // dependencies  
    db := make(map[int]storage.ProductAttributes)  
    st := storage.NewProductsMap(db, 0)  
    hd := handlers.NewProducts(st)  
    // server  
    rt := chi.NewRouter()  
    // -> routes  
    // -> -> products group  
    rt.Route("/products", func(rt chi.Router) {  
        // update or create  
        rt.Put("/{id}", hd.UpdateOrCreate)  
    })  
    // -> run  
    if err := http.ListenAndServe(":8080", rt); err != nil {  
        panic(err)  
    }  
}
```





3

PATCH en GO

IT BOARDING

BOOTCAMP



Petición/Respuesta

Se recibe una solicitud **PATCH** con un **path parameter** que indique el *id* del producto almacenado que se está intentando modificar, y se modifica solo el campo nombre.

Petición:

```
PATCH ▼ http://localhost:8080/products/2

JSON ▼ Auth ▼ Query Head
1 ▼ {
2   "name": "microwave"
3 }
```

Respuesta:

```
1 ▼ {
2   "id": 2,
3   "name": "microwave",
4   "type": "home appliances",
5   "quantity": 5,
6   "price": 15000.0
7 }
```



Paso 1 - Agregar método en Interface

Agregamos el método **Update** en la **interface Products**, el cual podremos utilizar en el método **PATCH**, dónde se **actualiza** el recurso parcialmente, pero actualizando completamente con valores antiguos previamente cargados y los nuevos parcheados

```
// Products is an interface that contains the methods of a storage of products
type Products interface {
    // Get returns all the products
    Get() (p []Product, err error)
    // GetByID returns a product by id
    GetByID(id int) (p *Product, err error)
    // Save saves a product
    Save(p *Product) (err error)
    // UpdateOrCreate updates or creates a product if it does not exist
    UpdateOrCreate(p *Product) (err error)
    // Update updates a product by id
    Update(id int, p *Product) (err error)
}
```



Paso 2 - Storage (Map Implementation)

Se implementa la funcionalidad para **actualizar** el producto en memoria, en **caso** de **no existir**, se retorna un **error**

```
{}
```

```
// Update updates a product by id
func (pm *ProductsMap) Update(id int, p *Product) (err error) {
    // search
    if _, ok := pm.db[id]; !ok {
        err = fmt.Errorf("%w: %d", ErrNotFound, id)
        return
    }
    // serialize
    attr := ProductAttributes{Name: p.Name, Type: p.Type, Quantity: p.Quantity, Price: p.Price}
    // update
    pm.db[id] = attr
    return
}
```



Paso 3 - Handler

Creamos un struct **RequestBody** struct para manejar el **http.RequestBody** de la petición. También parseamos el **ID**.

```
// Update updates a product by id
type RequestBodyUpdate struct {
    Name      string `json:"name"`
    Type      string `json:"type"`
    Quantity  int    `json:"quantity"`
    Price     float64 `json:"price"`
}

func (p *Products) Update(w http.ResponseWriter, r *http.Request) {
    // request
    id, err := strconv.Atoi(chi.URLParam(r, "id"))
    if err != nil {
        code := http.StatusBadRequest
        body := map[string]any{"message": "invalid id", "data": nil}

        response.JSON(w, code, body)
        return
    }
}
```



Paso 3 - Handler

Previo a aplicar el **decoding** desde **http.RequestBody** hacia el **struct RequestBodyUpdate**, primero debemos **cargarlo** con los **valores originales** del recurso.

{}

```
func (p *Product) Update(w http.ResponseWriter, r *http.Request) {  
    // process  
    // -> get searched product to reference old values  
    pr, err := p.st.GetByID(id)  
    if err != nil {  
        var code int; var body map[string]any  
        switch {  
        case errors.Is(err, storage.ErrNotFound):  
            code = http.StatusNotFound  
            body = map[string]any{"message": "product not found", "data": nil}  
        default:  
            code = http.StatusInternalServerError  
            body = map[string]any{"message": "internal server error", "data": nil}  
        }  
        response.JSON(w, code, body)  
        return  
    }  
}
```



Paso 3 - Handler

Una vez cargado los valores originales, aplicamos el **decoding**.

```
func (p *Products) Update(w http.ResponseWriter, r *http.Request) {  
    // process  
    // -> serialize (RequestBodyUpdate loaded with original values)  
    reqBody := RequestBodyUpdate{  
        Name:    pr.Name,  
        Type:    pr.Type,  
        Quantity: pr.Quantity,  
        Price:   pr.Price,  
    }  
    if err := request.JSON(r, &reqBody); err != nil {  
        code := http.StatusBadRequest  
        body := map[string]any{"message": "invalid request body", "data": nil}  
  
        response.JSON(w, code, body)  
        return  
    }  
}
```

{}



Paso 3 - Handler

Luego de aplicado el **decoding** en **RequestBodyUpdate**, conteniendo tanto los **valores originales**, así como los **nuevos** enviados por el cliente, aplicamos la **deserialización** y actualizamos.

```
func (p *Products) Update(w http.ResponseWriter, r *http.Request) {  
    // process  
    // -> deserialize  
    pr = &storage.Product{Id: id, Name: reqBody.Name, Type: reqBody.Type, Quantity: reqBody.Quantity, Price: reqBody.Price}  
    // -> update  
    err = h.st.Update(id, pr)  
    if err != nil {  
        var code int; var body map[string]any  
        switch {  
        case errors.Is(err, storage.ErrNotFound):  
            code = http.StatusNotFound  
            body = map[string]any{"message": "product not found", "data": nil}  
        default:  
            code = http.StatusInternalServerError  
            body = map[string]any{"message": "internal server error", "data": nil}  
        }  
        response.JSON(w, code, body)  
        return  
    }  
}
```

{}



Paso 3 - Handler

Finalmente devolvemos nuestra respuesta final.

```
{}  
  
func (p *Products) Update(w http.ResponseWriter, r *http.Request) {  
    // response  
    code := http.StatusOK  
    body := map[string]any{"message": "product updated", "data": nil}  
  
    response.JSON(w, code, body)  
}
```



Paso 4 - Router

Dentro de la función **main**, registramos el **endpoint** “/products/{id}” con método **PATCH**, asociado al handler **Update**.

```
{}
```

```
func main() {  
    // dependencies  
    db := make(map[int]storage.ProductAttributes)  
    st := storage.NewProductsMap(db, 0)  
    hd := handlers.NewProducts(st)  
    // server  
    rt := chi.NewRouter()  
    // -> routes  
    // -> -> products group  
    rt.Route("/products", func(rt chi.Router) {  
        // update  
        rt.Patch("/{id}", hd.Update)  
    })  
    // -> run  
    if err := http.ListenAndServe(":8080", rt); err != nil {  
        panic(err)  
    }  
}
```





4

DELETE en GO

IT BOARDING

BOOTCAMP



Petición/Respuesta

Se recibe una solicitud **DELETE** con un **path parameter** que indica el *Id* del producto almacenado que se está intentando eliminar.

Petición:

- No necesita tener datos en su **body**.
- Solo precisa indicar el “/d” del producto que se desea eliminar para identificarlo correctamente.
- Se pasará como parámetro de ruta.

Respuesta:

Si al procesar la petición, el servidor encuentra el “/d” especificado, elimina el producto y devuelve un status 2xx.



La respuesta, en su **body**, puede contener un detalle actualizado de los productos para constatar que el indicado fue eliminado.



Paso 1 - Agregar método en Interface

Agregamos el método **Delete** en la **interface Products**, el cual podremos utilizar en el método **DELETE**, dónde se **elimina** el recurso buscado.

```
// Products is an interface that contains the methods of a storage of products
type Products interface {
    // Get returns all the products
    Get() (p []Product, err error)
    // GetByID returns a product by id
    GetByID(id int) (p *Product, err error)
    // Save saves a product
    Save(p *Product) (err error)
    // UpdateOrCreate updates or creates a product if it does not exist
    UpdateOrCreate(p *Product) (err error)
    // Update updates a product by id
    Update(id int, p *Product) (err error)
    // Delete deletes a product by id
    Delete(id int) (err error)
}
```



Paso 2 - Storage (Map Implementation)

Se implementa la funcionalidad para **eliminar** el producto en memoria, en **caso** de **no existir**, se retorna un **error**.

```
{}
```

```
// Delete deletes a product by id
func (pm *ProductsMap) Delete(id int) (err error) {
    // search
    if _, ok := pm.db[id]; !ok {
        err = fmt.Errorf("%w: %d", ErrNotFound, id)
        return
    }
    // delete
    delete(pm.db, id)
    return
}
```



Paso 3 - Handler

Parseamos el **ID** del recurso que se busca **eliminar**.

```
{  
  // Delete deletes a product by id  
  func (p *Products) Delete(w http.ResponseWriter, r *http.Request) {  
    // request  
    id, err := strconv.Atoi(chi.URLParam(r, "id"))  
    if err != nil {  
      code := http.StatusBadRequest  
      body := map[string]any{"message": "invalid id", "data": nil}  
  
      response.JSON(w, code, body)  
      return  
    }  
  }  
}
```



Paso 3 - Handler

Procedemos a **eliminar** el recurso a través del método Delete de **storage**.

```
func (p *Products) Delete(w http.ResponseWriter, r *http.Request) {  
    // process  
    // -> delete  
    err = h.st.Delete(id)  
    if err != nil {  
        var code int; var body map[string]any  
        switch {  
        case errors.Is(err, storage.ErrNotFound):  
            code = http.StatusNotFound  
            body = map[string]any{"message": "product not found", "data": nil}  
        default:  
            code = http.StatusInternalServerError  
            body = map[string]any{"message": "internal server error", "data": nil}  
        }  
        response.JSON(w, code, body)  
        return  
    }  
}
```



Paso 3 - Handler

Finalmente devolvemos nuestra respuesta final. Un **204** sin body. En este caso el package **response** si recibe **nil** en el **body**, solo setea el **code**.

```
func (p *Products) Delete(w http.ResponseWriter, r *http.Request) {  
    // response  
    code := http.StatusOK  
    body := map[string]any(nil)  
  
    response.JSON(w, code, body)  
}
```



Paso 4 - Router

Dentro de la función **main**, registramos el **endpoint** “/products/{id}” con método **DELETE**, asociado al handler **Delete**.

{ }

```
func main() {  
    // dependencies  
    db := make(map[int]storage.ProductAttributes)  
    st := storage.NewProductsMap(db, 0)  
    hd := handlers.NewProducts(st)  
    // server  
    rt := chi.NewRouter()  
    // -> routes  
    // -> -> products group  
    rt.Route("/products", func(rt chi.Router) {  
        // update  
        rt.Delete("/{id}", hd.Delete)  
    })  
    // -> run  
    if err := http.ListenAndServe(":8080", rt); err != nil {  
        panic(err)  
    }  
}
```



Live Coding





Conclusiones

En esta clase aprendimos los conceptos teóricos de las peticiones **HTTP PUT, PATCH y DELETE**.

Además aprendimos cómo aplicarlas en GO para hacer que nuestras aplicaciones puedan interactuar correctamente con sus usuarios.



Actividad





Gracias.

IT BOARDING

BOOTCAMP

